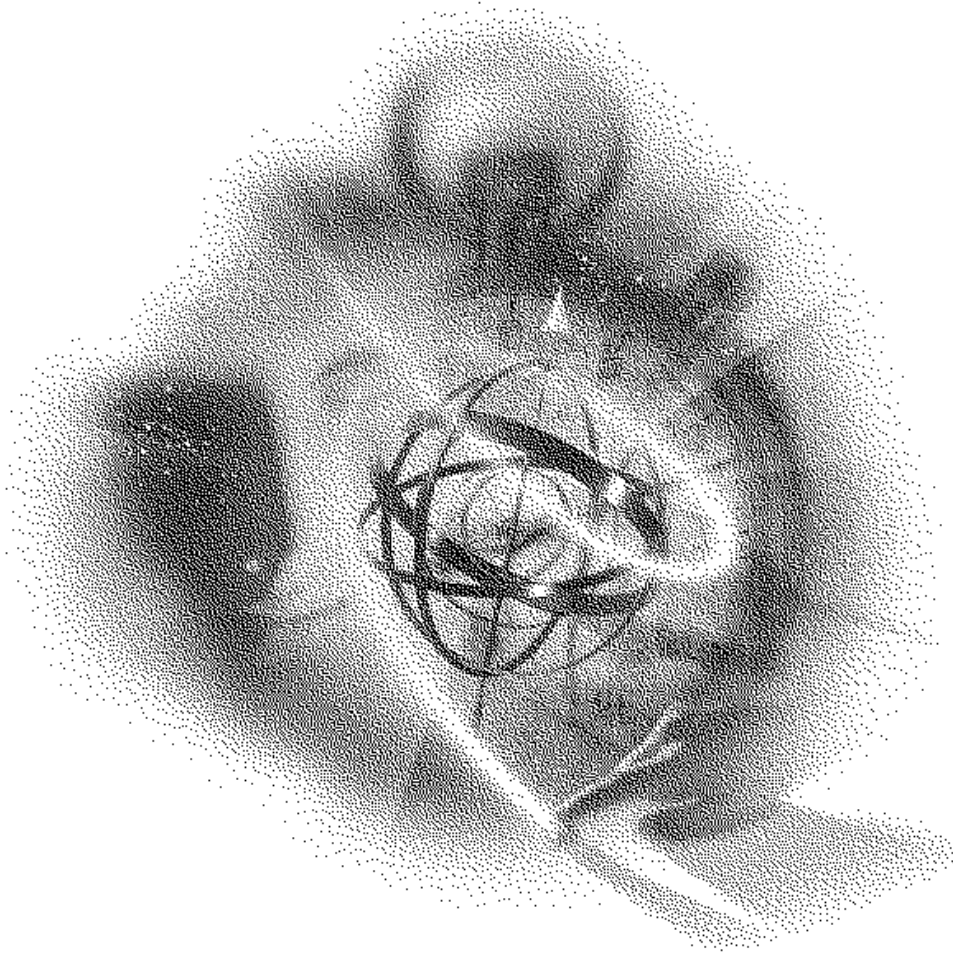


SPEC VERSION 1.11

Protocol Stacks and MLIDs
(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: Protocol Stacks and MLIDs (C Language)
January 6, 1998
100-004006-001**

Contents

AEvent Control Blocks (ECBs)	viii
BPortability Issues	viii
CPlatform Specific Information	viii
DODI HEADER FILE	ix

Preface

Document Organization	xviii
Referenced Documents	xix
Execution Times	xx
Process Time	xx
Privileged Time	xx
Portability Requirements	xxi
Typedef Definitions	xxii
Standard Definitions	xxii
Definitions for Standard Types	xxiv
PROT_ID Structure	xxiv
NODE_ADDR Structure	xxiv
ODISTAT Enumeration	xxiv
SFTIII_STAT Enumeration	xxv
CHNPOS Enumeration	xxvi
Definition for Statistics Table Entries	xxvii
STAT_TABLE_ENTRY Structure	xxvii
Definition for API Function Array Passing	xxviii
INFO_BLOCK Structure	xxviii
Definitions for LSL	xxviii
LOG_BRD_STAT_TABLE_ENTRY Structure	xxviii
LSL_CONFIG_TABLE Structure	xxviii
LSL_STATS_TABLE Structure	xxx
Definitions for Lookahead and Event Control Blocks	xxxi
FRAGMENT_STRUCT Structure	xxxi
ECB Structure	xxxi
AES_ECB Structure	xxxii
LOOKAHEAD Structure	xxxii

Definitions for Protocol Stack	xxxiii
PS_CONFIG_TABLE Structure	xxxiii
PS_STATS_TABLE Structure	xxxiii
Definitions for MLID and Misc. Structures	xxxiv
MLID_CONFIG_TABLE Structure	xxxiv
MLID_STATS_TABLE Structure	xxxv
MLID_REG Structure	xxxvi
PS_BOUND_NODE Structure	xxxvi
PS_CHAINED_RX_NODE Structure	xxxvi
PS_CHAINED_TX_NODE Structure	xxxvii
SFTIII_EXCHANGE_NODE Structure	xxxvii

1 Introduction to ODI

Chapter Overview	1-1
Open Data-Link Interface (ODI)	1-1
Protocol Stacks	1-2
Link Support Layer (LSL).	1-4
Multiple Link Interface Drivers (MLIDs).	1-5
Data Flow	1-6
Send Data Flow	1-6
Receive Data Flow	1-8

1 Overview of Protocol Stacks

Chapter Overview	2-1
Protocol Stack	2-1
Protocol Stack Multiplexing.	2-1
Packet Flow with Multiple Protocol Stacks.	2-5
Routing a Packet to the Correct Protocol Stack	2-5
Routing a Packet to the Correct Logical Board.	2-8
Packet Reception with Multiple Protocol Stacks	2-9

2 Protocol Stack Data Structures

Chapter Overview	3-1
Protocol Stack Configuration Table	3-1
Protocol Stack Configuration Table Structure Sample Code.	3-1
Protocol Stack Configuration Table Field Descriptions.	3-2
Protocol Stack Statistics Table.	3-4
Protocol Stack Statistics Table Structure Sample Code	3-4
Protocol Stack Statistics Table Field Descriptions	3-5
STAT_TABLE_ENTRY Structure Sample Code	3-6
STAT_TABLE_ENTRY Field Descriptions	3-7

ii ODI Specification: Protocol Stacks and MLIDs (C Language)

3 Protocol Stack Initialization

Chapter Overview	4-1
Protocol Stack Initialization Steps	4-1
Locating the LSL	4-2
Registering Protocol Stacks with the LSL	4-2
Determining Which Logical Board(s) to Service	4-3
Explicit Method	4-3
Dynamic Method	4-3
Adding Protocol IDs	4-4
Multiple Board Support	4-5
Obtaining Protocol ID Values	4-5
Customizing the Protocol Stack	4-5
Line Speed	4-6
Measuring Effective Network Performance	4-6
Maximum Packet Size	4-6
Multicast Support	4-7
Receive Lookahead	4-8
Binding to Logical Boards	4-8
Chaining Prescan and Default Protocol Stacks	4-9
Final Initialization	4-12

4 Protocol Stack Packet Reception

Chapter Overview	5-1
Protocol Stack Packet Receive Operation	5-1
Receive Routine Events	5-2
Protocol Stack Packet Reception Methods	5-4
Bound Protocol Stack	5-4
Prescan Protocol Stack	5-4
Default Protocol Stack	5-4
Choosing a Packet Reception Method	5-4
Receive Lookahead	5-5
Receive Handler	5-6
LOOKAHEAD Structure	5-6
Protocol Receive Handler for Bound Stacks	5-16
Protocol Receive Complete Handler for Bound Stacks	5-20
Protocol Receive Handler for Prescan and Default Stacks	5-22
Protocol Receive Complete Handler for Prescan and Default Stacks	5-29

5 Protocol Stack Packet Transmission

Chapter Overview	6-1
Transmit Routine Events	6-1
Prescan Transmit Protocol Stack Method	6-2

Packet Transmission	6-2
Supporting Multiple Outstanding Transmit Requests	6-3
Transmitting the Packet	6-3
Priority Sends	6-3
Event Control Blocks	6-4
ECB_ESR Field	6-5
ECB_StackID Field	6-5
ECB_BoardNumber Field	6-6
ECB_ProtocolID Field	6-6
ECB_ImmediateAddress Field	6-7
ECB_DataLength Field	6-7
ECB_FragmentCount Field	6-7
Fragment Descriptors	6-8
Transmit Handler	6-9
Protocol Transmit Handler for Prescan Stacks	6-10
Protocol Transmit Complete Handler	6-13

6 Protocol Stack Control Routines

Chapter Overview	7-1
Bind	7-3
GetBoundNetworkInfo	7-6
GetProtocolStackConfiguration	7-8
GetProtocolStackStatistics	7-9
GetProtocolStringForBoard	7-10
MLIDDeRegistered	7-12
PromiscuousState	7-14
ProtocolManagement	7-16
Unbind	7-19

8 Overview of the LSL

Chapter Overview	8-1
Link Support Layer (LSL)	8-1
Completion Codes	8-2
Specification Version String	8-2

9 LSL Data Structures

Chapter Overview	9-1
LSL Configuration Table	9-1
LSL Configuration Table Structure Sample Code	9-1
LSL Statistics Table	9-7
LSL Statistics Table Structure Sample Code	9-7

iv ODI Specification: Protocol Stacks and MLIDs (C Language)

10 LSL Support Routines

Chapter Overview	10-1
LSL API Services	10-1
Locating the LSL	10-6
CLSL_AddProtocolID	10-8
CLSL_BindProtocolToBoard	10-10
CLSL_BindStack	10-12
CLSL_CancelAESEvent	10-14
CLSL_CancelEvent	10-15
CLSL_ControlStackFilter	10-16
CLSL_DeRegisterDefaultChain	10-18
CLSL_DeRegisterMLID	10-20
CLSL_DeRegisterPreScanChain	10-21
CLSL_DeRegisterStack	10-23
CLSL_FastHoldEvent	10-25
CLSL_FastSendComplete	10-27
CLSL_GetBoundBoardInfo	10-29
CLSL_GetIntervalMarker	10-31
CLSL_GetLSLConfiguration	10-32
CLSL_GetLSLStatistics	10-33
CLSL_GetMaxECBBufferSize	10-34
CLSL_GetMLIDControlEntry	10-35
CLSL_GetMultipleECBs	10-37
CLSL_GetPhysicalAddressOfECB	10-39
CLSL_GetPIDFromStackIDBoard	10-40
CLSL_GetProtocolControlEntry	10-42
CLSL_GetSizedECB	10-44
CLSL_GetStackECB	10-46
CLSL_GetStackIDFromName	10-49
CLSL_GetStartofChain	10-51
CLSL_HoldEvent	10-53
CLSL_ModifyStackFilter	10-55
CLSL_RegisterDefaultChain	10-58
CLSL_RegisterMLID	10-61
CLSL_RegisterPreScanChain	10-64
CLSL_RegisterStack	10-69
CLSL_ReSubmitDefault	10-72
CLSL_ReSubmitPreScanRx	10-75
CLSL_ReSubmitPreScanTx	10-77
CLSL_ReturnECB	10-79
CLSL_ScheduleAESEvent	10-80
CLSL_SendComplete	10-83
CLSL_SendPacket	10-85

CLSL_SendProtocolInfoToPartner	10-87
CLSL_SendProtocolInfoToOtherEngine	10-89
CLSL_ServiceEvents	10-91
CLSL_UnbindStack	10-92

11 Overview of the MLID

Chapter Overview	11-1
Multiple Operating System Support	11-1
NetWare MLID	11-2
MLID Procedures	11-2
MLID Initialization	11-3
Board Service Routine	11-4
Packet Transmission	11-4
Control Routines	11-4
Timeout Detection	11-5
Driver Remove	11-5
MLID Data Structures and Variables	11-5
MLID Configuration Table	11-5
MLID Statistics Table	11-5
MLID Functionality	11-6
Reentrancy	11-6
Multiple Frame Support	11-7
Other Functionality	11-12
MLID Design Considerations	11-12
Hardware Issues	11-12

12 MLID Data Structures

Chapter Overview	12-1
Frame Data Space	12-1
MLID Configuration Table	12-2
MLID Configuration Table Structure Sample Code	12-2
MLIDCFG_ModeFlags Field	12-15
MLIDCFG_Flags Field	12-18
MLIDCFG_SharingFlags Field	12-20
Adapter Data Space	12-24
MLID Statistics Table	12-24
MLID Statistics Table Structure	12-25
Field Descriptions	12-25
MLID Statistics Table Media Specific Counters	12-32

13 MLID Initialization

Chapter Overview	13-1
----------------------------	------

The MLID Initialization Routine	13-1
Initialization Parameters Passed on the Stack	13-2
Locating the LSL	13-3
Frame and Adapter Data Spaces	13-4
Determining Hardware Options	13-4
Registering Hardware Options	13-7
Initializing the Adapter	13-7
Registering with the LSL	13-8
Setting up a Board Service Routine	13-8
Scheduling Timeout Callbacks	13-9
 14 MLID Packet Reception	
Chapter Overview	14-1
Reception Methods	14-1
Reception Method - Option 1	14-2
Reception Method - Option 2	14-4
Reception Method - Option 3	14-5
Using Shared Interrupts	14-8
 15 MLID Packet Transmission	
Chapter Overview	15-1
MLID Packet Transmission Routine	15-1
Priority Transmission Support	15-4
 16 MLID Timeout Routine	
Chapter Overview	16-1
Establishing a Timeout Routine	16-1
Scheduling a Timeout Check	16-1
Determining the Wait Interval	16-2
Identifying a Timeout Error	16-2
Reinitializing the LAN Adapter	16-2
 17 MLID Remove Routine	
Chapter Overview	17-1
Removing the MLID	17-1
DeRegistering Logical Boards	17-1
Canceling Timeout Check and Polling Routines	17-2
Shutting Down the LAN Adapter	17-2
Remove Data Spaces	17-2

18 MLID Control Routines

Chapter Overview	18-1
MLID Control Routine Overview	18-1
AddMulticastAddress.	18-7
DeleteMulticastAddress	18-11
GetMLIDConfiguration	18-13
GetMLIDStatistics	18-15
GetMulticastInfo	18-17
Index 15 (0x0F) 18-17	
MLIDManagement	18-20
MLIDReset	18-22
MLIDShutdown.	18-24
PromiscuousChange.	18-26
RegisterMonitor	18-30
Index 11 (0x0B) 18-30	
RemoveNetworkInterface	18-34
Index 16 (0X10) 18-34	
ResetNetworkInterface.	18-36
Index 18 (0X12) 18-36	
SetLookAheadSize.	18-38
ShutdownNetworkInterface	18-40
Index 17 (0X11) 18-40	

Appendix A Event Control Blocks (ECBs)

Appendix Overview	A-1
Event Control Blocks.	A-1
Event Control Block Structure Sample Code.	A-1

Appendix B Portability Issues

Portability Issues Overview	B-1
Portability Rules	B-1
Translation Limits.	B-4
Assumptions	B-5
Data Packing and Alignment	B-5

Appendix C Platform Specific Information

Overview	C-1
Intel Processors	C-1
Building the CHSM	C-1

Appendix D ODI HEADER FILE

Appendix Overview	D-1
ODI.H	D-1

Glossary

Revision History

x ODI Specification: Protocol Stacks and MLIDs (C Language)

Figures

Figure 1-1	The ODI Specification	1-2
Figure 1-2	How the 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)	1-6
Figure 1-5	Data Flow from Application to LSL	1-7
Figure 1-6	Data Flow from the LSL to the Board	1-7
Figure 1-7	Data Flow from the Board to the Wire	1-8
Figure 1-8	Receive Data Flow from Wire to Application	1-9
Figure 1-1	One Protocol Stack Using Multiple Frame Types	2-2
Figure 1-2	Multiple Protocol Stacks Using One Frame Type	2-3
Figure 1-3	Multiple Protocol Stacks Using Multiple FrameTypes	2-4
Figure 1-4	Typical Configuration in Protocol Stack Multiplexing	2-6
Figure 1-5	MLID/Protocol Stack Multiplexing	2-8
Figure 3-1	Receive Prescan and Default Protocol Stack Chaining Overview	4-10
Figure 3-2		

	Transmit Prescan Protocol Stack	
	Chaining Overview	4-11
Figure 10-1	LSL Interfaces	10-1
Figure 11-1	Implementations of	
	Multiple Frame Support Using Ethernet	11-10
Figure 11-2	Implementation of	
	Multiple Boards/Frame Support	11-11
Figure 12-1	MLIDCFG_ModeFlags Field Default Values	12-15
Figure 12-2	MLIDCFG_Flags Field Default Values	12-18
Figure 12-3	MLIDCFG_SharingFlags Field Default Values	12-20

Tables

Table 2-1	Protocol Stack Statistics Table Field Descriptions	3-2
Table 2-2	Protocol Stack Statistics Table Field Descriptions	3-5
Table 2-3	Generic Counters Array STAT_TABLE_ENTRY	3-7
Table 3-1	Receive and Control Handlers and the Stack ID	4-2
Table 4-1	Protocol Stack Receive Routine.	5-2
Table 4-2	Receive Handler Event Sequence	5-19
Table 4-3	Receive Complete Handler Event Sequence	5-21
Table 4-4	Receive Complete Handler Event Sequence	5-28
Table 4-5	Receive Complete Handler Event Sequence	5-31
Table 5-1	Protocol Stack Transmit Routine	6-1
Table 5-2	Transmit Handler Event Sequence.	6-9
Table 5-3	Transmit Complete Handler Event Sequence for Bound Stacks	6-14
Table 5-4	Transmit Complete Handler Event Sequence for Prescan Stacks	6-14
Table 9-1	LSL Configuration Table Field Descriptions.	9-3
Table 9-2	LSL Statistics Table Field Descriptions	9-8
Table 9-3	LOG_BRD_STAT_TABLE_ENTRY Field Descriptions	9-8
Table 9-4	Generic STAT_TABLE_ENTRY Counters Array Fields	9-10
Table 10-1		

Table 10-2	Finding LSL API Entry Points for an MLID	10-6
Table 12-1	CLSL_ModifyStackFilter	10-57
Table 12-2	MLID Configuration Table Field Descriptions	12-4
Table 12-3	MLIDCFG_ModeFlags Bits Description	12-15
Table 12-4	MLIDCFG_Flags Bit Description	12-18
Table 12-5	MLIDCFG_SharingFlags Bits Description	12-20
Table 12-6	Frame Types Versus Size Fields	12-22
Table 12-7	MLID Statistics Table Fields	12-27
Table 12-8	MLID Statistics Table Generic Counters	12-30
Table 12-9	Media Specific Counters for Token-Ring	12-32
Table 12-10	Media Specific Counters for Ethernet	12-34
Table A-1	Media Specific Counters for FDDI	12-36
Table C-1	Fragment Structure and ECB Field Descriptions.	A-2
Table C-2	Linker Definition File Example Definitions	C-3
	Load Keywords and Parameters Descriptions	C-6

Preface

This document describes the Open Data-Link Interface™ (ODI™) specification and how to write protocol stacks and network communications drivers (LAN drivers) for NetWare. ODI allows multiple protocols to operate in the NetWare 3.1x (and higher), DOS, OS/2, Windows NT, NEST, and other embedded environments. Writing a LAN driver that conforms to the ODI specification ensures compatibility with any protocol that is also written to the ODI specification (for example, TCP/IP, ISO, IPX™, etc.).

Important



Protocol Stack Developers: It is possible for routines that are part of a protocol stack to execute asynchronously as a result of various interrupt events. This fact, along with today's optimizing compilers, can cause problems with variables changing value when the compiler did not expect them to. The following are two possible solutions for this problem.

At a minimum, you **must** declare a variable *volatile* if it can be referenced by a piece of code that can be executed asynchronously. These pieces of code include event service routines (due to transmit completions), reception completions, AES event completions, and MLID control functions where an asynchronous event has completed.

A better solution than declaring variables *volatile* is to make protocol stacks fully reentrant. This solution not only solves the volatility problem, but also allows code to work on multiprocessor platforms. We strongly recommend this solution, especially since fully reentrant protocol stacks will be required in future versions of this specification.

Document Organization

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

- Section I Introduction

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

- Section II Protocol Stacks

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

- Section III LSL

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

- Section IV MLIDs

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

- Section V Appendixes

Referenced Documents

This document refers to the following Novell documents.

- *Novell ODI Specification: NetWare HSMs (C Language)*, part number 107-000053-001
- *ODI Specification Supplement: The MLID Installation Information File*, part number 107-000056-001
- *ODI Specification Supplement: The Hub Management Interface*, part number 107-000023-001
- *ODI Specification Supplement: Source Routing*, part number 107-000058-001
- *ODI Specification Supplement: Canonical and Noncanonical Addressing*, part number 107-000059-001
- *ODI Specification Supplement: Frame Types and Protocol IDs*, part number 107-000055-001
- *ODI Specification Supplement: Standard MLID Message Definitions*, part number 107-000060-001

Execution Times

The two principal execution times are process time and privileged time. You must be aware of whether a routine is called at process time or at privileged time. The times at which a routine is called effect the support routines it can access.

Process Time

At process time you can allocate memory and (with certain exceptions) perform file input and output (I/O).

Privileged Time

When a routine is called by a privileged process, this routine becomes privileged. At privileged time, the routine should not allocate memory or attempt file I/O, should not suspend its execution, and should not make any calls to routines that may suspend execution. Privileged time routines must be highly optimized and should limit their execution time.

xx ODI Specification: Protocol Stacks and MLIDs (C Language)

Portability Requirements

In writing your driver, if you want it to be portable across different operating systems and/or processors, you need to adhere to the following rules.

- Write your driver in ANSI C—this is extremely important.
- In general, do not declare any variable to be any of the C language basic types (*short*, *long*, *int*, *char*, etc.). Declare variables to be of an abstract type. Then, 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.

Appendix B: Portability Issues describes the above portability rules and additional rules and other important information in detail.

Typedef Definitions

The following is a list of typedef definitions for parameters that are used in this ANSI C implementation of the Open Data Link Interface (ODI) specification.



The structures designated below should be considered as packed structures when compiling.

Standard Definitions

The following are data type declarations and definitions that are used for portability.

MEON	Declared as an 8-bit unsigned character value which contains a 7-bit character or a portion of a double-byte character.
MEON_STRING	Declared as a NULL terminated string of MEON.
UINT8	Declared as an 8-bit unsigned integer.
UINT16	Declared as a 16-bit unsigned integer.
UINT32	Declared as a 32-bit unsigned integer.
UINT64	Declared as a 64-bit unsigned integer.
BOOLEAN	Declared as an unsigned char: FALSE = 0x0 TRUE = 0x1

All pointers are void pointers, or are pointers to a typedef. However, no size definitions may be assumed for them. You cannot assume that pointers are 32-bit values. For example, assuming that a void pointer (PVOID) and UINT32 are the same size is invalid.

All strings are MEON, NULL-terminated strings (ASCIIZ), which can contain double-byte characters. Double-byte characters imbedded in MEON strings may be handled by hardware directly on the platform on which this specification is implemented, but this is not the concern of this specification. In a platform that uses Unicode strings, Windows NT for example, it is the responsibility of the application, driver, etc. to present the Unicode string equivalents of the MEON strings to the platform.

All bit field descriptions are described as numeric values and the use or interrogation of bit field values is by numeric methods. This eliminates, as far as possible, the little endian and big endian conflicts.

For example:

```
#define bit_flag_x 0x0040 /* for b6 in a 16-bit
bit field. */
```

Note



or implementation on the Windows NT platforms, the following definitions are made:

```
typedef unsigned char MEON; /* equivalent UCHAR */
typedef unsigned char UINT8; /* equivalent UCHAR */
typedef unsigned short UINT16; /* equivalent USHORT */
typedef unsigned int UINT32; /* equivalent UINT */
typedef unsigned long UINT32; /* equivalent ULONG */
typedef unsigned char MEON_STRING; /* equivalent UCHAR */
typedef struct _UINT64 /* equivalent for UINT64 on 32-bit platform */
{
    UINT32 Low_UINT32;
    UINT32 High_UINT32;
} UINT64;
```

By convention, all MEON_STRINGs are NULL terminated strings—for example:

```
MEON_STRING amsg [ ] = "String";
```

Definitions for Standard Types

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:

```
#define  PID_SIZE  6
```

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:

```
#define  ADDR_SIZE  6
```

ODISTAT Enumeration

ODISTAT enumerates the values returned in the ODI platform by function calls; these values are used to indicate success or an error.

```
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_IN_CRITICAL_SECTION= -117,
ODISTAT_TRANSMIT_FAILED= -116,
ODISTAT_PACKET_UNDELIVERABLE= -115,
ODISTAT_CANCELED= -4
} ODISTAT;

```

Note



ODISTAT_NO_SUCH_DRIVER has been equated to
ODISTAT_MLID_SHUTDOWN:

```
#define ODISTAT_NO_SUCH_DRIVER ODISTAT_MLID_SHUTDOWN
```

SFTIII_STAT Enumeration

SFTIII_STAT enumerates the SFTIII status values returned in the ODI platform by function calls; these values are used to indicate success or an error.

```

typedef enum _SFTIII_STAT_
{
    SFTIII_STAT_SUCCESSFUL= 0,
    SFTIII_STAT_MIRROR_NOT_ACTIVE= 1,
    SFTIII_STAT_NO_PARTNER= 2,
    SFTIII_STAT_OUT_OF_RESOURCES= 3,
    SFTIII_STAT_NOT_SUPPORTED= -1
} SFTIII_STAT;

```

Note



SFTIII_STAT_NOT_SUPPORTED is assumed to be all bits set. s

CHNPOS Enumeration

CHNPOS enumerates the protocol stack chain position for chained protocol stacks.

```
typedef enum _CHNPOS_  
{  
    CHNPOS_FIRST_MUST,  
    CHNPOS_FIRST_NEXT,  
    CHNPOS_LOAD_ORDER,  
    CHNPOS_LAST_NEXT,  
    CHNPOS_LAST_MUST  
} CHNPOS;
```

Definition for Statistics Table Entries

STAT_TABLE_ENTRY Structure

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

where the following are the permissible *StatUseFlag* values:

ODI_STAT_UNUSED	StatCounter entry not in use.
ODI_STAT_UINT32	StatCounter is a pointer to an UINT32 counter.
ODI_STAT_UINT64	StatCounter is a pointer to an UINT64 counter.
ODI_STAT_MEON_STRING	StatCounter is a pointer to a Null terminated string of MEON.
ODI_STAT_UNTYPED	StatCounter is a pointer to a UINT32 length preceded array of UINT8.
ODI_STAT_RESETTABLE	StatCounter can be reset by an external entity when needed.

StatString is a pointer to a NULL terminated MEON string that describes the statistics counter.

StatCounter is as defined by *StatUseFlag*.

Definition for API Function Array Passing

INFO_BLOCK Structure

```
typedef struct _INFO_BLOCK_
{
    UINT32  NumberOfAPIs;
    void ( **SupportAPIArray ) ();
} INFO_BLOCK;
```

Definitions for LSL

LOG_BRD_STAT_TABLE_ENTRY Structure

```
typedef struct _LOG_BRD_STAT_TABLE_ENTRY_
{
    UINT32  LogBrd_TransmittedPackets;
    UINT32  LogBrd_ReceivedPackets;
    UINT32  LogBrd_UnclaimedPackets;
    UINT32  LogBrd_TxOverloaded;
} LOG_BRD_STAT_TABLE_ENTRY;
```

LSL_CONFIG_TABLE Structure

```
typedef struct _LSL_CONFIG_TABLE_
{
    UINT16      LConfigTableMajorVer;
    UINT16      LConfigTableMinorVer;
    MEON_STRING *LSLLongName;
    MEON_STRING *LSLShortName;
    UINT16      LSLMajorVer;
    UINT16      LSLMinorVer;
    UINT32      LMaxNumberOfBoards;
    UINT32      LMaxNumberOfStacks;
    UINT32      LConfigTableReserved0;
    UINT32      LConfigTableReserved1;
    UINT32      LConfigTableReserved2;
    UINT8       LSLCFG_ODISpecMajorVer;
```

xxviii ODI Specification: Protocol Stacks and MLIDs (C Language)

```

UINT8      LSLCFG_ODISpecMinorVer;
UINT16     LConfigTableReserved3;
UINT32     LSLCFG_SystemFlags;
UINT32     LSLCFG_SmallECBCount;
UINT32     LSLCFG_MediumECBCount;
UINT32     LSLCFG_LargeECBCount;
UINT32     LSLCFG_XLargeECBCount;
UINT32     LSLCFG_HugeECBCount;
UINT32     LSLCFG_SmallECBBelow16Count;
UINT32     LSLCFG_MediumBelow16ECBCount;
UINT32     LSLCFG_LargeBelow16ECBCount;
UINT32     LSLCFG_XLargeBelow16ECBCount;
UINT32     LSLCFG_HugeBelow16ECBCount;
UINT32     LSLCFG_SmallECBMinCount;
UINT32     LSLCFG_MediumECBMinCount;
UINT32     LSLCFG_LargeECBMinCount;
UINT32     LSLCFG_XLargeECBMinCount;
UINT32     LSLCFG_HugeECBMinCount;
UINT32     LSLCFG_SmallECBMaxCount;
UINT32     LSLCFG_MediumECBMaxCount;
UINT32     LSLCFG_LargeECBMaxCount;
UINT32     LSLCFG_XLargeECBMaxCount;
UINT32     LSLCFG_HugeECBMaxCount;
UINT32     LSLCFG_SmallECBSize;
UINT32     LSLCFG_MediumECBSize;
UINT32     LSLCFG_LargeECBSize;
UINT32     LSLCFG_XLargeECBSize;
UINT32     LSLCFG_HugeECBSize;
} LSL_CONFIG_TABLE;

```

LSL_STATS_TABLE Structure

```

typedef struct _LSL_STATS_TABLE_
{
    UINT16          LStatTableMajorVer;
    UINT16          LStatTableMinorVer;
    UINT32          LNumGenericCounters;
    STAT_TABLE_ENTRY (*LGenericCountersPtr)[];
    UINT32          LNumLogicalBoards;
    LOG_BRD_STAT_TABLE_ENTRY
        (*LogicalBoardStatTablePtr)[];
    UINT32          LNumCustomCounters;
    STAT_TABLE_ENTRY (*LCustomCountersPtr)[];
} LSL_STATS_TABLE;

```

xxx ODI Specification: Protocol Stacks and MLIDs (C Language)

Definitions for Lookahead and Event Control Blocks

FRAGMENT_STRUCT Structure

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

ECB Structure

```
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
    {
        {
            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;
```

xxxi

```

        UINT32    ECB_FragmentCount;
    FRAGMENT_STRUCT    ECB_Fragment[1];
}    ECB;

```

AES_ECB Structure

```

typedef    struct    _AES_ECB_
{
    struct    _AES_ECB_    *AES_Link;
    UINT32        AES_MSecondValue;
    UINT16        AES_Status;
    void            (*AES_ESR)(struct    _AES_ECB_    *);
    UINT32        AES_Reserved;
    void            *AES_ResourceObj;
    void            *AES_Context;
}    AES_ECB;

```

LOOKAHEAD Structure

```

typedef    struct    _LOOKAHEAD_
{
    ECB            *LkAhd_PreFilledECB;
    UINT8          *LkAhd_MediaHeaderPtr;
    UINT32         LkAhd_MediaHeaderLen;
    UINT8          *LkAhd_DataLookAheadPtr;
    UINT32         LkAhd_DataLookAheadLen;
    UINT32         LkAhd_BoardNumber;
    UINT32         LkAhd_PktAttr;
    UINT32         LkAhd_DestType;
    UINT32         LkAhd_FrameDataSize;
    UINT16         LkAhd_PadAlignBytes1;
    PROT_ID        LkAhd_ProtocolID;
    UINT16         LkAhd_PadAlignBytes2;
    NODE_ADDR      LkAhd_ImmediateAddress;
    UINT32         LkAhd_FrameDataStartCopyOffset;
    UINT32         LkAhd_FrameDataBytesWanted;
    ECB            *LkAhd_ReturnedECB;
    UINT32         LkAhd_PriorityLevel;
}

```

xxxii ODI Specification: Protocol Stacks and MLIDs (C Language)


```

        void                *LkAhd_Reserved;
    }   LOOKAHEAD;

```

Definitions for Protocol Stack

PS_CONFIG_TABLE Structure

```

typedef struct   _PS_CONFIG_TABLE_
{
    UINT16        PConfigTableMajorVer;
    UINT16        PConfigTableMinorVer;
    MEON_STRING   *PProtocolLongName;
    MEON_STRING   *PProtocolShortName;
    UINT16        PProtocolMajorVer;
    UINT16        PProtocolMinorVer;
}   PS_CONFIG_TABLE;

```

PS_STATS_TABLE Structure

```

typedef struct   _PS_STATS_TABLE_
{
    UINT16        PStatTableMajorVer;
    UINT16        PStatTableMinorVer;
    UINT32        PNumGenericCounters;
    STAT_TABLE_ENTRY  (*PGenericCountersPtr)[];
    UINT32        PNumCustomCounters;
    STAT_TABLE_ENTRY  (*PCustomCountersPtr)[];
}   PS_STATS_TABLE;

```

Definitions for MLID and Misc. Structures

MLID_CONFIG_TABLE Structure

```
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;
}
```

xxxiv ODI Specification: Protocol Stacks and MLIDs (C Language)

```

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;

```

MLID_STATS_TABLE Structure

```

typedef struct _MLID_STATS_TABLE_
{
    UINT16      MStatTableMajorVer;
    UINT16      MStatTableMinorVer;
    UINT32      MNumGenericCounters;
    STAT_TABLE_ENTRY (*MGenericCountersPtr)[ ];
    UINT32      MNumMediaCounts;
    STAT_TABLE_ENTRY (*MMediaCountersPtr)[ ];
    UINT32      MNumCustomCounts;
    STAT_TABLE_ENTRY (*MCustomCountersPtr)[ ];
} MLID_STATS_TABLE;

```

XXXV

MLID_REG Structure

```
typedef struct _MLID_REG_
{
    void (*MLIDSendHandler)(ECB*, void *);
    INFO_BLOCK *MLIDControlHandler;
    void *MLIDSendContext;
    void *MLIDResourceObj;
    void *MLIDModuleHandle;
} MLID_REG;
```

PS_BOUND_NODE Structure

```
typedef struct _PS_BOUND_NODE_
{
    MEON_STRING *ProtocolName;
    ODISTAT (*ProtocolReceiveHandler)
        (LOOKAHEAD*);
    INFO_BLOCK *ProtocolControlHandler;
    void *ProtocolResourceObj;
} PS_BOUND_NODE;
```

PS_CHAINED_RX_NODE Structure

```
typedef struct _PS_CHAINED_RX_NODE_
{
    struct _PS_CHAINED_RX_NODE_ *StackChainLink;
    UINT32 StackChainBoardNumber;
    CHNPOS StackChainPositionRequested;
    ODISTAT (*StackRxChainHandler)(LOOKAHEAD*,
        struct _PS_CHAINED_RX_NODE_ *);
    INFO_BLOCK *StackChainControl;
    UINT32 StackChainFilter;
    void *StackChainContext;
    void *StackChainResourceObj;
} PS_CHAINED_RX_NODE;
```

xxxvi ODI Specification: Protocol Stacks and MLIDs (C Language)

PS_CHAINED_TX_NODE Structure

```

typedef struct _PS_CHAINED_TX_NODE_
{
    struct _PS_CHAINED_TX_NODE_ *StackChainLink;
    UINT32      StackChainBoardNumber;
    CHNPOS      StackChainPositionRequested;
    ODISTAT      (*StackTxChainHandler)(ECB*,
                                         struct _PS_CHAINED_TX_NODE_ *);
    INFO_BLOCK *StackChainControl;
    UINT32      StackChainFilter;
    void        *StackChainContext;
    void        *StackChainResourceObj;
} PS_CHAINED_TX_NODE;

```

SFTIII_EXCHANGE_NODE Structure

```

typedef struct _SFTIII_EXCHANGE_NODE_
{
    UINT32SubFunction,
    void *Parameter1,
    void *Parameter2
} SFTIII_EXCHANGE_NODE;

```

xxxviii ODI Specification: Protocol Stacks and MLIDs (C Language)

Introduction to ODI

Overview

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

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



Note

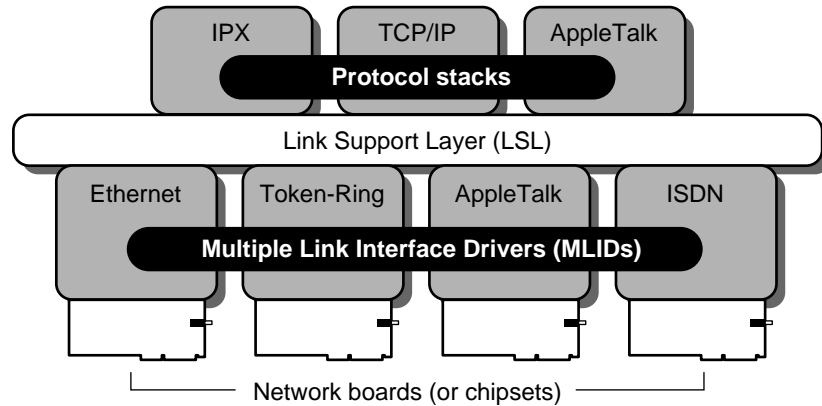
The terms *MLID* and *LAN driver* can be interchanged.

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

Open Data-Link Interface (ODI)

ANSI/ANSIC C language MLIDS and protocol stacks must conform to the ODI specification. Figure 1.1 illustrates the elements that make up the ODI specification.

Figure 1-1
The ODI Specification



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

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

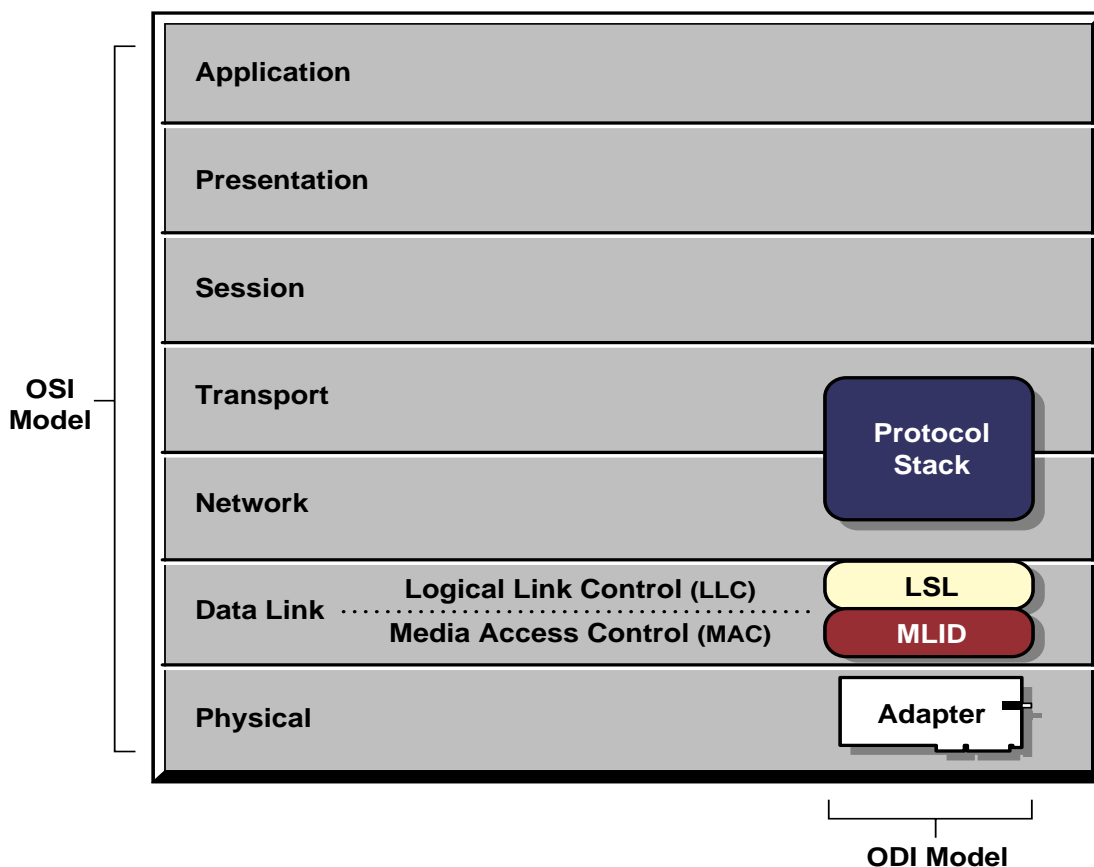
Protocol Stacks

Protocol Stack Functionality

Network Layer protocol stacks transmit and receive data over a logical or physical network. They also handle routing, connection services, and APIs, and provide an interface to allow higher layer protocols or applications access to the protocol stack's services. As a general rule, protocol stacks written to the ODI specification provide OSI (Open Systems Interconnection) Network Layer functionality; however, they are not limited to this. Figure 1.2 illustrates the ODI/OSI correspondence.

1-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

Figure 1-2
How the ODI Fits into the OSI Model



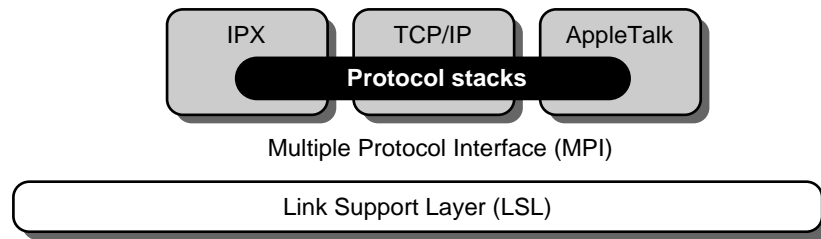
The Multiple Protocol Interface (MPI)

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. However, protocol stacks written to the ODI specification 3 and later also have full access to the NLM APIs documented in the *NetWare Loadable*

Introduction to ODI **1-3**

Module Library Reference—Volume II. Protocol stacks also have full access to the NLM APIs documented in *NetWare Loadable Module Library Reference—Volume II.*

Figure 1-3
The Multiple Protocol Interface (MPI)



Link Support Layer (LSL)

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



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

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

In addition, the LSL performs the following services:

1-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

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

Multiple Link Interface Drivers (MLIDs)

MLID Functionality

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

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

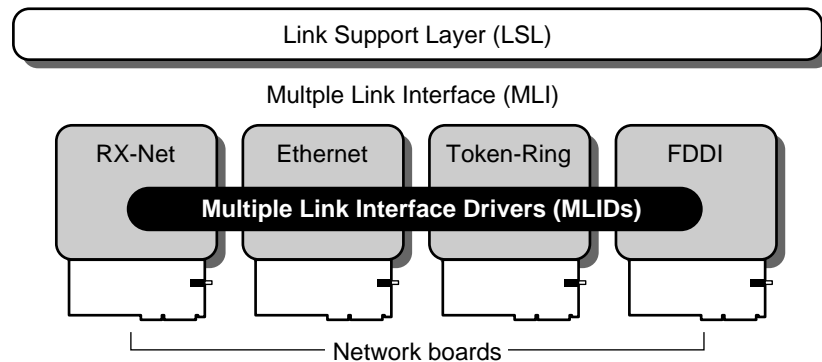
All MLIDs can handle packets from various protocols because the MLID does not interpret the packet. Instead, it passes received packets to the Link Support Layer (LSL) using Event Control Blocks (ECBs). ECBs are data structures that the MLID uses to send or receive packets or to schedule events.

Introduction to ODI 1-5

The Multiple Link Interface (MLI)

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

Figure 1-4
The Multiple Link Interface (MLI)



Data Flow

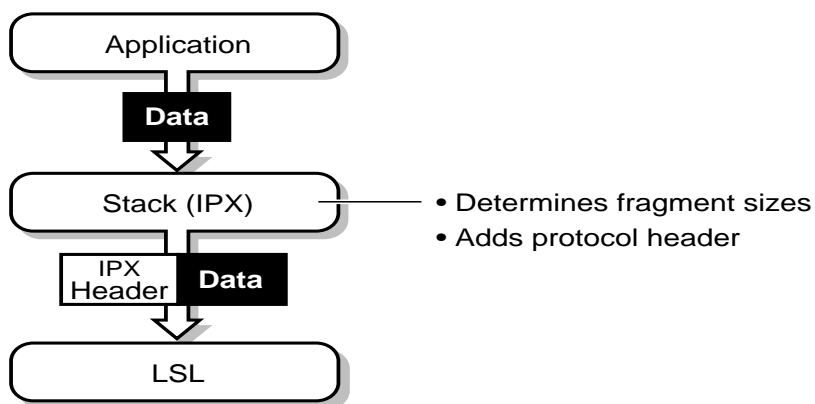
When messages are sent and received, the various protocols or layers add and remove their own information at each layer. The following diagrams illustrate basic data flow.

Send Data Flow

As Figure 1.6 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.

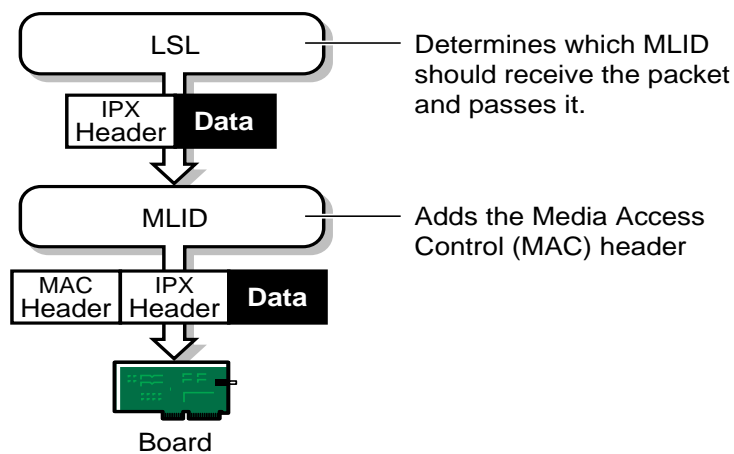
1-6 ODI Specification: Protocol Stacks and MLIDs (C Language)

Figure 1-5
Data Flow from Application to LSL



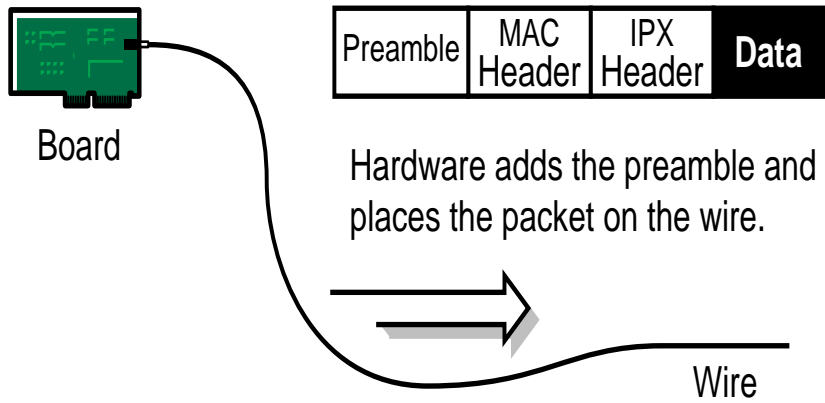
As illustrated by Figure 1.7 , the LSL directs the packet to the appropriate MLID. The MLID then adds the MAC header to the packet and hands the packet to the LAN adapter.

Figure 1-6
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.

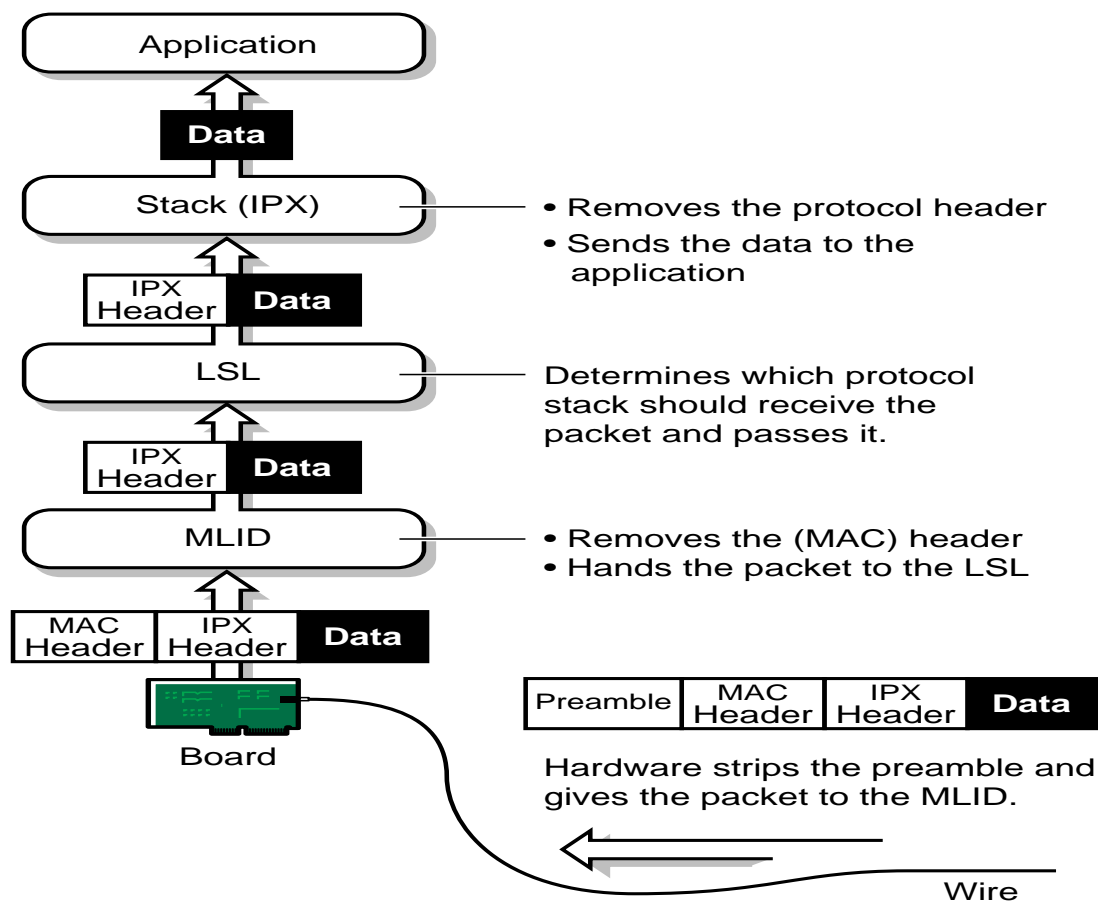
Figure 1-7
Data Flow from the Board to the Wire



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 1-8
Receive Data Flow from Wire to Application



1-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

Chapter Overview

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

You should read this chapter if you have not previously developed an ODI protocol stack.

Protocol Stack

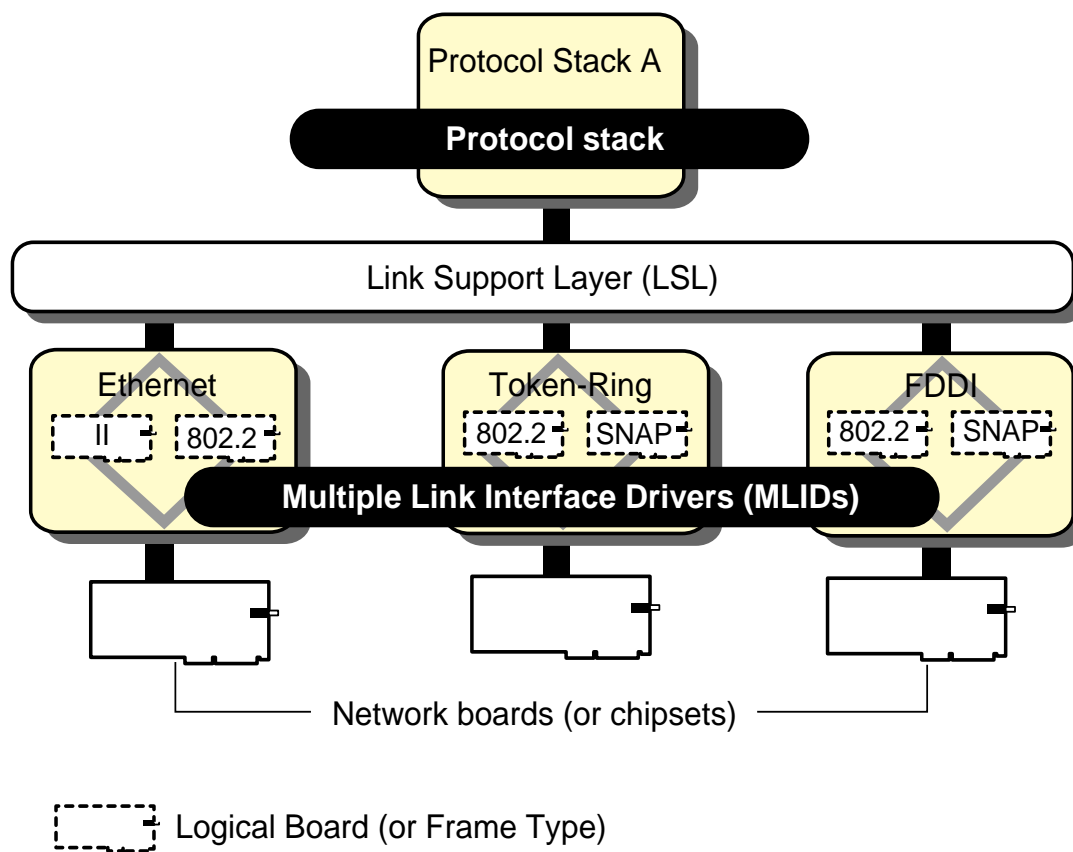
Protocol stacks transmit and receive data over a network. They provide the interface that allows higher layer protocols or applications access to the protocol stack's services such as routing and connection.

Protocol Stack Multiplexing

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

- One protocol stack can concurrently use multiple frame types (also called *logical boards*)
- Multiple protocol stacks can be concurrently used by a frame type
- Or any combination of multiple protocols and multiple frame types is possible. (See Figures 2-1 through 2-3 .)

Figure 1-1
One Protocol Stack Using
Multiple Frame Types



1-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

Figure 1-2
Multiple Protocol Stacks Using
One Frame Type

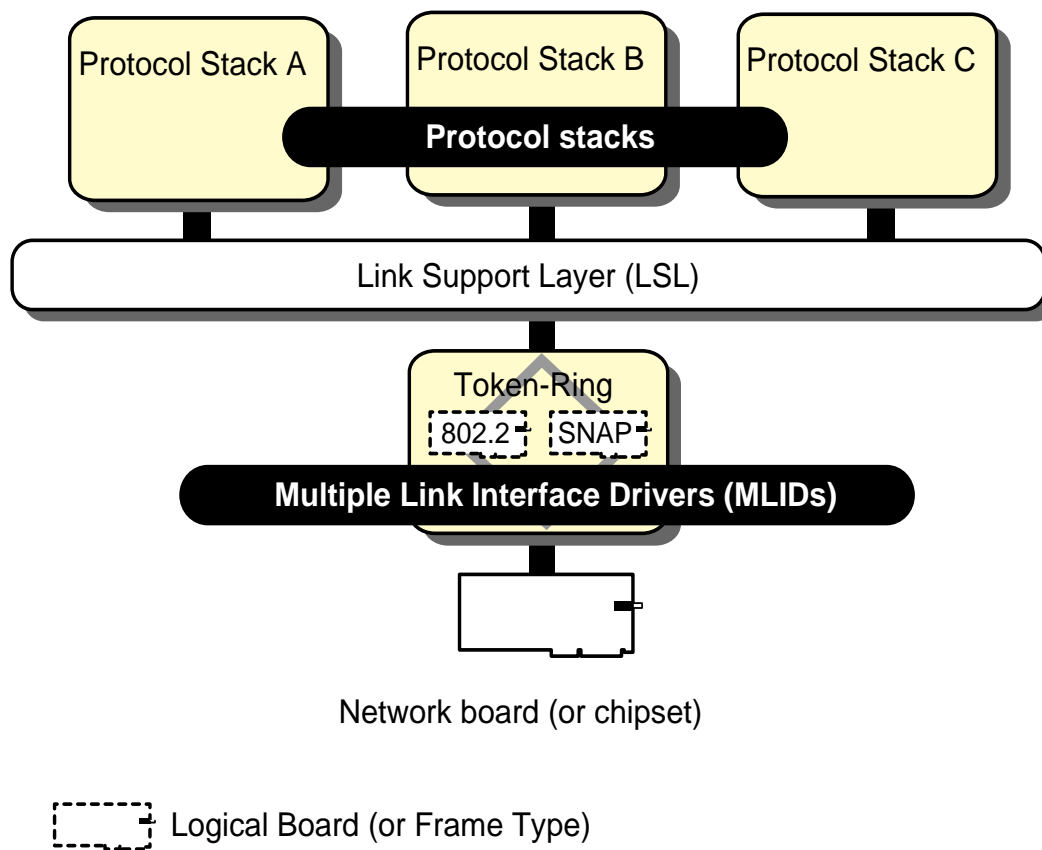
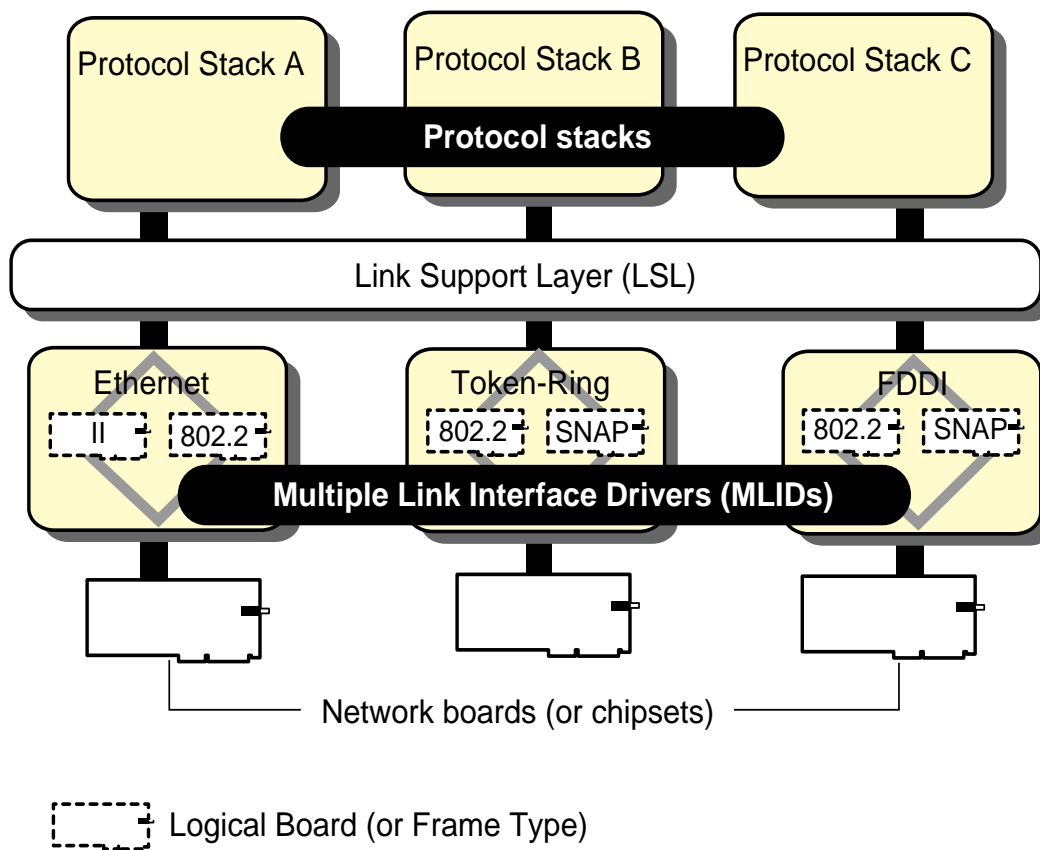


Figure 1-3
Multiple Protocol Stacks Using
Multiple FrameTypes



1-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

Packet Flow with Multiple Protocol Stacks

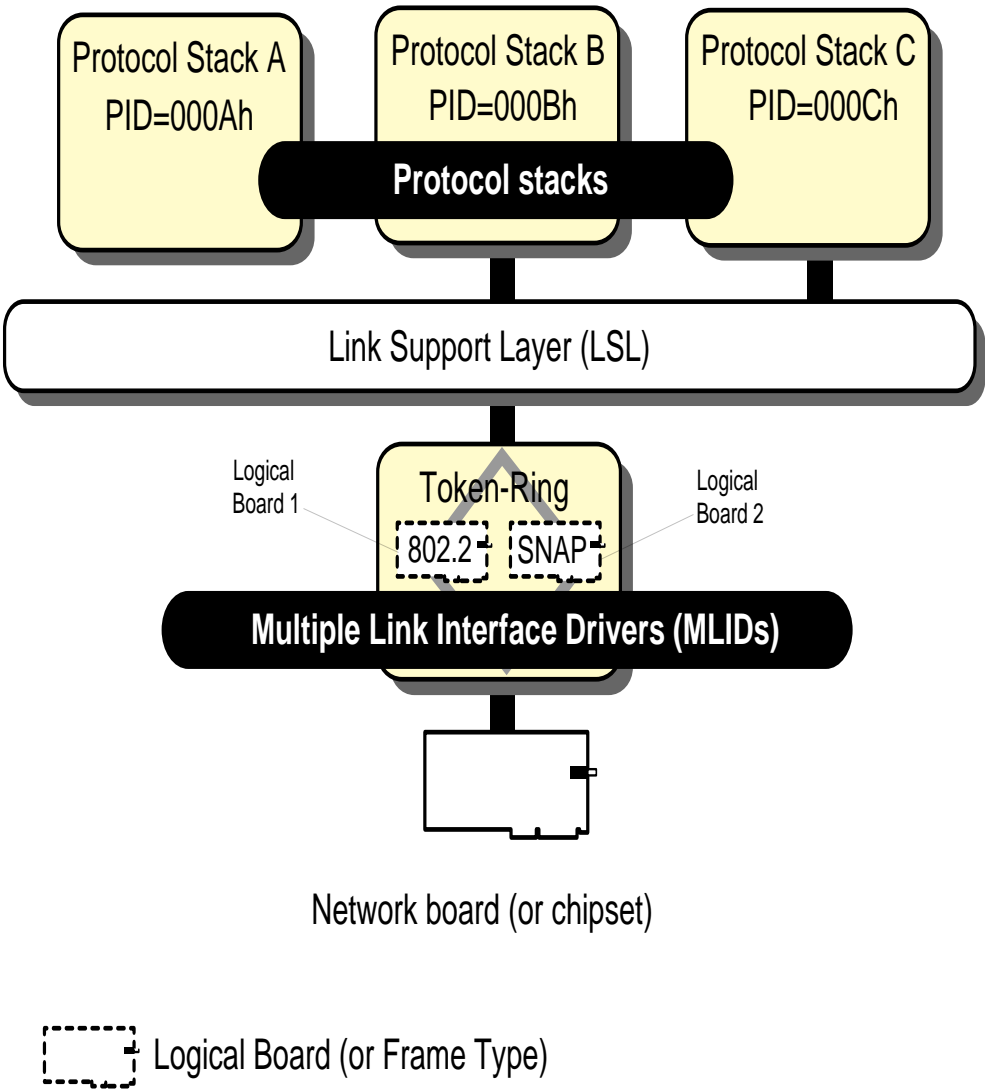
Protocol stacks are media and frame-type unaware. Therefore, in order for multiple protocol stacks to communicate with the logical boards, the LSL must have a unique value identifying each protocol stack and logical board (frame type).

Routing a Packet to the Correct Protocol Stack

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

Figure 2-4 illustrates the configuration for the following discussion.

Figure 1-4
Typical Configuration in
Protocol Stack Multiplexing



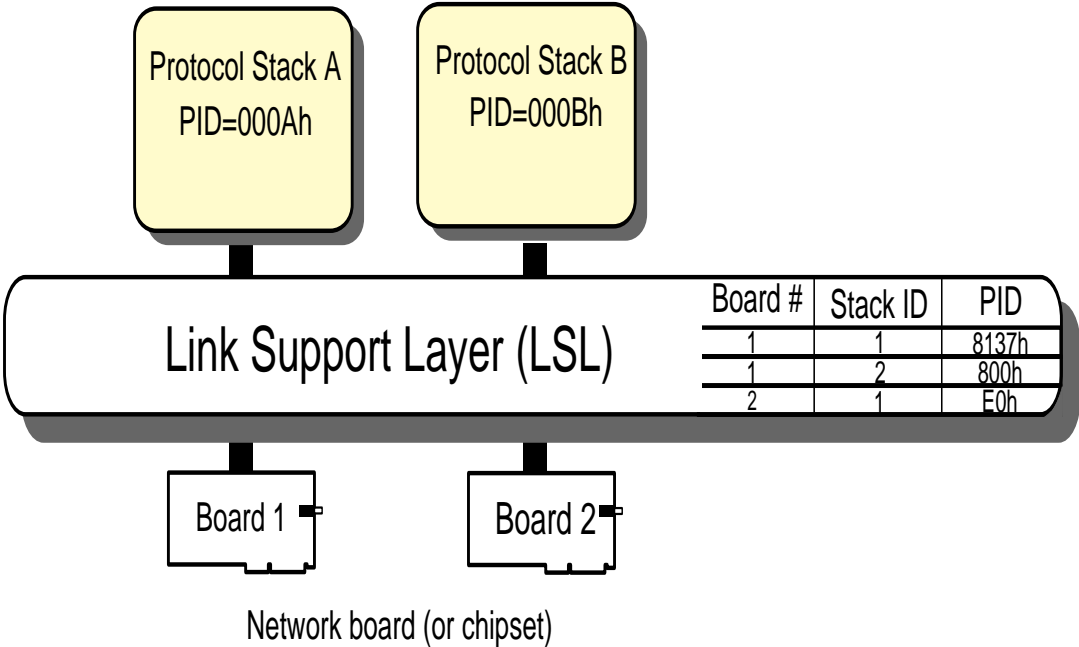
During protocol stack initialization, the stack registers with the LSL. The LSL assigns a unique value known as a Stack ID (SID) to each protocol stack. When the LSL binds the protocol stack to a frame type (logical board), the LSL assigns a predefined Protocol ID (PID) to that protocol stack. The LSL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

When a protocol stack sends a request to be transmitted, the MLID transmitting the request embeds the appropriate PID in the MAC header of the request packet. The location and format of the PID in the frame header is topology and frame dependent and does not concern the protocol stack.

When a logical board in the MLID receives a request from the wire, its ISR (Interrupt Service Routine) fills in the LOOKAHEAD structure's *LkAhd_BoardNumber* field or Event Control Block (ECB) structure's *ECB_BoardNumber* field with that logical board number. (An ECB is a buffer that contains information regarding the packet and fragment descriptors pertaining to the packet data. For information on the ECB, see *Appendix A: Event Control Blocks (ECBs)*.) The logical board in the MLID takes the PID from the MAC header and places it in the *LkAhd_ProtocolID/ECB_ProtocolID* field. The MLID hands the LOOKAHEAD structure, which contains the prefilled ECB if the MLID uses ECBs, to the LSL. The LSL uses the logical board number and the PID to index the table and determine the protocol SID of the stack that is to receive the packet. (See Figure 2-5.)

Overview of Protocol Stacks 1-7

Figure 1-5
MLID/Protocol Stack Multiplexing



Routing a Packet to the Correct Logical Board

When a response is transmitted, the LSL is able to check the *ECB_BoardNumber* field (filled out in the process above) to determine the logical board in the MLID that prepares the packet for transmission.

Packet Reception with Multiple Protocol Stacks

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

- **Board Number** The board number specifies the logical board and the frame type.
- **Protocol ID (PID)** The PID, together with the board number, specifies the protocol stack that the packet is sent to.

When a protocol stack registers with the LSL, the stack gives the LSL the address of the stack's receive handler routine. This routine is usually called at privileged time.

Protocol Stack Packet Reception Methods

The ODI specification defines these protocol stack reception methods:

- Bound
- Prescan
- Default

Prescan and default protocol stacks can be chained (see Chapter 4, "Protocol Stack Initialization").

Bound Protocol Stacks

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

The LSL uses the PID in the *LkAhd_ProtocolID/ECB_ProtocolID* field to locate the appropriate protocol stack to receive the packet. A bound protocol stack receives only the packets that have the same PID as that registered for that stack.

Overview of Protocol Stacks 1-9

A registered protocol stack only receives packets whose PID corresponds to a logical board. Protocol stacks containing a limited number of network layer protocols that use different PIDs (for example, TCP/IP, ARP, RARP) must be registered to the LSL as separate and distinct protocols. These protocol stacks are logically fragmented and each fragment must register with the LSL as a separate protocol stack. However, these fragments can still be located in the same piece of software and can specify the same receive handler routine. The receive handler routine then examines the *LkAhd_ProtocolID/ECB_ProtocolID* field to determine the subprotocol that the frame is intended for.

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

Prescan Protocol Stacks

Prescan protocol stacks receive all incoming packets from a particular LAN adapter before the packet is routed to the appropriate bound protocol stack. If the prescan stack consumes the packet, it must either resubmit the packet or discard the packet. Special purpose protocol stacks such as packet filters, diagnostic utilities, or compression protocol stacks are used as prescan stacks.

If a prescan protocol stack chain exists, other prescan protocol stacks will still be placed into their requested position in the chain. This allows multiple prescan protocol stacks in the system.

Default Protocol Stacks

Default protocol stacks receive every frame not claimed by any other protocol stack (prescan, bound, or other default stacks in the chain). In other words, these stacks receive all leftover packets.

Default protocol stacks typically provide a Logical Link Control Layer solution. If a default protocol stack chain exists, other default protocol stacks will be placed into their requested position in the chain. This allows multiple default protocol stacks in the system.

Packet Reception Process

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

1-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

information required to route packets from MLIDs to protocol stacks. The steps involved in packet reception are given in Chapter 5, "Protocol Stack Packet Reception".

.

Overview of Protocol Stacks **1-11**

1-12 ODI Specification: Protocol Stacks and MLIDs (C Language)

chapter **2** *Protocol Stack Data Structures*

Chapter Overview

This chapter presents the structure code and field descriptions of a protocol stack configuration table and a protocol stack statistics table.

Protocol Stack Configuration Table

Protocol Stack Configuration Table Structure Sample Code

```
typedef struct _PS_CONFIG_TABLE_
{
    UINT16          PConfigTableMajorVer;
    UINT16          PConfigTableMinorVer;
    MEON_STRING     *PProtocolLongName;
    MEON_STRING     *PProtocolShortName;
    UINT16          PProtocolMajorVer;
    UINT16          PProtocolMinorVer;
    UINT8           PConfigTable_ODISpecMajorVersion;
    UINT8           PConfigTable_ODISpecMinorVersion;
    UINT8           PConfigTable_ProtocolAPIMajorVersion;
    UINT8           PConfigTable_ProtocolAPIMinorVersion;
    UINT32          PConfigTable_SystemFlags;
    UINT32          PConfigTable_ProtocolFlags;
    UINT32          PConfigTable_ProtocolReserved;
} PS_CONFIG_TABLE;
```

Protocol Stack Configuration Table Field Descriptions

Table 2-1
Protocol Stack Statistics Table Field Descriptions

Field	Description
PConfigTableMajorVer	The major version number of the protocol stack configuration table. Use PSTK_CONFIG_TABLE_MAJOR_VER, defined in ODI.H.
PConfigTableMinorVer	The minor version number of the protocol stack configuration table. Use PSTK_CONFIG_TABLE_MINOR_VER, defined in ODI.H.
PProtocolLongName	Pointer to a NULL terminated MEON string describing the protocol stack in detail.
PProtocolShortName	Pointer to a NULL terminated MEON string containing the short name for the protocol stack, which is used to register the protocol stack. This string cannot have more than 15 characters (not including the NULL terminator).
PProtocolMajorVer	Decimal value that indicates the major version number of the protocol stack.
PProtocolMinorVer	Decimal value that indicates the minor version number of the protocol stack (0 through 99).
PConfigTable_ODISpecMajorVersion	The major version of the ODI Specification that the protocol stack is written to. For example, if the ODI Specification is version 1.11, the value of this field is 1. To set this field, protocol stacks should use ODI_SPEC_MAJOR_VER, defined in ODI.H.
PConfigTable_ODISpecMinorVersion	The minor version of the ODI Specification that the protocol stack is written to. For example, if the ODI Specification is version 1.11, the value of this field is 11. To set this field, protocol stacks should use ODI_SPEC_MINOR_VER, defined in ODI.H.
PConfigTable_ProtocolAPIMajorVersion	The major version of the protocol stack API interface. For example, if the API interface is version 1.00, the value of this field is 1. The protocol stack determines the value of this field.
PConfigTable_ProtocolAPIMinorVersion	The minor version of the protocol stack API interface. For example, if the API interface is version 1.00, the value of this field is 00. The protocol stack determines the value of this field.

2-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 2-1
Protocol Stack Statistics Table Field Descriptions

Field	Description
PConfigTable_SystemFlags	<p>The constants for this field are defined in ODI.H as follows:</p> <p>PSTK_CFG_AUTO_NETWORK_RESOLUTION_BIT</p> <p>SET if automatic network resolution is on.</p> <p>PSTK_CFG_AUTO_BIND_ACTIVE_BIT</p> <p>SET if automatic binding is on.</p> <p>PSTK_CFG_ROUTER_ACTIVE_BIT</p> <p>SET if a protocol router is present.</p> <p>PSTK_CFG_SERVER_BIT</p> <p>SET if the protocol stack is running in a server environment (mutually exclusive with PSTK_CFG_CLIENT_BIT).</p> <p>PSTK_CFG_CLIENT_BIT</p> <p>SET if the protocol stack is running in a client environment (mutually exclusive with PSTK_CFG_SERVER_BIT).</p>
PConfigTable_ProtocolFlags	Defined by individual protocol stacks.
PConfigTable_ProtocolReserved	Reserved for protocol specific use.

Protocol Stack Statistics Table

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

Protocol Stack Statistics Table Structure Sample Code

```
typedef struct _PS_STATS_TABLE_  
{  
    UINT16          PStatTableMajorVer;  
    UINT16          PStatTableMinorVer;  
    UINT32          PNumGenericCounters;  
    STAT_TABLE_ENTRY (*PGenericCountersPtr)[];  
    UINT32          PNumCustomCounters;  
    STAT_TABLE_ENTRY (*PCustomCountersPtr)[];  
} PS_STATS_TABLE;
```


Protocol Stack Statistics Table Field Descriptions

Table 2-2

Protocol Stack Statistics Table Field Descriptions

Field	Description
PStatTableMajorVer	The major version number of the protocol stack statistics table (2 for this specification).
PStatTableMinorVer	The minor version number of the protocol stack statistics table (0 for this specification).
PNumGenericCounters	The total number of generic <i>STAT_TABLE_ENTRY</i> counters in this portion of the table. Set this field to 0x0003 for this specification.
PGenericCountersPtr	Pointer to an array of <i>STAT_TABLE_ENTRY</i> counters [PNumGenericCounters].
PNumCustomCounters	The total number of custom <i>STAT_TABLE_ENTRY</i> counters in this portion of the table. The value in this field is protocol stack dependent.
PCustomCountersPtr	Pointer to an array of <i>STAT_TABLE_ENTRY</i> counters [PCustomCounters].

STAT_TABLE_ENTRY Structure Sample Code

```
#define          NUM_GENERIC_COUNTERS  3
UINT32          PTotalTxPackets,      PTotalRxPackets,  PIgnoredRxPackets;
MEON_STRING     PTotalTxPacketStr     "Total Tx Packets";
MEON_STRING     PTotalRxPacketStr     "Total Rx Packets";
MEON_STRING     PIgnoredRxPacketStr   "Rx Packets Ignored";
STAT_TABLE_ENTRY PGenericCounters[NUM_GENERIC_COUNTERS] =
{
    { ODI_STAT_UINT32,  &PTotalTxPackets,    &PTotalTxPacketsStr },
    { ODI_STAT_UINT32,  &PTotalRxPackets,    &PTotalRxPacketsStr },
    { ODI_STAT_UINT32,  &PIgnoredRxPackets,  &PIgnoredRxPacketsStr },
};
PS_STATS_TABLE  PS_StatsTable = {2,0,NUM_GENERIC_COUNTERS,
                                PGenericCounters,  0 , NULL };

```

Note



The strings for the protocol stack counters are initialized by language enabling code.

STAT_TABLE_ENTRY Field Descriptions

Table 2-3

Generic Counters Array STAT_TABLE_ENTRY

Size	Label	Description
UINT32	PTotalTxPackets	This field has the total number of <i>SendPacket</i> requests made to the LSL.
UINT32	PTotalRxPackets	This field contains the total number of incoming packets that were consumed by the protocol stacks.
UINT32	PIgnoredRxPackets	This field has the total number of times the protocol receive handler was called with lookahead data and the protocol stack did not return a receive ECB to the MLID to receive the packet.

2-8 ODI Specification: Protocol Stacks and MLIDs (C Language)

Chapter Overview

This chapter describes registering and binding prescan, bound, and default protocol stacks. This chapter also covers the information you need to know in order to chain protocol stacks.

You should review this chapter before writing the protocol stack initialization routine.

Protocol Stack Initialization Steps

Protocol stack initialization involves the following general steps:

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

If your protocol stack is to be resident, it should free all the memory it used to hold the initialization code and data before turning resident. (The process of freeing initialization code and data is a design implementation decision.)

Locating the LSL

The LSL module must reside in the system before the user can load any protocol stacks. On some platforms, such as the NetWare server, the LSL may already be preloaded. Refer to "Locating the LSL" in Chapter 10, "LSL Support Routines".

Registering Protocol Stacks with the LSL

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

- Gives the LSL pointers to the protocol stack's receive handler and to the protocol stack's control handler.
- Assigns a unique Stack ID (SID) to the protocol stack.

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

Table 3-1
Receive and Control Handlers and the Stack ID

Actor/Agent	Action
LSL	1 Calls the protocol stack's receive handler whenever a packet intended for that particular protocol stack is received.
Applications and LSL	2 Call the protocol stack's control handler to obtain configuration information and to issue defined control functions.
LSL	3 Uses the Stack ID (SID) to track the protocol stack. The LSL assigns the SID when the protocol stack registers.

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

If the protocol stack is using the prescan or default receive methods (see *Chapter 5: Protocol Stack Packet Reception*), it must register using **CLSL_RegisterPrescanChain** or **CLSL_RegisterDefaultChain**, respectively. The LSL does not assign a Stack ID (SID) to default or prescan

protocol stacks; however, pointers to the protocol's receive and control handlers are still necessary.

Determining Which Logical Board(s) to Service

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

Explicit Method

In the explicit method, the user explicitly specifies which logical boards the protocol stack binds to. We suggest that for each protocol entry the user specify a "bind" entry in the appropriate platform configuration file that looks like the following:

```
bind "stack name" <MLID Short Name> [ Board Number <Protocol ID>]
```

Note



The method of specifying the binding of a protocol stack to a logical board is entirely up to the protocol stack developer. The line specifying the binding information can be passed to the protocol stack or some entity to parse and the resultant binding information as to the stack and which logical board it is bound to is passed in the **CLSL_BindStack** or **CLSL_BindProtocolToBoard** call.

The first thing a protocol stack does is verify whether a specified board exists and whether a Protocol ID (PID) is available for the protocol that uses that particular board. The protocol stack can verify that a board exists by calling the **CLSL_GetMLIDControlEntry** function. If the board is valid, the protocol stack determines whether a PID exists for the protocol on that particular board by calling **CLSL_GetPIDFromStackIDBoard**. If a PID is not present for that protocol, the protocol stack adds a PID to use or stops the initialization procedure.

Dynamic Method

If no bind information is specified in the appropriate platform's configuration file, the protocol stack scans for board(s) to bind to. The protocol stack scans

through all the possible board numbers, starting with board 0, and calls the **CLSL_GetMLIDControlEntry** function, which returns whether or not the specified board number exists. The protocol stack continues scanning and calling **CLSL_GetMLIDControlEntry** until the message *ODISTAT_NO_MORE_ITEMS* is returned. The protocol stack then knows that no more boards exist at any higher board numbers. When the protocol stack encounters an active board, the stack queries the LSL for a PID by calling the **CLSL_GetPIDFromStackIDBoard** support function. If the protocol stack cannot find a board that has a PID for it, the protocol stack adds a PID to use or stops the initialization procedure.

Adding Protocol IDs

You should write your protocol stacks so that they are LAN medium and frame type unaware. Because PID values are determined by the frame type and LAN medium where they are used, the protocol stack does not interpret the PID. Usually, the user of your protocol stack will enter your protocol stack's PID with the configuration information for each frame type and board combination. As discussed in the "Explicit Method" and "Dynamic Method" sections above, the protocol stack obtains the PID by calling **CLSL_GetPIDFromStackIDBoard**. Your protocol stack can register an appropriate PID for each board it binds to. This procedure eases system configuration for the user because the user does not need to enter any PID values for your protocol stack.

To add a PID, the protocol stack must know the common PID value for each of the frames currently defined (for example, ETHERNET_802.2, TOKEN_RING, NOVELL_RX-NET, FDDI_SNAP, etc.). See *ODI Specification Supplement: Frame Types and Protocol IDs* for a list of the current frame types. Before the protocol stack adds the PID, it determines whether a PID has previously been registered for that stack on that particular board. The protocol stack determines this by calling **CLSL_GetPIDFromStackIDBoard**. If this call returns a PID, the protocol stack uses it. If a PID is not returned, the protocol stack looks at the MLID's configuration table *MLIDCFG_FrameID* field to determine whether the protocol stack has a known PID for that frame type. If the protocol stack does have a known PID for that frame type, the protocol stack calls **CLSL_AddProtocolID**. If the protocol stack does not have a known PID for that frame type (for example, perhaps a new frame type is being used), the protocol stack returns an error that indicates a PID must be entered with the configuration information.

3-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

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

- A PID for the protocol stack to use with a particular frame type has not been previously registered (determined by **CLSL_GetPIDFromStackIDBoard**).
- The protocol stack is internally aware of a PID for the board's frame type. (For example, the IPX PID on frame type ETHERNET_802.2 is usually 0xE0, and the TCP/IP PID on frame type Ethernet_II is usually 0x800.)

Multiple Board Support

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

Obtaining Protocol ID Values

The protocol stack usually obtains a Protocol ID (PID) value when the stack is determining which board it can bind to. The

CLSL_GetPIDFromStackIDBoard function returns the assigned PID for that protocol stack on the specified board. A protocol stack only needs the PID value(s) when it sends packets and when it registers with the LSL (see *Chapter 11: Overview of the MLID*). A protocol stack must not interpret the PID in any way so that LAN medium and frame type independence is maintained. The protocol stack simply saves the obtained PID(s) for use when transmitting packets.

Customizing the Protocol Stack

One of the ODI specification's goals is to keep the protocol stack interface to the LSL and the underlying MLIDs as general and independent of issues specific to LAN adapters as possible. However, there are still a number of issues that must be dealt with during initialization. This means that your protocol stack must be customized to the particular capabilities of the underlying MLIDs and the associated LAN adapters.

Line Speed

Most LAN media provide high speed data transfer rates (for example, 2M to 100M bits per second). Protocol stacks that retry transmit operations when they do not receive an expected acknowledgment within a specific period of time might have to customize time-out values so they are appropriate to the speed of the underlying physical LAN medium. Time-out values can usually be small because transmission and reception acknowledgment on most LAN media is very fast. However, keep in mind that the underlying medium might have relatively low data rates (such as 2400 baud). Unless the protocol stacks increase their internal time-out values when they are using a slow network, excessive and unneeded transmit retries will occur and adversely affect operation.

Measuring Effective Network Performance

Protocol stacks can use two fields in the MLID's configuration table to measure the effective performance of a particular network: *MLIDCFG_TransportTime* and *MLIDCFG_LineSpeed* (see *Chapter 12: MLID Data Structures* for more details regarding these fields).

MLIDCFG_TransportTime Field

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

MLIDCFG_LineSpeed Field

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

Maximum Packet Size

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

3-6 ODI Specification: Protocol Stacks and MLIDs (C Language)

MLIDCFG_MaxFrameSize Field

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

MLIDCFG_BestDataSize Field

The *MLIDCFG_BestDataSize* field represents the maximum number of data bytes that the protocol stack can send and receive when it does not use certain low-level headers (such as the source routing headers in Token-Ring).

MLIDCFG_WorstDataSize Field

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

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

Multicast Support

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

In order for a LAN adapter to become a member of a multicast group, the group's multicast address has to be enabled on the adapter so that any packets received by it will be passed to the host computer and not discarded at the hardware level. Protocol stacks determine whether an MLID supports multicast

by examining the *MLIDCFG_ModeFlags* field in the MLID configuration table (see *Chapter 12: MLID Data Structures*).

Multicast support and the format of the multicast addresses is LAN medium dependent, and some LAN media do not support any type of multicast capability. A protocol stack that utilizes multicasting must determine whether the MLID is using noncanonical or canonical addressing by examining the MLID configuration table entry *MLIDCFG_ModeFlags* (see *Chapter 12: MLID Data Structures*). (Canonical addressing is a “generic” form of addressing that is media independent See *ODI Specification Supplement: Canonical and Noncanonical Addressing* for more information.) If the MLID is using noncanonical addressing, the protocol stack must determine the LAN medium type of the underlying LAN adapter and use the appropriate multicast address. MLIDs have control functions that add and remove multicast addresses (see *Chapter 15: MLID Control Routines*).

If a protocol stack does not know the format of the LAN medium’s multicast address, or the LAN medium does not support multicasts, the protocol stack simply uses real broadcasts (0xFF FF FF FF FF FF) instead of multicasts.

Protocol stacks that use multicast addresses should also allow the user to specify the multicast addresses that the protocol stack will use for a particular board. This capability is usually accomplished by using a custom keyword at load time. This allows the protocol stack to use correctly formatted multicast addresses for LAN mediums other than the ones that the protocol’s multicast code was originally written to.

Receive Lookahead

As part of customization, your protocol stack informs the underlying MLID about the amount of receive lookahead data it must have in order to properly process received packets. (The **SetLookAheadSize** MLID control function is discussed in detail in *Chapter 15: MLID Control Routines*. It is noted here in order to specify that your protocol stack sets its needed lookahead size during this phase of initialization.)

Binding to Logical Boards

One of the last things a protocol stack must do before it becomes fully operational is to bind to the predetermined board(s). Binding the protocol stack enables the LSL to route incoming packets destined for that protocol stack to

its receive handler. Note, a protocol stack can send packets without being bound to any board(s).

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

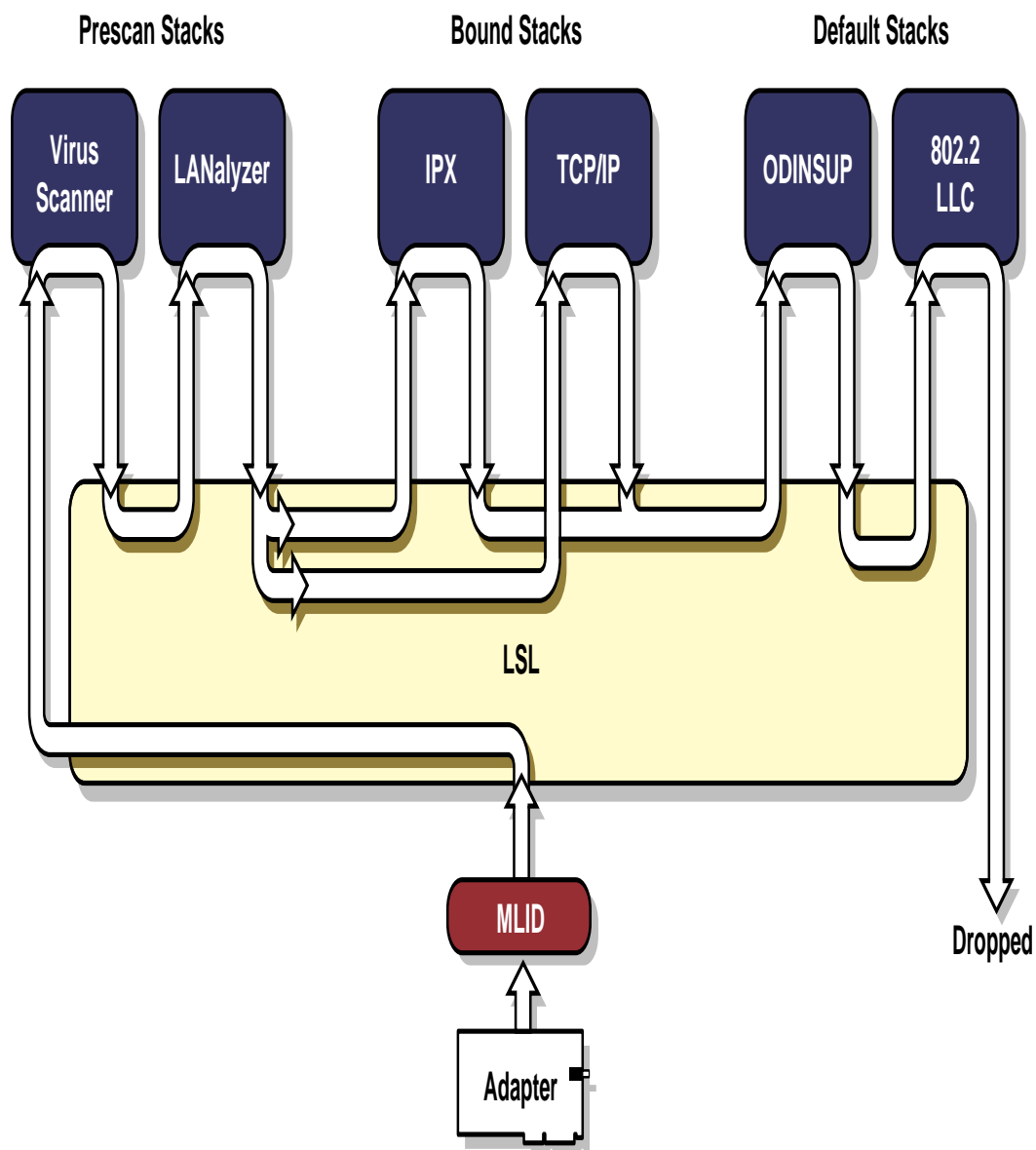
If the protocol stack uses the prescan or default receive methods, this step is not necessary. Packet reception for these types of protocol stacks begins when the *CLSL_RegisterPreScanChain* or *CLSL_RegisterDefaultChain* commands are issued.

Chaining Prescan and Default Protocol Stacks

Prescan and default protocol stacks can be chained, so the received and transmitted packets flow through the chained stacks in a prescribed order. Figure 1.1 illustrates sample receive packet flow through a system with chained prescan and default stacks.

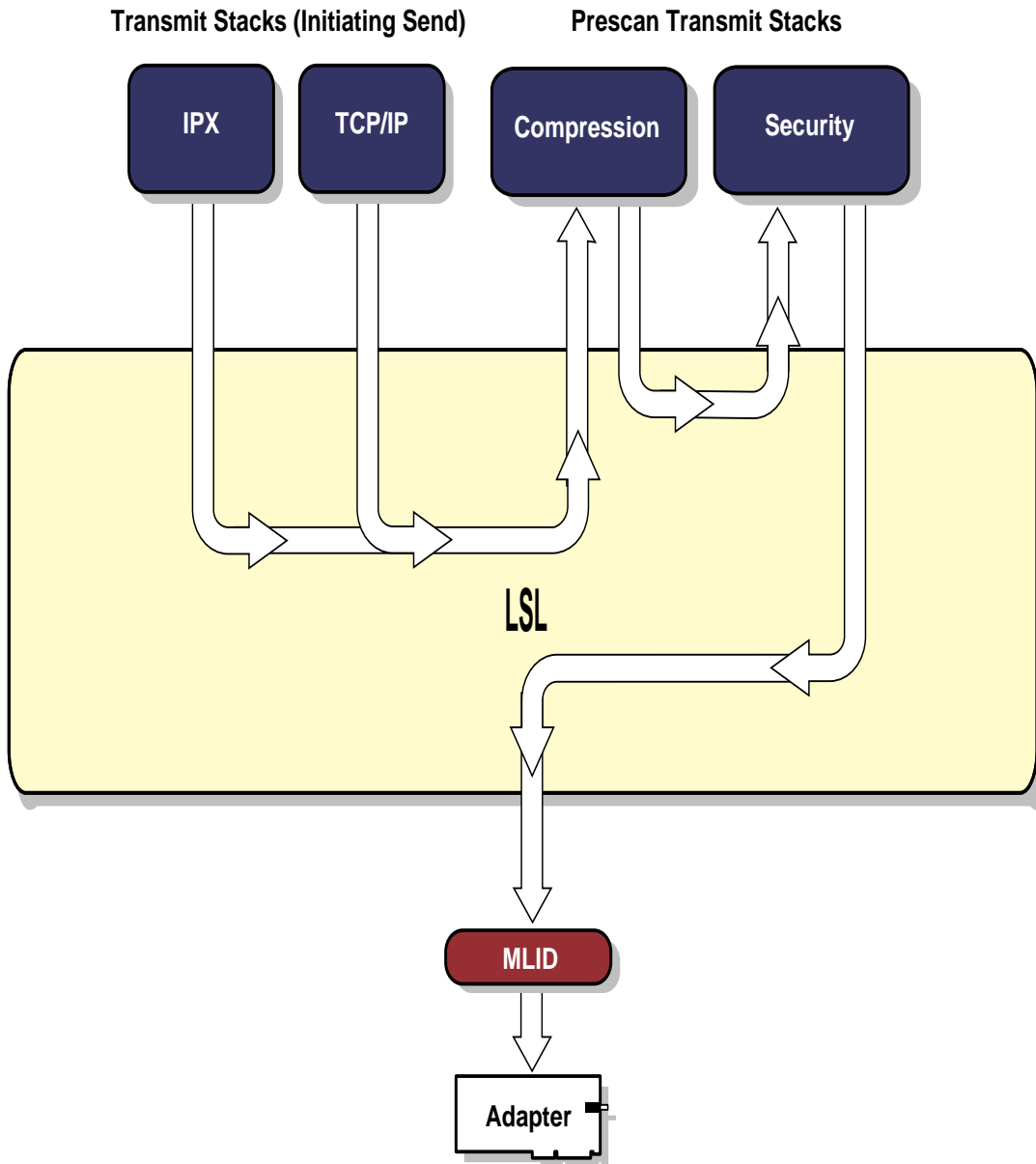
The LSL adds the chained stacks in the chain position order that the stacks request. If a stack must be first or must be last in the chain, and another stack that also must be first or must be last already occupies that position, the attempt to load the second stack returns an error message.

Figure 3-1
Receive Prescan and Default Protocol Stack
Chaining Overview



3-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

Figure 3-2
 Transmit Prescan Protocol Stack
 Chaining Overview



Note



If more than one prescan stack is loaded on a machine and each prescan stack attaches it's own protocol header to the transmit data, the transmit load order of the stacks on the transmitting machine will dictate the receive load order of the stacks on the receiving machine. For example, if on the transmitting machine you have three prescan stacks loaded in the order A B C, then in order to process the headers properly, you must load the prescan stacks on the machine receiving the packets in the order C B A. This is not a problem if both the transmit and receive PreScan chain stacks are loaded at the same time using **CLSL_RegisterPreScanChain**.

Final Initialization

At this point the protocol stack is fully operational. Whenever possible, we recommend that your module display, to the user, the logical board(s) that it is bound to and servicing. The display should include the MLID's short name and any other information that the user might find useful.

chapter **4** *Protocol Stack Packet Reception*

Chapter Overview

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

Protocol Stack Packet Receive Operation

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

Receive Routine Events

Table 5-1 lists the events that must occur during a protocol stack receive routine.

Table 4-1
Protocol Stack Receive Routine

Actor/Agent	Action
MLID	1. When a packet is received, a <i>LOOKAHEAD</i> structure is filled out and the MLID LSL support routine CLSL_GetStackECB is called to obtain from a protocol stack a receive buffer for the packet data. (For more information on CLSL_GetStackECB , see Chapter 10, "LSL Support Routines").
LSL	2. Determines whether bound, prescan, or default protocol stacks will be receiving the packet. 3. Calls the protocol stack that is to receive the data and passes to the protocol stack a pointer to the <i>LOOKAHEAD</i> structure describing the received packet.
Protocol stack	4. Determines whether to receive the packet. If the <i>LOOKAHEAD</i> structure has a prefilled ECB associated with it and the protocol stack utilizes the prefilled ECB (the protocol stack responds <i>ODISTAT_SUCCESS_TAKEN</i>), step 6 is ignored. If a protocol stack consumes the prefilled ECB by returning <i>ODISTAT_SUCCESS_TAKEN</i> with the <i>LkAhd_ReturnedECB</i> field set to NULL, it returns the prefilled ECB to the LSL using CLSL_ReturnECB , after it has finished processing it. 5. Builds an ECB describing a set of receive buffers that the packets are dispersed into. 6. Signals to the LSL that this protocol stack will consume the packet.
LSL	7. If the <i>LOOKAHEAD</i> structure has a prefilled ECB associated with it and the protocol stack utilizes the prefilled ECB, it assumes control of the ECB. The protocol stack will return the ECB to the LSL using CLSL_ReturnECB after it has finished processing it and returns the status <i>ODISTAT_SUCCESS_TAKEN</i> through the LSL to the MLID, and steps 8 and 9 are ignored. If the <i>LOOKAHEAD</i> structure has a prefilled ECB associated with it and the protocol stack does not utilize the prefilled ECB (the protocol stack returns an ECB to be filled), the LSL will disperse the receive packet data from the prefilled ECB into the receive buffers of the ECB supplied by the protocol stack. The LSL will then place the ECB returned by the protocol stack onto the LSL hold event queue, and steps 8 and 9 are ignored. The MLID is returned the status <i>ODISTAT_SUCCESS_TAKEN</i> .

Table 4-1
Protocol Stack Receive Routine

Actor/Agent	Action
	<p>Note: If an MLID presents a prefilled ECB in the LOOKAHEAD structure, no further processing for that lookahead indication is required apart from incrementing appropriate counters dependent on the returned status. For example, if the status returned is <i>ODISTAT_OUT_OF_RESOURCES</i>, the MLID counter <i>MNoECBAvailableCount</i> is incremented by the MLID.</p> <p>If the LOOKAHEAD structure has no prefilled ECB associated with it and the protocol stack returns an ECB to be filled (<i>ODISTAT_SUCCESSFUL</i>), steps 8 through 11 are executed.</p> <p>Note: The <i>LkAhd_ReturnedECB</i> field may be NULL when <i>ODISTAT_SUCCESSFUL</i> is returned, if the protocol stack was able to retrieve the necessary information from the LOOKAHEAD data. Returning a NULL ECB allows the MLID to adjust its statistics accordingly; for instance, if the packet was not rejected.</p> <p>If the protocol stack returns <i>ODISTAT_OUT_OF_RESOURCES</i>, the packet is discarded and step 10 is executed.</p>
MLID	<ol style="list-style-type: none"> 8. Copies the packet data into the provided data buffers. 9. The ECB is placed onto the LSL hold event queue, CLSL_HoldEvent, or calls CLSL_FastHoldEvent and skips steps 10 and 11. 10. Calls CLSL_ServiceEvents.
LSL	<ol style="list-style-type: none"> 11. Dispatches the defined ESR (event service routine), signaling that packet reception is complete.

Protocol Stack Packet Reception Methods

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

- bound
- prescan
- default

Bound Protocol Stack

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

Prescan Protocol Stack

Prescan protocol stacks look at all packets received by a particular logical board (adapter and frame type combination). The protocol stack can consume select packets and allow others to be passed to the appropriate bound or default protocol stack.

Default Protocol Stack

Default protocol stacks receive packets not consumed by the prescan and bound protocol stacks. A default protocol stack can choose to consume or reject a packet. If the packet is rejected, it is discarded from the system.

Choosing a Packet Reception Method

We strongly discourage you from using the prescan receive or default method of packet reception for the following two reasons:

- A protocol stack might conflict with another protocol stack using the same method of packet reception. This would prevent the use of both protocols with the same board.

4-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

- A protocol stack parses the packet header to determine if the packet is the correct type. Therefore, a protocol stack might receive a packet that passes the protocol stack's acceptance tests, but in reality is not the correct type. This could cause unpredictable results in the network station.

The prescan receive and default reception methods must only be used for specialized protocol stacks that must receive packets having a large range of PIDs. For example, the 802.2 protocol stack must receive packets with any Destination SAP. Protocol stacks that provide a data-link layer interface to the network layer protocol stacks are candidates for using prescan receive or default receive methods. Also, protocol stacks that provide compression services are candidates for prescan receive and default protocol stacks. A receive monitor can also be implemented as a prescan receive stack by registering for all its relevant destination types including packets containing errors.

Multiple Chained Protocol Stacks

Multiple chained protocol stacks for each logical board can use the prescan receive or default reception method. The bound method allows multiple protocol stacks to receive packets for a board. This method uses the PID found in every packet to multiplex and demultiplex protocol packets.

MAC Packet Reception

To receive MAC frames, bound protocol stacks must register using either the MACTOK or MACFDI protocol ID. All three reception methods (bound, prescan, and default) must set their filter mask to include DT_MAC_FRAME.

Receive Lookahead

The receive method known as "receive lookahead" entails passing the beginning portion of the packet up to the protocol stack. In most cases, this allows the receive packet data to be dispersed directly into the application buffers. This is the optimal situation because the receive data only crosses the host's bus once, and this method allows unwanted packets to be rejected without ever leaving the adapter's buffers.

Receive Handler

Regardless of whether the protocol stack is bound, prescan, or default, the protocol stack is passed lookahead data whenever its receive handler is invoked. This data is used to determine into which receive buffers (if any) the data is placed. (Receive buffers can be fragmented.) If the protocol stack wants to consume the packet, it must build an ECB that describes the receive buffers and then returns that ECB to the MLID. The MLID uses the ECB's description of the receive buffers to move the data from the network adapter into the described protocol receive buffers. When the MLID has completed the data move, it passes the ECB to the LSL for event completion. All protocol stack receive handlers must not block on the lookahead indication.

Alternatively, if the lookahead has an ECB associated with it (*LkAhd_PrefilledECB* is not NULL), the protocol stack can accept the packet by signaling that it has accepted the packet and taken the associated ECB, which it returns to the LSL later. If a protocol stack performs this operation, it must queue the ECB (using **CLSL_HoldEvent**) after placing an address to be called in the *ECB_ESR* field or in an internal queue for processing at a later point, since the lookahead indication is usually made at privileged time by an MLID. If the protocol stack chooses not to use the provided ECB in the LOOKAHEAD structure and returns its own ECB to be filled, the LSL can perform the prefilled ECB to stack ECB data copy and, by so doing, simplify the MLID's operation.

After placing the ECB on the LSL's hold queue (using **CLSL_HoldEvent**), the event is completed when the MLID issues the LSL support command **CLSL_ServiceEvents**. This calls the ECB's ESR and allows the protocol stack to process the packet. (For more information on the **CLSL_ServiceEvents**, **CLSL_HoldEvent**, and **CLSL_FastHoldEvent** routines see Chapter 10: "LSL Support Routines".)

LOOKAHEAD Structure

The following LOOKAHEAD structure is given to the protocol stack's receive handler.

```
typedef struct _LOOKAHEAD_
{
    ECB            *LkAhd_PrefilledECB;
    UINT8          *LkAhd_MediaHeaderPtr;
    UINT32         LkAhd_MediaHeaderLen;
    UINT8          *LkAhd_DataLookAheadPtr;
```

4-6 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

UINT32      LkAhd_DataLookAheadLen;
UINT32      LkAhd_BoardNumber;
UINT32      LkAhd_PktAttr;
UINT32      LkAhd_DestType;
UINT32      LkAhd_FrameDataSize;
UINT16      LkAhd_PadAlignBytes1;
PROT_ID     LkAhd_ProtocolID;
UINT16      LkAhd_PadAlignBytes2;
NODE_ADDR   LkAhd_ImmediateAddress;
UINT32      LkAhd_FrameDataStartCopyOffset;
UINT32      LkAhd_FrameDataBytesWanted;
ECB         *LkAhd_ReturnedECB;
UINT32      LkAhd_PriorityLevel;
void        *LkAhd_Reserved;
} LOOKAHEAD;

```

Field descriptions:

LkAhd_PreFilledECB

If this field is not NULL, it contains a pointer to a completely filled ECB (obtained from the LSL), which contains data referenced by the lookahead. Only the *LkAhd_MediaHeaderPtr* and *LkAhd_MediaHeaderLen* fields are valid.

The lookahead method allows bus master adapters to provide a received packet with only one data copy. The protocol stack can elect to accept the already filled ECB and return the ECB to the LSL at a later point, or it can provide its own ECB to be filled. The *MM_PREFILLED_ECB_BIT* bit is implemented in the MLID configuration table's *MLIDCFG_ModeFlags* field to indicate adapters that always supply prefilled ECBs in the *LkAhd_PreFilledECB* field, which allows protocol stacks to optimize their receive handler at initialization time.

Note



Only ECBs provided by the LSL can be indicated in this field.

LkAhd_MediaHeaderPtr

Pointer to a buffer containing the complete low-level media header. The protocol stack typically does not look at the low-level header information.

LkAhd_MediaHeaderLen

Contains the length of the media header pointed to by *LkAhd_MediaHeaderPtr*.

Protocol Stack Packet Reception 4-7

LkAhd_DataLookAheadPtr

Pointer to a buffer containing the start of the frame's data, in other words, the protocol's header and data.

Note



The *LkAhd_MediaHeaderPtr* buffer is not guaranteed to immediately precede the *LkAhd_DataLookAheadPtr* buffer (for example, an 802.3/802.2 MAC header and the following frame data might not necessarily be in contiguous memory).

LkAhd_DataLookAheadLen

Contains the length of the buffer pointed to by *LkAhd_DataLookAheadPtr*. This value is normally the MLID's currently configured lookahead size. Adapters that have the received packet available in memory, such as shared RAM, can set this field to the length of the data packet **available**, because *LkAhd_DataLookAheadPtr* points at the packet located in memory. Note, you need to allow for the case of a packet wrapping over the end of the shared memory space with the rest of the packet appearing at the start of the shared memory space, etc.

If the received packet's data length is less than the MLID's configured lookahead size, this field is set to the actual packet data length. A protocol stack must verify that this field is at least the minimum length required for the protocol stack. Note, if the contents of this field equal those of *LkAhd_FrameDataSize*, you can assume that the packet has been fully received and the contents of *LkAhd_PktAttr* are valid.

LkAhd_BoardNumber

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

LkAhd_PktAttr

Contains the attributes of the received packet. The following defines the packet attribute bits.

Value	Description
PAE_CRC_BIT	CRC error—for example, frame check sequence (FCS) error.
PAE_CRC_ALIGN_BIT	CRC and frame alignment error.
PAE_RUNT_PACKET_BIT	Runt packet.
PAE_TOO_BIG_BIT	Packet larger than allowed by the media.
PAE_NOT_ENABLED_BIT	Received packet for a frame type not supported (logical board not registered for the frame type of the received packet). A board number associated with the physical adapter is placed in the <i>LOOKAHEAD</i> structure.
PAE_MALFORMED_BIT	Malformed packet—for example, the packet size is smaller than the minimum size for the media header, such as incomplete MAC header.
	Contents of the length field in an Ethernet 802.3 header is larger than the total packet size.
PAE_NO_COMPRESS_BIT	Do not decompress the received packet.
PA_NONCAN_ADDR_BIT	Implies that the address present in <i>LkAhd_ImmediateAddress</i> is in noncanonical format.

If no error bits are set, the received packet was received without error and the data contained within can be used. All undefined bits are cleared. If any error bit is set, the *LkAhd_DestType* field's *GlobalError* bit will also be set.

If the value in *LkAhd_FrameDataSize* is -1, the error status bits are invalid; however, the value *PA_NONCAN_ADDR_BIT* indicates that noncanonical addressing is still valid.

LkAhd_DestType

Contains bits that indicate the type of received packet. The following are the bit definitions.

Value	Description
DT_MULTICAST	<i>Multicast:</i> The packet was destined to a subset of group addresses on the physical network that the MLID has been programmed to support.
DT_BROADCAST	<i>Broadcast:</i> The packet was destined to all nodes on the physical network. Note: on receiving a broadcast both b0 and b1 are set to 1, since a broadcast address is also a group address.
DT_REMOTE_UNICAST	<i>UnicastRemote:</i> The packet was directly destined to another workstation on the physical network. Generally, this bit is set only after the MLID has been entered into promiscuous mode or has received a packet due to source routing.
DT_REMOTE_MULTICAST	<i>MulticastRemote:</i> The packet was destined to a subset of group addresses on the physical network that the MLID has not been programmed to support. Generally, this bit is set only after the MLID has been entered into promiscuous mode.
DT_SOURCE_ROUTE	<i>SourceRoute:</i> 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.
DT_ERRORED	<i>GlobalError</i> : Packet contains errors. See <i>LkAhd_PktAttr</i> as to specific error. This is an exclusive bit. If set, all other bits must be 0. This value supersedes <i>SourceRoute</i> and <i>MacFrame</i> .
DT_MAC_FRAME	<i>MacFrame</i> : Packet is a non-data frame(for example, the MAC layer frame). This is an exclusive bit, if set, all other bits must be 0. Note, MAC frames by definition are not source routable.
DT_DIRECT	<i>Direct</i> : The packet was destined to this station only.
DT_8022_TYPE_I	The received packet is an 802.2 Type I frame.
DT_8022_TYPE_II	The received packet is an 802.2 Type II frame.
DT_RX_PRIORITY	<i>RxPriorityFrame</i> : The received packet is a priority packet. This is only valid for topologies that support a distinction in priority levels. When this bit is set, the <i>LkAhd_PriorityLevel</i> field will contain the priority level of the frame. This bit is not set if the received frame is at the normal priority level or lower.



For 802.2 frame types, the received packet's DSAP is returned in the last byte of the *LkAhd_ProtocolID* field with all other bytes set to 0. For an ETHERNET_II frame, neither the 802.2 Type I (0x0000 0100) nor the 802.2 Type II (0x0000 0200) values are set, which implies a non 802.2 frame.

In promiscuous mode with MAC frames enabled, all MAC frames are received, including those transmitted by the MLID, generated by the MLID's hardware, and transmitted by protocol stacks performing raw sends. For example, Token-Ring adapters can receive management frames but not data frames. The *MacFrame* value is set in the *LkAhd_DestType* field for these frames. The PID associated with MAC layer frames is **MACTOK** for Token-Ring management frames and **MACFDI** for FDDI management frames. Refer to the "Protocol Stack Packet Reception Methods" section of this chapter for more information about MAC packet reception.

All undefined bits are set to 0.

LkAhd_FrameDataSize

Contains the total number of data bytes in the received packet.

Note



If the content of this field is not UNUSED, the entire received packet and its error status is available from the adapter.

Not every LAN driver knows the exact size of a received frame when **CLSL_GetStackECB** is called (RX-Net or pipelined LAN adapters are examples of this). If the size of the received packet or its error status is not known, the protocol stack will need to check its error status and size when the packet has been fully received. This condition is indicated by setting the contents of this field to -1.

The *MM_DATA_SZ_UNKNOWN_BIT* bit is defined in the MLID configuration table *MLIDCFG_ModeFlags* field to indicate an adapter that can set *LkAhd_FrameDataSize* to -1, which allows protocol stacks to optimize their receive handler at initialization time.

LkAhd_ProtocolID

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

LkAhd_ImmediateAddress

Contains a representation of the address of sending station's node address, that was embedded in the low-level media header.

Note



If the MLID is using canonical addressing, the address in the *LkAhd_ImmediateAddress* field is in canonical form.

LkAhd_FrameDataStartCopyOffset

4-12 ODI Specification: Protocol Stacks and MLIDs (C Language)

Contains the offset from the start of the frame data to start copying data from. The returned value must not exceed the value presented in the *LkAhd_DataLookAheadLen* field.

LkAhd_FrameDataBytesWanted

Contains the number of bytes of frame data to move into the receive buffers starting after the number of data bytes skipped, which is returned in *LkAhd_FrameDataStartCopyOffset*. If this field contains -1, the entire packet is copied into the receive buffers. This field is usually set by the protocol stack when an ECB is returned to be filled.



This value can be larger than the number of data bytes in the frame, but if it is larger, it is the responsibility of the MLID to ensure that no errors occur. In other words, if the *LkAhd_FrameDataBytesWanted* field value is larger than the number of data bytes copied from the frame, the data length field (found in the returned ECB structure) will be adjusted and filled in correctly by the MLID.

LkAhd_ReturnedECB

If this field is not NULL, it contains a pointer to an ECB provided by the protocol stack that is to be filled with the packet's data.



MLIDs rely on the value returned by **CLSL_GetStackECB** and this field to decide whether a protocol stack accepts the packet and if it returns an ECB to copy the packet into.

LkAhd_PriorityLevel

This field contains the priority level of the received packet. This field is only valid if the *RxPriorityFrame* bit in the *LkAhd_DestType* field is set and if the priority level is higher than the normal priority level. This value must not exceed the value in the MLID configuration table's *MLIDCFG_PrioritySup* field.

LkAhd_Reserved

Reserved for future use.

The LOOKAHEAD structure is valid only in the context of the receive handler.

The amount of receive lookahead data needed by a protocol's receive handler is usually different for each type of protocol stack (prescan, bound, or default). The protocol stack can configure the amount of receive lookahead data the MLID provides by invoking the **SetLookAheadSize** MLID control function as part of the protocol's initialization. **SetLookAheadSize** informs the MLID that the protocol stack needs the specified number of packet data bytes to properly determine into which receive buffers a packet is placed. The lookahead size

value can be any value between 0 and 128 bytes inclusive. The requested size does not include any room for possible media headers, because the LAN driver will internally adjust the lookahead size value to include the LAN medium's worst case low-level media header size. If the requested size is larger than the current lookahead size, the MLID will use the new value. However, if the requested size is smaller than the current size, the MLID will not decrease the current size. (See *Chapter 15: MLID Control Routines* for information regarding invoking the **SetLookAheadSize** MLID control function.)

Protocol stacks must not assume that the lookahead data is valid if the lookahead field *LkAhd_FrameDataSize* is -1. The protocol stack must wait until the filled ECB's ESR is called. Then, from within the ESR, the protocol stack must check the received packet's error status. In other words, do NOT copy the packet data if the MLID's lookahead field *LkAhd_FrameDataSize* is -1, and do not assume that the data is valid. Do not make any permanent decisions internal to the protocol stack that cannot be undone later when the ECB's ESR is called. This is due to pipelined adapters presenting data to the protocol stacks before they have finished completely receiving the packet. Hence, pipelined adapters cannot inform protocol stacks as to the size of the received packet or whether the packet contains errors—for example, pipelined adapters cannot inform protocol stacks of CRC errors until after the packet has been fully received.

An exception to this is when the received packet's size is less than the protocol stacks configured lookahead size or is less than the MLID's configured lookahead size. Then the packet can be assumed to have been fully received, and the *LkAhd_PktAttr* field contains the status of the received packet. This condition is reported when the number of data bytes reported for the *LkAhd_DataLookAheadLen* field in the LOOKAHEAD structure is **less** than the configured lookahead size. For protocol stacks that utilize small packet reception through the LOOKAHEAD structure, the protocol stacks set their lookahead requirements to their requirements plus one to ensure that they can continue this function with pipelined adapters. Also a packet can be assumed to have been fully received if the LOOKAHEAD structure's *LkAhd_DataLookAheadLen* and *LkAhd_FrameDataSize* fields are equal.

If the LOOKAHEAD structure has a prefilled ECB associated with it (*LkAhd_PrefilledECB* is not NULL), the protocol stack can accept the packet. The protocol stack accepts the packet by queuing the prefilled ECB and returning *ODISTAT_SUCCESS_TAKEN* to the LSL. (By queuing the prefilled ECB instead of processing it immediately, the MLID is not required to wait for the processing to be completed.) Once *ODISTAT_SUCCESS_TAKEN* is returned by the protocol stack receive handler, it is the protocol stack's

4-14 ODI Specification: Protocol Stacks and MLIDs (C Language)

responsibility to call the receive complete handler and to return the prefilled ECB using **CLSL_ReturnECB**.

If the protocol stack chooses not to process the prefilled ECB in the LOOKAHEAD structure and returns its own ECB (a stack ECB) to be filled, the LSL can perform the prefilled ECB to stack ECB copy of data. This process simplifies the operation of the MLID—for example, a bus mastering MLID can provide a lookahead indication and not worry about touching the data again. If the protocol stack provides a stack ECB, the LSL can fill that ECB from the prefilled ECB, place the stack ECB on its service events queue for processing, and return the prefilled ECB to the LSL buffer pool. If the protocol stack rejects the packet, the protocol stack returns *ODISTAT_OUT_OF_RESOURCES* and the LSL returns the prefilled ECB to the LSL buffer pool.



If *LkAhd_PrefilledECB* is not NULL, only the *LkAhd_MediaHeaderPtr* and *LkAhd_MediaHeaderLen* fields are valid. All other fields are referenced by their ECB equivalents—for example, *LkAhd_BoardNumber* is referenced by *ECB_BoardNumber*.

Protocol Receive Handler for Bound Stacks

The protocol receive handler for bound stacks is invoked when a packet is received and the LSL determines that the packet is intended for the protocol stack.

Syntax

```
#include <odi.h>

ODISTAT ( *StackRxHandler )
( LOOKAHEAD *LkAhead );
```

Input Parameters

LkAhead

Pointer to the received LOOKAHEAD structure.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack has returned a pointer in <i>LkAhd_ReturnedECB</i> to an ECB to be filled with the packet.
ODISTAT_SUCCESS_TAKEN	The protocol stack has accepted the packet and has taken the prefilled ECB associated with the LOOKAHEAD structure.
ODISTAT_OUT_OF_RESOURCES	The protocol stack will not receive the packet, in other words, rejects the packet.

Remarks

This definition applies to bound stacks. The protocol stack examines the lookahead data as described by the LOOKAHEAD structure and returns an ECB when appropriate.

The LOOKAHEAD structure and its fields are only valid in the context of this function. This routine must complete quickly, since it executes during privileged time. LAN driver functions must not be invoked inside this function (for example, **CLSL_SendPacket** must not be invoked inside this function).

The ECB must have the following fields and descriptors set before it returns: *ECB_ESR*, *ECB_FragmentCount*, and fragment descriptors. You can specify more than one fragment descriptor. The *ECB_FragmentCount* field must not be set to 0 and must not exceed 16 ($0 < ECB_FragmentCount \leq 16$).

The LOOKAHEAD structure must have the *LkAhd_FrameDataBytesWanted*, *LkAhd_FrameDataStartCopyOffset*, and *LkAhd_ReturnedECB* fields set with appropriate values if the receive packet is accepted.

If the protocol stack also requires the *ECB_ProtocolID*, *ECB_BoardNumber*, and *ECB_ImmediateAddress* fields to be filled in, the protocol stack fills it in with the protocol identifier board number and immediate address supplied in the LOOKAHEAD structure. If an ECB is returned from this function, the ESR is called at a later time, which signals that the packet data has been transferred to the described receive buffers either successfully or with an error.

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

Note



Protocol receive handlers are not called with a LOOKAHEAD structure if the structure contains errors unless they register to receive packets with errors. Pipelined adapters are an exception to this and cannot know the received status of a packet when the protocol's receive handler is called, refer to the LOOKAHEAD structure definitions.

The filter mask for bound protocol stacks defaults to receiving direct, multicast direct, and broadcast addressed packets when they register. If any other filter bit is needed or one of the above needs to be cleared, a protocol stack can use **CLSL_ModifyStackFilter**. For more information, see **CLSL_RegisterStack** in *Chapter 10: LSL Support Routines*.

The following table illustrates the event sequence of the receive handler.

Table 4-2
Receive Handler Event Sequence

Actor/Agent	Action
MLID	1. Provides a <i>LOOKAHEAD</i> structure to the LSL (<i>CLSL_GetStackECB</i>).
LSL	2. The LSL determines the correct protocol stack for the received packet.
Protocol Stacks	3. Use the <i>LOOKAHEAD</i> structure to determine whether to keep or reject the packet. If the protocol stack requires the <i>ECBProtocolID</i> , <i>BoardNumber</i> , and <i>ImmediateAddress</i> fields to be filled in, the protocol stack should get these from the lookahead structure. If an ECB is returned from this function, the MLID will call the ESR at a later time, signaling that the packet data has been transferred to the described receive buffers either successfully or with an error. The receive handler should add the ECB to the protocol stack's internal work-to-do queue for processing at non-privileged time. The receive handler routine should not poll for transmissions or receptions. If the PreScan Receive stack or the Default Chain stack cannot understand the MAC header, they should leave the MAC header unchanged and pass it on by returning <i>ODISTAT_OUT_OF_RESOURCES</i> .
MLID	4. Disperses the receive packet data into the receive buffers supplied by the protocol stack. The MLID sets the following fields in the ECB (see Appendix A): <i>ECB_PreviousLink</i> <i>ECB_Status</i> <i>ECB_DriverWorkSpace</i> <i>ECB_DataLength</i>
	5. Places the ECB onto the LSL hold event queue, or processes the ECB using the CLSL_FastHoldEvent routine and step 6 is skipped.
	6. Invokes the LSL's service events routine after the MLID has finished servicing the network adapter.
LSL's Service Events Routine	7. Calls each of the previously queued ECB's event service routine (ESR). (The protocol stack sets the address of the ESR before the protocol stack returned an ECB to the MLID.)
	8. Transfers ownership of the ECB and its associated data buffers back to the protocol stack when the ECB's ESR is called.

Protocol Receive Complete Handler for Bound Stacks

The LSL invokes this event service routine after the MLID has dispersed the receive packet data (with or without error) into the provided ECB data buffers and has given the ECB to the LSL for processing via **CLSL _HoldEvent/CLSL_ServiceEvents** or **CLSL_FastHoldEvent**.

Syntax

```
#include <ODI.H>

void (ECB_ESR)(ECB*);
```

Input Parameters

ECB
Pointer to an ECB.

Output Parameters

None.

Return Values

None.

Remarks

When this function is called, the LSL transfers ownership of the ECB and its associated data buffers back to the protocol stack.

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

- ECB_Previous Link
- ECB_Status
- ECB_DriverWorkspace
- ECB_DataLength

See the ECB field descriptions in Appendix A, "Event Control Blocks (ECBs)".

Do not poll for transmissions or receptions when this function is called.

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

Table 4-3
Receive Complete Handler Event Sequence

Actor/Agent	Action
Protocol Stacks	<div>1. Increment receive statistics counters.</div> <div>2. Restore user's ESR to the <i>ECB_ESR</i> field of the ECB structure. (The original user's ESR should have been stored in the <i>ECB_ProtocolWorkspace</i> field.)</div> <div>3. Set the <i>ECB_Status</i> field of the ECB structure to <i>ODISTAT_SUCCESSFUL</i>.</div> <div>4. If the stack accepted a prefilled ECB, return the prefilled ECB to the LSL by calling <i>CLSL_ReturnECB</i>.</div>



Protocol Receive Handler for Prescan and Default Stacks

The receive handler for prescan and default protocol stacks is similar to the receive handler for bound protocol stacks. After the ECB has been filled (or prefilled), the protocol stack is responsible for returning its ECB. The stack is also responsible for calling **CLSL_ReSubmitPreScanRx** with prescan stacks, or for calling **CLSL_ReSubmitDefault** with default stacks, for continued processing of the ECB by protocol stacks further down the chain.

Syntax

```
#include <odi.h>

ODISTAT ( *StackRxChainHandler )
( LOOKAHEAD *LkAhead,
  struct _PS_CHAINED_RX_NODE_
    *StackChainNode );
```

Input Parameters

LkAhead

Pointer to the received LOOKAHEAD structure.

StackChainNode

Pointer to the stack chain's node that was passed when the stack chain registered for the board that generated this receive packet.

Return Values

ODISTAT_SUCCESSFUL	<p>The protocol stack has returned a pointer to a receive ECB that is to be filled with the packet in the LOOKAHEAD structure's <i>LkAhd_ReturnedECB</i> field.</p> <p>If the protocol stack was able to get everything it needed from the look ahead, the <i>LkAhd_ReturnedECB</i> field will be set to NULL.</p> <p>If the <i>LKAhd_PreFilledECB</i> field contains a pointer to an ECB, the LSL will perform an ECB to ECB copy and queue the protocol stack ECB on its hold event queue.</p>
ODISTAT_SUCCESS_TAKEN	<p>The protocol stack has accepted the packet and has taken the prefilled LSL ECB associated with the LOOKAHEAD structure's <i>LkAhd_PreFilledECB</i> field.</p> <p>If <i>ODISTAT_SUCCESS_TAKEN</i> is returned, the LSL assumes that the protocol stack has consumed the ECB. The protocol stack ESR will not be called, and the protocol stack is responsible for calling <i>CLSL_ReturnECB</i>.</p>
ODISTAT_OUT_OF_RESOURCES	<p>Reports an error condition—for example, the LSL was unable to obtain an ECB for this packet. The LSL routes the LOOKAHEAD structure to the next prescan stack, the appropriate bound stack, or the next default stack in the chain.</p>

Remarks

A chained protocol stack must specify in its filter mask the type of packets it wants to receive when it registers for the board. A stack can receive packets after binding/registering with a board and can modify its filter to allow it to specify the type of packets that it wants the LSL to pass to it.

The LSL is responsible for calling the next prescan receive chained stack. The adding and calling of protocol stacks to the chain is at process time only and is in load order (first to load is first in chain, second to load is second in chain, etc.), unless an order position is specified (“must be very first” or “must be very last”).

If the protocol stack rejects the packet, the LSL routes the LOOKAHEAD structure to the next stack in the chain.

If the protocol stack wishes to accept the packet (by returning `ODISTAT_SUCCESSFUL`), the stack will need to make a copy of the LOOKAHEAD structure that has been passed to it. This is because sometime after or during the execution of the stack’s receive complete handler, the stack will need to behave similarly to the LSL (see “Protocol Receive Complete Handler for Prescan and Default Stacks” later in this chapter). The stack will have to provide lookahead information to the next stack in the chain (`CLSL_ReSubmitDefault` or `CLSL_ReSubmitPreScanRx` process).

The ECB that is returned should have the *ECB_NextLink* and *ECB_PreviousLink* fields set to NULL. *ECB_ESR* should be set to the stack’s receive complete handler.

Note



Prescan and default stacks have no Stack IDs, so assigning a value to the *ECB_StackID* field is optional.

Ownership of the LOOKAHEAD structure and its associated data buffer is passed to the protocol stack when the LSL calls the protocol’s receive handler. The protocol’s receive handler is called from the context of privileged time—for example, a hardware interrupt, deferred procedure call (DPC) in Windows NT, or at process time (due to the resubmit process). The MLID will not have finished servicing the network adapter but is waiting for an ECB from a protocol stack to fill or a response to discard the received packet. Protocol stacks must decide if they want the packet and provide an ECB to fill or must respond that the LSL must route the LOOKAHEAD structure.

Also, if the stack wishes to accept the packet, the stack should set the LOOKAHEAD structure’s *LkAhd_FrameDataBytesWanted* field to be the

4-24 ODI Specification: Protocol Stacks and MLIDs (C Language)

number of bytes wanted by the stack. The data pointed to by the *LkAhd_MediaHeaderPtr* pointer should be copied into a buffer so that it can be used by stacks further down the chain. Finally, the LOOKAHEAD structure's *LkAhd_ReturnedECB* field should be set to the address of the ECB that describes the receive buffers.

After the MLID finishes filling a protocol stack ECB, the protocol stack's ECB ESR routine ((*ECB_ESR)(ECB*)) is called. The MLID is finished servicing the network adapter and is ready to restore the processor state and exit from its event handler, (it's interrupt handler). Protocol stacks can queue the event and process it from a handler that is running at process time, or process the event as a run-to-completion event aware that this can degrade performance.

If the LOOKAHEAD structure *LKAhd_PreFilledECB* field is not NULL, and the protocol stack does not handle the prefilled ECB immediately, the protocol stack places the address of its receive handler ESR in the ECB's *ESR_ESR* field, then calls **CLSL_HoldEvent**, which queues the ECB for later processing. The stack then returns *ODISTAT_SUCCESS_TAKEN*.

If the protocol stack processes the received ECB by queuing it and servicing the ECB at process time, it can resubmit the ECB for further processing by other prescan, bound, or default protocol stacks by the appropriate resubmit function. Packet reception ordering must be maintained. MLID control routines must **not** be invoked from this routine because they can only be called at process time. However, the protocol stack can freely make requests to the LSL (such as *CLSL_GetSizedECB*). If the protocol stack consumes an ECB supplied by the LSL, after it has finished with it, it returns the ECB to the LSL using the *CLSL_ReturnECB* support function.

If the protocol stack has only minor activity to perform on an ECB that is not time intensive, it could, as an optimization, perform its functions, then pass the ECB on rather than queuing it and resubmitting it for processing later. This is usually done in the prescan/default receive complete handler.

Protocol Stack Packet Reception 4-25

Tasks for Accepting a Packet

Some tasks to perform when accepting a packet might be as follows:

1. Make a copy of the original LOOKAHEAD buffer, which is used when the **CLSL_ResubmitPreScanRx/ CLSL_ResubmitDefault** pair is called. The LSL can pass state information in this field if **ODISTAT_OUT_OF_RESOURCES** is returned. The **LKAhd_Prefilled** ECB must equal **NULL** if the stack is not dealing with a prefilled ECB.

2. Verify that the packet is wanted. The **MLID** will still be servicing the network adapter and will be waiting either for an ECB from the protocol stack to fill or a response to discard the received packet.

If the protocol stack wants the packet, it must provide an ECB to fill.

If the protocol stack wants to reject the packet, it must return **ODISTAT_OUT_OF_RESOURCES**, and the LSL will route the LOOKAHEAD structure to the next stack in the chain.

Ownership of the LOOKAHEAD structure and its associated data buffer is passed to the protocol stack when the LSL calls the protocol's receive handler. The protocol's receive handler is called from the context of privileged time, such as a hardware interrupt, a deferred procedure call (DPC) in Windows NT, or a resubmit process at process time.

If the packet is wanted, get a pointer to an ECB from the LSL by calling **CLSL_GetSizedECB**. If the pointer to the ECB is **NULL**, return **ODISTAT_OUT_OF_RESOURCES**.

3. Copy the lookahead media header information into a buffer so that it can be used later by the stack's receive complete ESR.
4. If the stack is dealing with a prefilled ECB, set the *ECB_ProtocolWorkspace* field to point to the chained stack ECB returned by **CLSL_GetSizedECB**. Since copying the prefilled ECB to the stack ECB is time intensive, it must be done at process time by adding the prefilled ECB to the work queue using **CLSL_FastHoldEvent**. Then, return **ODISTAT_SUCCESS_TAKEN**.



Once **ODISTAT_SUCCESS_TAKEN** has been returned to the LSL, the lookahead indication is no longer valid. Therefore, make sure that the *ECB_ESR* field in the chained stack contains the proper pointer to the stack's receive complete ESR.

5. Set the LOOKAHEAD structure's *LkAhd_FrameDataBytesWanted* field to be equal to the number of bytes wanted by the stack.
6. If required by the stack, set the ECB's *ECB_BoardNumber*, *ECB_ImmediateAddress*, *ECB_ProtocolID*, and *ECB_DataLength* fields to either the values found in the LOOKAHEAD structure or the values needed by the stack.
7. Verify that the packet has been fully received. If the packet has not totally arrived, hold off on doing any processing of the packet until the stack's receive complete handler is called.
8. Set the ECB's *ECB_NextLink* and *ECB_PreviousLink* fields to equal NULL. Set the *ECB_Status* field to equal *ODISTAT_SUCCESSFUL*. Set the *ECB_ESR* field equal to the stack's receive complete handler function. Verify that the *ECB_FragmentCount* field is properly set.
9. Set the *LkAhd_FrameDataStartCopyOffset* field in the LOOKAHEAD structure to the offset value required for the stack to process the packet. Then, assign the pointer to the ECB to the LOOKAHEAD structure's *LkAhd_ReturnedECB* field and return *ODISTAT_SUCCESSFUL*.
10. After the MLID finishes filling the protocol stack's ECB, the protocol stack's ECB ESR routine is called. The MLID is now finished servicing the network adapter and is ready to restore the processor state and exit from its event handler (its interrupt handler).

Protocol Stack Packet Reception **4-27**

Prescan and Default Stacks Receive Complete Handler Event Sequence

The following table illustrates the event sequence of the receive handler.

Table 4-4
Receive Complete Handler Event Sequence

Actor/Agent	
MLID	1. Provides a LOOKAHEAD structure to the LSL using CLSL_GetStackECB .
LSL	2. The LSL determines the correct protocol stack for the received packet.
Protocol Stack	3. Uses the LOOKAHEAD structure to determine whether to keep or reject a packet.
MLID	4. Disperses the receive packet data into the receive buffers supplied by the protocol stack.
	5. Places the ECB onto the LSL hold event queue using CLSL_HoldEvent , or processes the ECB using the CLSL_FastHoldEvent routine and step 6 is skipped.
	6. Invokes the LSL's service events routine after the MLID has finished servicing the network adapter.
LSL's Service Events Routine	7. Calls each of the previously queued ECB's event service routine (ESR). (The protocol stack sets the address of the ESR before the protocol stack returned an ECB to the MLID.)
	8. Transfers ownership of the ECB and its associated data buffers back to the protocol stack when the ECB's ESR ((*ECB_ESR)(ECB*)) is called.

Protocol Receive Complete Handler for Prescan and Default Stacks

The LSL invokes this event service routine function after the MLID has dispersed the receive packet's data (either with or without error) into the previously provided ECB's data buffers and has placed the ECB on the LSL's holding queue or called directly via *CLSL_FastHoldEvent*.

Syntax

```
#include <odi.h>

void (*ECB_ESR)(ECB *);
```

Input Parameters

ECB
Pointer to an ECB.

Output Parameters

None.

Return Values

None.



Remarks

When this function is called, the LSL transfers ownership of the ECB and its associated data buffers back to the protocol stack.

Steps Performed by a Prescan and Default Receive Complete Handler

The receive complete handler for a prescan stack or a default stack must perform the following steps:

1. Increment the receive statistics counters.
2. If the stack contains a header, adjust the *LOOKAHEAD* structure's *LkAhd_DataLookAheadPtr* pointer to point to the data that follows the header.

Next, adjust the *LOOKAHEAD* structure's *LkAhd_FrameDataSize* field to be equal to the frame data size minus the size of the stack's header.
3. If the stack has a header and received a prefilled ECB, adjust the ECB's *ECB_Fragment[n].FragmentAddress* field so that the header is skipped and only the data that belongs to the next stack is present.

Next, adjust the ECB's *ECB_Fragment[n].FragmentLength* field to be equal to the length of the fragment minus the length of the stack's header.

Finally, adjust the ECB's *ECB_DataLength* field to be equal to the data length minus the length of the stack's header.

4. If the stack does not consume the packet, it notifies the next stack in the chain that a packet has arrived. Call **CLSL_ReSubmitPreScanRx/CLSL_ReSubmitDefault**.
 - a. If *ODISTAT_SUCCESSFUL* is returned, take the ECB found in the *LOOKAHEAD* structure's *LkAhd_ReturnedECB* field and copy into it *LkAhd_FrameDataBytesWanted* bytes of the data pointed to by the *LOOKAHEAD* structure's *LkAhd_DataLookAheadPtr* pointer. Then, using the *LOOKAHEAD* structure's *LkAhd_ReturnedECB* field as a parameter, issue a call to **CLSL_FastHoldEvent**.
 - b. If *ODISTAT_SUCCESS_TAKEN* is returned, the next protocol stack in the chain wants the packet and has taken the prefilled LSL ECB associated with the *LOOKAHEAD* structure.

c. If *ODISTAT_OUT_OF_RESOURCES* is returned, the Link Support Layer was unable to find another stack that wanted the packet.

5. The stack that returns *ODISTAT_SUCCESS_TAKEN* is given the ownership and responsibility of the ECB and its associated data buffers. If the next stack in the chain does not return *ODISTAT_SUCCESS_TAKEN*, it is the responsibility of the current stack to call *CLSL_ReturnECB*.

After this function has been called, the following ECB fields and values have been set by the MLID:

- *ECB_PreviousLink*
- *ECB_Status*
- *ECB_DriverWorkspace*
- *ECB_DataLength*

For a description of these fields, see the ECB field descriptions in Appendix A, "Event Control Blocks (ECBs)".

Do not poll for transmits or receives when this function is called.

Table 4-5
Receive Complete Handler Event Sequence

Actor/Agent	Action
Protocol Stacks	<ol style="list-style-type: none">1. Increments the receive statistics counters.2. Notify the next stack in the chain that a packet has arrived.3. If the current stack is responsible for returning the ECB to the LSL, call CLSL_ReturnECB.

4-32 ODI Specification: Protocol Stacks and MLIDs (C Language)



Chapter Overview

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

Note



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

Transmit Routine Events

The events listed in Table 1.1 must occur during a protocol stack transmit routine.

Table 5-1
Protocol Stack Transmit Routine

Actor/Agent	Action
Protocol stack	1. Hands the ECB to the LSL for transmission.
LSL	2. If the ECB contains a valid <i>BoardNumber</i> , the underlying MLID transmit handler is called with a pointer to the ECB, or the LSL send packet routine returns the <i>ODISTAT_ITEM_NOT_PRESENT</i> status to the protocol stack and returns ownership of the ECB back to the protocol stack.
	3. Passes ownership of the ECB and its associated packet data buffers to the MLID.

Table 5-1
Protocol Stack Transmit Routine

Actor/Agent	Action
MLID	4. Transmits the packet.
	5. Shows the transmitted packet to any transmit monitor, if one is registered, including the media headers generated by the MLID, regardless of whether the packet transmission was completed successfully or with an error.
	6. Passes ownership of the ECB and its associated packet data buffers to the LSL, regardless of whether the packet transmission was completed successfully or with an error.
LSL	7. Calls the event service routine specified in the ECB.
Note: The ECB and its associated data buffers must not be modified until ownership is returned to the protocol stack that originated the transmit request.	

Prescan Transmit Protocol Stack Method

The ODI specification defines the prescan transmit protocol stack method for protocol stack packet transmission.

Prescan transmit protocol stacks are given the transmit ECB before it is given to the MLID by the LSL. The protocol stack can consume select packets and allow others to be passed to the appropriate MLID. The prescan transmit protocol stack can alter the data to be transmitted—for example, compressing a packet, but it must treat the original ECB and its data as read-only. In other words, when the original ECB is returned to the originating protocol stack, it must be in its original state. The prescan transmit protocol stack can also discard or pass the data to the next chained prescan transmit protocol stack. Multiple chained protocol stacks for each logical board can use the prescan transmit method.

Packet Transmission

A protocol stack can usually transmit packets at any time.



A protocol stack must not poll for a transmit complete or a receive packet inside an event service routine (ESR). Polling for a transmit complete or a receive packet inside of an ESR without following the above rule can create a dead-lock. A protocol stack must also not issue transmit requests inside its packet

5-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

lookahead receive handler routine (see *Chapter 5: Protocol Stack Packet Reception*).

Supporting Multiple Outstanding Transmit Requests

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

When the protocol stack must transmit bursts of packets, it achieves its best performance by passing multiple transmit requests to an underlying MLID. In theory, an MLID can handle any number of outstanding transmit requests.

For example, on Ethernet, three outstanding requests are adequate, and an MLID that handles more than three active transmit requests generally does not have higher throughput. (Three active transmit requests saturate most Ethernet LAN adapters). In other words, a protocol stack should be able to have at least three transmits outstanding on a particular board.

Note: The number of active requests that an MLID can handle depends on the adapter, the media, and the bus that the adapter was designed for.

Transmitting the Packet

The protocol stack must provide data buffers and an ECB describing the data to be sent to send a packet. The protocol stack can specify from one to sixteen data buffers per transmit request. The underlying MLID will then combine the buffers to form a single data packet.

Priority Sends

The MLID sets the *MLIDCFG_PrioritySup* field in the MLID configuration table to indicate the number of priority levels available. The MLID indicates that priority support is active by setting or clearing the MF_PRIORITY_BIT in the *MLIDCFG_Flags* field of the MLID configuration table. The MLID can set or reset the MF_PRIORITY_BIT as the MLID changes from the Priority Queue Support Enabled state to the Disabled state.

The protocol sets the *ECB_StackID* field to a value greater than or equal to 0xFFFF0. The values have the following meanings:

0xFFFF	Raw send packets. No priority.
0xFFFFE-0xFFFF8	Raw send packets. Priority level 1-7.
0xFFFF7	Non-raw send packets. No priority.
0xFFFF6-0xFFFF0	Non-raw send packets. Priority level 1-7.

Priority levels are defined as follows:

```
0 = no priority
7 = highest priority
```

To extract the priority level, NEG (1s complement) the *ECB_StackID* field, and AND it with 0x07. The result will be a number from 1 to 7.

The MLID will normally send the packet directly. If the MLID is busy and the ECB is a priority transmit ECB, the MLID will either queue it in a priority queue for transmission as soon as possible, or transmit the packet through a priority channel.

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

Event Control Blocks

An event control block (ECB) is a general purpose request control block used for transmit and receive events in the ODI specification. The *ECB_ProtocolWorkspace* field can be used for any purpose by the protocol stack, because the *ECB_ProtocolWorkspace* field is not modified by the LSL or the MLIDs. (See *Appendix A: Event Control Blocks (ECBs)* for a more detailed discussion of ECBs.)

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

- ECB_ESR

5-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

- ECB_StackID
- ECB_BoardNumber
- ECB_ProtocolID
- ECB_ImmediateAddress
- ECB_DataLength
- ECB_FragmentCount
- Fragment descriptors

These fields are treated as read-only by the LSL and MLIDs. Therefore, you do not need to re-initialize each field after a transmit operation unless that field's value needs to be changed.

ECB_ESR Field

The *ECB_ESR* field contains a pointer to a routine the LSL calls when the underlying MLID has finished with the ECB and its data buffers. (See the “Transmit Complete” section later in this chapter.)

ECB_StackID Field

The *ECB_StackID* field is initialized with the protocol's assigned Stack ID (see **CLSL_GetStackIDFromName**). Raw sends and priority sends are indicated in this field.

Raw Sends

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

Because a raw send is an optional capability, some MLIDs do not support it. To determine if a particular board supports raw sends, the protocol stack checks *MM_RAW_SENDS_BIT* in the *MLIDCFG_ModeFlags* field. If this bit is set,

Protocol Stack Packet Transmission 5-5

the MLID supports raw sends (see **GetMLIDConfiguration** for more information).

A protocol stack signals a raw send to the MLID by placing 0xFFFFy (see *ECB_StackID* field in *Appendix A: Event Control Blocks (ECBs)* for more information), instead of its Stack ID (SID) in the *ECB_StackID* field. The underlying MLID checks this field for 0xFFFFy; where y is a value from 0 to F (see page 6-4). If this value is present in the field, the MLID skips over the code used to build the low-level header. The first fragment of the ECB must then contain all of the low-level header information.

The first data fragment must contain the complete MAC header, including the source address for the media in use. However, in some cases this address is not used; some adapters automatically insert the source node address in the low-level header. If an adapter supports raw sends, it should not overwrite the source address provided in the MAC header with its node address.

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

ECB_BoardNumber Field

The *ECB_BoardNumber* field specifies the logical board used to transmit the packet. The board number specifies the physical adapter and the low-level frame format (frame type) used.

ECB_ProtocolID Field

The *ECB_ProtocolID* field specifies the Protocol ID (PID) value embedded into the frame header. This value stamps the packet as a particular protocol type (for example, IPX, TCP/IP, etc.). (See **CLSL_GetPIDFromStackIDBoard** for more information.)

For example, the 802.2 frame *ECB_ProtocolID* field contains the Destination Service Access Point (DSAP). The Source SAP (SSAP) is set equal to the Destination SAP (DSAP or Protocol ID) when the MLID builds the frame header. The MLID also sets the 802.2 control byte equal to 0x03 (UI).

In order to allow a protocol stack to specify the complete 802.2 header (for example, DSAP, SSAP, Control 0, Control 1), MLIDs that support the 802.2 frame allow a special flag in the transmit *ECB_ProtocolID* field. When this flag

5-6 ODI Specification: Protocol Stacks and MLIDs (C Language)

is present, the MLID uses the specified 802.2 header instead of setting SSAP equal to DSAP and *Control* equal to 0x03 (the usual method).

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

- | | | |
|-------------|---|---|
| Byte 0 | x | This byte is normally 0. However, if a nonzero number is specified, the MLID will look for the explicit header information. x = a zero-based number of bytes in the explicit number (for example, 0x00 signifies DSAP, 0x02 signifies DSAP, SSAP, or Control 0 802.2 Type I header, and 0x03 signifies DSAP, SSAP, Control 0, Control 1 802.2 Type II header). A value of <i>a</i> , where $0x40 < a < 0x7F$, indicates that the values in this field contain a management ID—for example, <i>HUBMGR</i> . |
| Bytes 1 - 5 | | These bytes are set by the protocol stack for an explicit 802.2 header (see <i>ODI Specification: Frame Types and Protocol IDs</i>). Unused bytes are set to 0. |

ECB_ImmediateAddress Field

The *ECB_ImmediateAddress* field contains the destination address that specifies the node on the local network where the packet is sent. This can be a direct, multicast, or broadcast address. If your protocol stack must receive its own sends, it must emulate loopback capabilities. (If the MLID is using canonical addressing, the destination address must also be in canonical form.)

Note



The address 0xFF FF FF FF FF FF always indicates a broadcast packet. (All adapters on the physical network will receive the packet.)

ECB_DataLength Field

The *ECB_DataLength* field holds the total length of the packet in bytes. This is the length of the data portion of the packet.

ECB_FragmentCount Field

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

Fragment Descriptors

Each fragment descriptor contains the location and length of a contiguous section of RAM memory. The protocol stack can specify a maximum of sixteen fragment descriptors. The MLID combines the fragments together to form one contiguous packet.

Note



The length field of a fragment descriptor can be 0.

A frame containing only an 802.2 Type II header can be transmitted by setting the length fields of the fragment descriptors for the ECB containing the transmit information to 0. The *ECB_FragmentCount* field must be equal to at least 1. The *ECB_ProtocolID* field contains the entire explicit 802.2 Type II header.

Transmit Handler

The interface to the Transmit Handler is defined by each protocol stack.

Table 6-2 describes the events that occur to transmit a packet transmission.

Table 5-2
Transmit Handler Event Sequence.

Actor/Agent	Action
Protocol stack	1. Gives the ECB to the MLID by calling CLSL_SendPacket .
LSL	2. Determines which MLID to give the packet to via the board number.
MLID	3. Transmits the ECB.
	4. Presents the transmitted packet to any transmit monitor, if one is registered.
	5. Returns the ECB to the LSL via CLSL_SendComplete .
LSL	6. Places the ECB into a temporary event queue.
MLID	7. Calls the LSL's <i>CLSL_ServiceEvents</i> routine after the MLID has finished servicing the hardware.
LSL's Service	8. Removes each ECB from the queue in turn and calls the ESR defined in the <i>ECB_ESR</i> field.
Events Routine	9. The ESR is the protocol stack's transmit complete handler. The MLID can invoke the protocol stack's transmit complete handler before the call to CLSL_SendPacket returns. This is called a lying send.



If **CLSL_FastSendComplete** is called, the LSL can call the ESR in the ECB directly, depending on the platform. However, this does not affect protocol stack or MLID operations.

Protocol Transmit Handler for Prescan Stacks

Syntax

```
#include <odi.h>

ODISTAT ( *StackTxChainHandler )
( ECB *TransmitECB,
  struct _PS_CHAINED_TX_NODE_
    *StackChainNode );
```

Input Parameters

- TransmitECB*
Pointer to the transmit ECB.
- StackChainNode*
Pointer to the stack chain's node passed when the stack chain registered with the board that will transmit this packet.

Return Values

- ODISTAT_SUCCESSFUL The protocol stack wants the packet and is responsible for returning the ECB to the LSL or calling the LSL later (asynchronously) for further processing of the ECB.
- ODISTAT_FAIL The protocol stack does not want the packet and returns a pointer to an ECB. The LSL routes the ECB structure to the next stack in the chain. The pointer to the ECB is unchanged.

Remarks

The LSL is responsible for calling the next prescan transmit chained protocol stack. The adding to the chain and calling of protocol stacks is in load order (first to load is first in chain, second to load is second in chain, etc.) unless an order position is specified— "must be very first" or "must be very last". Register/DeRegister Protocol Stack can be made at process time only.

On transmission the prescan transmit stack is called with a pointer to a transmit ECB.

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

Protocol stacks can queue an event and process it from a handler that is running at process time, or process the event as a run-to-completion event—be aware that this can degrade performance.

The network hardware can be fully functional at this point; hence, packet transmission ordering must be maintained. MLID control routines must **not** be invoked from this routine, because they can only be called at process time. However, the protocol stack can freely make requests to the LSL (such as **CLSL_GetSizedECB**) to obtain another ECB buffer. If the protocol stack consumes the ECB after it has finished with it, it returns the ECB to the LSL using the **CLSL_ReturnECB** support function.

If the protocol stack processes the transmit ECB by queuing it and servicing the ECB at process time, it can resubmit the ECB for further processing by other transmit prescan protocol stacks by the appropriate resubmit function (see **CLSL_ReSubmitPreScanRx**, **CLSL_ReSubmitPreScanTx**, and **CLSL_ReSubmitDefault** for more information). If the protocol stack has only minor activity to perform on an ECB, which is not time intensive, it can, as an optimization, perform its functions even at privileged (interrupt or deferred procedure call in Windows NT) time and then pass on the ECB rather than queue it and resubmit the ECB for processing later.

Transmitting prescan protocol stacks must treat ECBs that have data to be modified as read-only. The protocol stack must make a copy of the ECB and process the copy. For example, the protocol stack would install its Event Service Routine (ESR) in the copy of the ECB. The reason for this is the originating protocol stack, for instance a bound stack, can manipulate data in the original ECB when its ESR is called. If the prescan stack manipulates the data (for example, compression), the data will be incomprehensible to the original stack. Hence, when the prescan stack's Transmit Complete ESR is called, it in turn calls the Transmit Complete ESR in the original ECB with a pointer to the original ECB.

Note



Data transmitted by prescan stacks are still limited by the transmitting MLID configuration table's *MLIDCFG_WorstDataSize* field. Also calling the LSL function **CLSL_SendPacket** from a transmit prescan stack can cause the prescan stack's protocol transmit handler to be called from itself.

Protocol Stack Packet Transmission 5-11

The Prescan Transmit Handler should add the ECB to the protocol stack's internal work-to-do queue for processing at non-privileged time.

If the PreScan Receive stack or the Default Chain stack cannot understand the MAC header, they should leave the MAC header unchanged and pass it on.

Do not use this function to poll for transmissions or receptions.

Protocol Transmit Complete Handler

Called when a previous transmit request has completed successfully or with an error.

Syntax

```
#include <odi.h>

void (*ECB_ESR)(ECB *);
```

Input Parameters

ECB
Pointer to the completed ECB.

Output Parameters

None.

Return Values

None.

Remarks

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

This routine must complete quickly because it is usually invoked from a board service routine during privileged context time—for example, interrupt handler, deferred procedure call (DPC) in Windows NT. Transmit requests can be issued from this routine, but a protocol stack must not poll for a transmit to complete.

Do not poll when calling this function.

The *ECB_Status* field is set to one or more of the following.



Note

The *ODISTAT* type is cast to UINT16 for the *ECB_Status* field.

ODISTAT_SUCCESSFUL	The MLID determined that the transmit was successful. Because the transmit was connectionless, this completion code does not mean that the destination received the packet.
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).
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.
ODISTAT_CANCELED	The ECB is being returned without being transmitted. This usually occurs if the ECB was held in an MLID's queues, then the MLID clears its queues due to a shut down request.

The following tables describe the events that occur during the transmit complete handler.

Table 5-3

Transmit Complete Handler Event Sequence for Bound Stacks

Actor/Agent	Action
Protocol stack	<ol style="list-style-type: none"> 1. Increment counters. 2. Restore the original ECBs ESR to the ECB_ESR field. 3. Call ECB_ESR using the ECB from the transmit complete handler.

Table 5-4

Transmit Complete Handler Event Sequence for Prescan Stacks

Actor/Agent	Action
Protocol stack	<ol style="list-style-type: none"> 1. Retrieve the pointer to the original Tx ECB.

5-14 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 5-4
Transmit Complete Handler Event Sequence for Prescan Stacks

Actor/Agent	Action
	2. Increment counters.
	3. Set the original Tx ECB_Status to the current ECB_Status.
	4. Call the original Tx ECB ESR using the original Tx ECB as the parameter.
	5. Call CLSLReturnECB to return the current ECB.

5-16 ODI Specification: Protocol Stacks and MLIDs (C Language)

Chapter Overview

The ODI specification requires protocol stacks to provide control functions to the LSL for use by applications and other protocol stacks. When a protocol stack registers with the LSL, the LSL passes a pointer to the protocol stack's information block (INFO_BLOCK) for control functions. Applications and other protocol stacks use these pointers as entry points to get configuration information and statistics about specific protocols (see **CLSL_GetProtocolControlEntry**).


All reserved and unsupported control functions must have pointers in the information block (INFO_BLOCK), which, when called, will return ODISTAT_BAD_COMMAND as the completion code.

The following functions are currently defined for these entry points:

```
Bind
GetBoundNetworkInfo
GetProtocolStackConfiguration
GetProtocolStackStatistics
GetProtocolStringForBoard
MLIDDeRegistered
PromiscuousStatus
ProtocolManagement
UnBind
```

The functions above are accessed through the information block using indexes. The location of the various protocol stack control functions in the information block are as follows:

Index	Function
0	GetProtocolStackConfiguration
1	GetProtocolStackStatistics
2	Bind
3	UnBind
4	MLIDDeRegistered
5	PromiscuousState
6	Reserved
7	GetProtocolStringForBoard
8	ProtocolManagement
9	GetBoundNetworkInfo

Note  Access to the Protocol Stack APIs, independent of the link method (dynamic or static), in the information block can be accomplished by using the macro definitions in ODI.H. The macros are listed below. The *infoBlock* parameter is returned by **CLSL_GetProtocolControlEntry**. Refer to the API definitions for details on the rest of the parameters.

```
PStkCntl_GetConfig(infoBlock, stackIdentifier)
PStkCntl_GetStats(infoBlock, stackIdentifier)
PStkCntl_Bind(infoBlock, boardNumber, userParmString, stackIdentifier)
PStkCntl_MLIDDeReg(infoBlock, boardNumber, stackIdentifier)
PStkCntl_Unbind(infoBlock, boardNumber, userParmString, stackIdentifier )
PStkCntl_PromiscState(infoBlock, boardNumber, promiscuousMask,
                      stackIdentifier)
PStkCntl_GetProtocolString(infoBlock, boardNumber, printString,
                          stackIdentifier)
PStkCntl_ProtManage(infoBlock, ManagementECB, stackIdentifier)
PStkCntl_GetBoundNetInfo(infoBlock, boardNumber, networkAddress,
                        stackIdentifier)
```

Bind

Index 2

Provides a consistent method of binding a protocol stack with an MLID.

Syntax

```
#include <odi.h>

ODISTAT Bind (
    UINT32 BoardNumber,
    MEON_STRING *UserParmString,
    void *StackIdentifier );
```

Input Parameters

BoardNumber

The board number for the protocol stack to bind to.

UserParmString

Pointer to an optional user specified MEON parameter string that is implementation dependent; NULL if unused.

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—

LSL_ConfigTable.LMaxNumberOfStacks), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully bound to an MLID.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified <i>StackIdentifier</i> does not exist.
ODISTAT_DUPLICATE_ENTRY	The protocol stack is already bound to this MLID.
ODISTAT_FAIL	The protocol stack failed to bind to the specified MLID.
ODISTAT_ITEM_NOT_PRESENT	No Protocol ID (PID) has been registered for use by this protocol stack with the specified board's frame type. In other words, you must register a PID by calling <code>CLSL_AddProtocolID</code> for the board's frame type that is used with this stack.
ODISTAT_OUT_OF_RESOURCES	The call could not allocate enough memory.

Remarks

All protocol stacks must support this function.

The protocol stack is expected to issue the **CLSL_BindStack** or **CLSL_BindProtocolToBoard** call to the LSL as well as perform any other maintenance commands required to bind to an MLID.

This function is invoked when the user issues the “bind” command to a protocol stack to bind to a logical board, for example:

```
bind <MLID Short Name> [ Board Number <Protocol ID> ]
```



Note The method of specifying the binding of a protocol stack to a logical board is entirely up to the protocol stack developer. The line specifying the binding information can be passed to the protocol stack or some entity to parse and the resultant binding information as to the stack and which logical board it is bound to is passed in the **CLSL_BindStack** or **CLSL_BindProtocolToBoard** call.

The first thing a protocol stack does is verify whether a specified board exists and whether a Protocol ID (PID) is available for the protocol that uses that particular board. The protocol stack can verify that a specified board exists by

calling the **CLSL_GetMLIDControlEntry** function. If the board is valid, the protocol stack determines whether a PID exists for the protocol on that particular board by calling **CLSL_GetPIDFromStackIDBoard**. If a PID is not present for that protocol, the protocol stack adds a PID to use or stops the initialization procedure.

GetBoundNetworkInfo

Index 9

Gets the bound network address for a board / protocol stack combination.

Syntax

```
#include <odi.h>

ODISTAT GetBoundNetworkInfo (
    UINT32          BoardNumber,
    NETWORK_ADDRESS_INFO *networkAddress
    void            *StackIdentifier);
```

Input Parameters

boardNumber

The board number the protocol stack is to return the network address for.

networkAddress

Pointer to a buffer where the bound network address for the protocol is returned.

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—

LSL_ConfigTable.LMaxNumberOfStacks), or the pointer is a pointer to a stack chain node.

Output Parameters

networkAddress

NULL is placed at the start of the buffer if no address is returned.

Return Values

ODISTAT_SUCCESSFUL	The network address was successfully returned. Note: ODISTAT__SUCCESSFUL is returned even if the <i>addressType</i> and <i>size</i> fields are zero, and the <i>address</i> field is NULL; this implies that there is no network address for the board and protocol combination.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified <i>StackIdentifier</i> .

Remarks

The protocol stack will fill in the NETOWRK_ADDRESS_INFO structure *addressType* field with it's assigned transport address type, the size field with the length of the address, and the address field with the bound network address. IPX returns all 12 bytes, network:node:socket. IP returns 4 bytes, network address only (no socket).

The following Transport Address types have been assigned:

IPX	1
IP	2
DDP	3
NETBEUI	4

The networkAddress structure is defined in ODI.H as follows:

```
typedef struct_NETWORK_ADDRESS_INFO_  
{  
    UINT32      addressType;  
    UINT32      size;  
    UINT8      address[32];  
}NETWORK_ADDRESS_INFO;
```

GetProtocolStackConfiguration

Index 0

Returns a pointer to the protocol stack's configuration table.

Syntax

```
#include <odi.h>

PS_CONFIG_TABLE *GetProtocolStackConfiguration (
    void *StackIdentifier );
```

Input Parameters

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—**LSL_ConfigTable.LMaxNumberOfStacks**), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

ConfigTable Returns a pointer to the protocol stack configuration table.

NULL *StackIdentifier* is invalid

Remarks

All protocol stacks must support this function. (See *Chapter 3: Protocol Stack Data Structures* for the format of the protocol stack's configuration table.)

GetProtocolStackStatistics

Index 1

Returns a pointer to the protocol stack’s statistics table.

Syntax

```
#include <odi.h>

PS_STATS_TABLE *GetProtocolStackStatistics (
    void *StackIdentifier );
```

Input Parameters

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—**LSL_ConfigTable.LMaxNumberOfStacks**), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

StatsTable	Returns a pointer to the protocol stack statistics table.
NULL	<i>StackIdentifier</i> is invalid

Remarks

All protocol stacks must support this function. (See *Chapter 3: Protocol Stack Data Structures* for the format of the protocol stack’s statistics table.)

GetProtocolStringForBoard

Index 7

Obtains a unique ID string for a board and protocol stack combination.

Syntax

```
#include <odi.h>

ODISTAT GetProtocolStringForBoard (
    UINT32 BoardNumber,
    MEON_STRING *PrintString,
    void *StackIdentifier );
```

Input Parameters

BoardNumber

The board number of the protocol stack to return a unique ID string for.

PrintString

Pointer to a buffer where a unique ID string for a protocol stack is returned.

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—

LSL_ConfigTable.LMaxNumberOfStacks), or the pointer is a pointer to a stack chain node.

Output Parameters

PrintString

Zero is placed at the start of the buffer if no string is returned. The maximum number of MEONs, including the NULL terminator, is 255.

Return Values

- ODISTAT_SUCCESSFUL ID string was successfully obtained.
- ODISTAT_BAD_PARAMETER The MLID corresponding to the requested board number or the protocol stack corresponding to the specified *StackIdentifier* does not exist.

Remarks

For example, an IPX protocol stack might return a string similar to "Network FADE2200" for the board on which the protocol stack is functioning. In this string, the IPX's network number, "FADE2200", is being used with that particular board. A TCP/IP protocol stack might return a string similar to "128.34.31.01".



ODISTAT_SUCCESSFUL is returned even if a NULL is placed in the buffer pointed to by *PrintString*; this implies that there is no unique ID string for a board and protocol stack combination.

MLIDDeRegistered

Index 4

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

Syntax

```
#include <odi.h>

void MLIDDeRegistered (
    UINT32 BoardNumber,
    void *StackIdentifier );
```

Input Parameters

- BoardNumber*
 The number of the board that has deregistered from the LSL.
- StackIdentifier*
 Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—**LSL_ConfigTable.LMaxNumberOfStacks**), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

None.

Remarks

The LSL invokes **MLIDDeRegistered** whenever the logical board that a protocol stack is using has deregistered. The protocol stack can use this information in any way it chooses and can even discard it. However, the specified board will not be available for packet transmission or reception.



When this function is called, a Prescan TX, RX, or default chain stack, must remove its node from the appropriate chain before this call returns.

PromiscuousState

Index 5

Allows MLIDs to notify protocol stacks through the LSL that their promiscuous modes have changed.

Syntax

```
#include <odi.h>

ODISTAT PromiscuousState (
    UINT32 BoardNumber,
    UINT32 PromiscuousMask,
    void *StackIdentifier );
```

Input Parameters

BoardNumber

The number of the board whose promiscuous mode has changed.

PromiscuousMask

The current state of the promiscuous modes bit mask as defined for MLID control function **PromiscuousChange**.

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—

LSL_ConfigTable.LMaxNumberOfStacks), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	Protocol stack notified of promiscuous mode changes.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified <i>StackIdentifier</i> does not exist.
ODISTAT_BAD_COMMAND	PromiscuousStatus function is not supported by the protocol stack.

Remarks

This function is useful in allowing specialized protocol stacks (for example, bridges) to be notified of a change in the promiscuous state of an MLID they are using and to take appropriate steps.

The implementation of this function is optional. If this function is not implemented, a function which returns ODISTAT_BAD_COMMAND must be used in place of the **PromiscuousState** function.

ProtocolManagement

Index 8 (0x08)

Provides a generic way to define protocol dependent functions.

Syntax

```
#include <odi.h>
ODISTAT ProtocolManagement (
    ECB          *ManagementECB,
    void         *StackIdentifier);
```

Input Parameters

ManagementECB
Pointer to the ECB that contains the management information. The first byte of the *ECB_ProtocolID* field must be greater than 0x40 and less than 0x7F.

StackIdentifier
Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—**LSL_ConfigTable.LMaxNumberOfStacks**), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The command was successfully executed. The ECB is returned to the caller.
ODISTAT_RESPONSE_DELAYED	The requested operation was successfully started, but will complete asynchronously. The ECB is not returned. The ManagementECB event service routine will be called with the completion code when the command has finished execution.
ODISTAT_BAD_PARAMETER	The first byte of the <i>ECB_ProtocolID</i> field is not greater than 0x40 or less than 0x7F.
ODISTAT_BAD_COMMAND	Protocol management functions are not supported.
ODISTAT_NO_SUCH_HANDLER	The Protocol ID value is not supported.

Remarks

This control function is a generic interface between protocols and protocol dependent management functions.

The implementation of this function is optional. If this function is not implemented, a function which returns ODISTAT_BAD_COMMAND must be used in place of the ProtocolManagement function.

The management ECB is in the form of an ECB; however, all fields below the *ECB_ProtocolID* field can be redefined by the protocol.

The *ECB_ProtocolID* field is defined as a 6-byte string that uniquely identifies the protocol. The first character of the string must be greater than or equal to 0x41 (A) and less than or equal to 0x7E (~). The remaining characters are defined by the protocol.

If the first character of the *ECB_ProtocolID* field is not greater than or equal to 0x41 (A) and less than or equal to 0x7E (~), ODISTAT_BAD_PARAMETER should be returned as the completion code.

If the protocol does not recognize the value in the *ECB_ProtocolID* field, ODISTAT_NO_SUCH_HANDLER should be returned as the completion code.

If the protocol must respond asynchronously to the management request, it should queue the ECB internally, and `ODISTAT_RESPONSE_DELAYED` should be returned as the completion code.

When the queued request is completed, the protocol should place the ECB on the LSL hold event queue by calling **`CLSL_HoldEvent`**. The LSL will then process the ECB during the next call to **`CLSL_ServiceEvents`**.

Unbind

Index 3

Unbinds a protocol stack from an adapter/frame type (logical board) combination.

Syntax

```
#include <odi.h>

ODISTAT Unbind (
    UINT32 BoardNumber
    MEON_STRING *UserParmString,
    void *StackIdentifier );
```

Input Parameters

BoardNumber

The board number for the protocol stack to unbind from.

UserParmString

Pointer to an optional user specified MEON parameter string that is implementation dependent; NULL if unused.

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the *StackIdentifier* parameter is less than the maximum number of bound protocol stacks supported—

LSL_ConfigTable.LMaxNumberOfStacks), or the pointer is a pointer to a stack chain node.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully unbound from an adapter/frame type (logical board) combination.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified <i>StackIdentifier</i> does not exist.
ODISTAT_ITEM_NOT_PRESENT	The specified binding does not exist.

Remarks

All protocol stacks must support this function.

Protocol stacks should place themselves in a safe state and then call **CLSL_UnbindStack**.

After this routine successfully returns, packet reception between the specified protocol stack and the logical board is disabled.

CLSL_DeRegisterStack performs this operation implicitly.

chapter **7** *Overview of the LSL*

Chapter Overview

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

Link Support Layer (LSL)

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

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

In addition, the LSL performs the following services:

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

Overview of the LSL **8-1**

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

Completion Codes

The following are the completion codes returned by the LSL:

```

ODISTAT_SUCCESSFUL
ODISTAT_RESPONSE_DELAYED
ODISTAT_SUCCESS_TAKEN
ODISTAT_BAD_COMMAND
ODISTAT_BAD_PARAMETER
ODISTAT_DUPLICATE_ENTRY
ODISTAT_FAIL
ODISTAT_ITEM_NOT_PRESENT
ODISTAT_NO_MORE_ITEMS
ODISTAT_MLID_SHUTDOWN
ODISTAT_NO_SUCH_HANDLER
ODISTAT_OUT_OF_RESOURCES
ODISTAT_RX_OVERFLOW
ODISTAT_IN_CRITICAL_SECTION
ODISTAT_TRANSMIT_FAILED
ODISTAT_PACKET_UNDELIVERABLE
ODISTAT_CANCELED

```

Specification Version String

In order to identify which version of this specification an LSL conforms to, a version string (the “specification version string”) is embedded within the LSL. The specification version string number (1. for this specification) is the actual

8-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

version number of the specification. The following is the specification version string for this specification; it is located in the LSL where the global variable declarations are made:

```
MEON_STRING  CODISPEC[ ] = "ODI_CSPEC_VERSION: 1.11";
```

Overview of the LSL **8-3**

8-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

chapter 8 *LSL Data Structures*

Chapter Overview

This chapter describes the Link Support Layer (LSL) configuration and statistics tables and each of the fields in these structures.

LSL Configuration Table

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

LSL Configuration Table Structure Sample Code

```
typedef struct _LSL_CONFIG_TABLE_  
{  
    UINT16      LConfigTableMajorVer;  
    UINT16      LConfigTableMinorVer;  
    MEON_STRING *LSLLongName;  
    MEON_STRING *LSLShortName;  
    UINT16      LSLMajorVer;  
    UINT16      LSLMinorVer;  
    UINT32      LMaxNumberOfBoards;  
    UINT32      LMaxNumberOfStacks;  
    UINT32      LConfigTableReserved0;  
    UINT32      LConfigTableReserved1;  
    UINT32      LConfigTableReserved2;  
    UINT8       LSLCFG_ODISpecMajorVer;  
    UINT8       LSLCFG_ODISpecMinorVer;  
    UINT16      LConfigTableReserved3;  
    UINT32      LSLCFG_SystemFlags;  
}
```

```

UINT32      LSLCFG_SmallECBCount;
UINT32      LSLCFG_MediumECBCount;
UINT32      LSLCFG_LargeECBCount;
UINT32      LSLCFG_XLargeECBCount;
UINT32      LSLCFG_HugeECBCount;
UINT32      LSLCFG_SmallECBBelow16Count;
UINT32      LSLCFG_MediumECBBelow16Count;
UINT32      LSLCFG_LargeECBBelow16Count;
UINT32      LSLCFG_XLargeECBBelow16Count;
UINT32      LSLCFG_HugeECBBelow16Count;
UINT32      LSLCFG_SmallECBMinCount;
UINT32      LSLCFG_MediumECBMinCount;
UINT32      LSLCFG_LargeECBMinCount;
UINT32      LSLCFG_XLargeECBMinCount;
UINT32      LSLCFG_HugeECBMinCount;
UINT32      LSLCFG_SmallECBMaxCount;
UINT32      LSLCFG_MediumECBMaxCount;
UINT32      LSLCFG_LargeECBMaxCount;
UINT32      LSLCFG_XLargeECBMaxCount;
UINT32      LSLCFG_HugeECBMaxCount;
UINT32      LSLCFG_SmallECBSize;
UINT32      LSLCFG_MediumECBSize;
UINT32      LSLCFG_LargeECBSize;
UINT32      LSLCFG_XLargeECBSize;
UINT32      LSLCFG_HugeECBSize;
} LSL_CONFIG_TABLE;

```

Table 9-1
LSL Configuration Table Field Descriptions

Field	Description
LConfigTableMajorVer	This field has the major version number of the LSL configuration table. Use CLSL_CFG_TABLE_MAJOR_VER, defined in ODI.H.
LConfigTableMinorVer	This field has the minor version number of the LSL configuration table. Use CLSL_CFG_TABLE_MINOR_VER, defined in ODI.H.
LSLLongName	Pointer to a NULL terminated MEON string containing the LSL long name.
LSLSHORTName	Pointer to a NULL terminated MEON string containing the LSL short name.
LSLMajorVer	This field has the major version number of the LSL. The number in this field is a decimal number.
LSLMinorVer	This field has the minor version number of the LSL (0 through 99). The number in this field is a decimal number.
LMaxNumberOfBoards	This field contains the maximum number of logical boards that the LSL can handle.
LMaxNumberOfStacks	This field contains the maximum number of bound protocol stacks that the LSL can to handle.
LConfigTableReserved0	This field is reserved for future use.
LConfigTableReserved1	This field is reserved for future use.
LConfigTableReserved2	This field is reserved for future use.
LSLCFG_ODISpecMajorVer	This field contains the major version of the ODI Specification that this version of the C LSL is written too. For example, if the version of the ODI specification is 1.11, the value of this field is 1.
LSLCFG_ODISpecMinorVer	This field contains the minor version of the ODI Specification that this version of the C LSL is written too. For example, if the version of the ODI specification is 1.11, the value of this field is 11.
LconfigTableReserved3	This field is reserved.

Table 9-1
LSL Configuration Table Field Descriptions

Field	Description
LSLCFG_SystemFlags	<p>The bits in this field are defined below:</p> <p>CLSL_CFG_SERVER_BIT When set to 1 this bit indicates the C LSL is running in a server environment. This bit is mutually exclusive with CLSL_CFG_CLIENT_BIT.</p> <p>CLSL_CFG_CLIENT_BIT When set to 1 this bit indicates the C LSL is running in a client environment. This bit is mutually exclusive with CLSL_CFG_SERVER_BIT</p>
LSLCFG_SmallECBCount	This field contains the current number of ECBs that have been allocated for the small ECB pool. The maximum value for this count is defined by LSLCFG_SmallECBMaxCount . The minimum value is defined by LSLCFG_SmallECBMinCount .
LSLCFG_MediumECBCount	This field contains the current number of ECBs that have been allocated for the medium ECB pool. The maximum value for this count is defined by LSLCFG_MediumECBMaxCount . The minimum value is defined by LSLCFG_MediumECBMinCount
LSLCFG_LargeECBCount	This field contains the current number of ECBs that have been allocated for the large ECB pool. The maximum value for this count is defined by LSLCFG_LargeECBMaxCount . The minimum value is defined by LSLCFG_LargeECBMinCount
LSLCFG_XLargeECBCount	This field contains the current number of ECBs that have been allocated for the extra large ECB pool. The maximum value for this count is defined by LSLCFG_XLargeECBMaxCount . The minimum value is defined by LSLCFG_XLargeECBMinCount
LSLCFG_HugeECBCount	This field contains the current number of ECBs that have been allocated for the huge ECB pool. The maximum value for this count is defined by LSLCFG_HugeECBMaxCount . The minimum value is defined by LSLCFG_HugeECBMinCount
LSLCFG_SmallECBBelow16Count	This field contains the current number of ECBs below 16 meg that have been allocated for the small ECB pool. The maximum value for this count is defined by LSLCFG_SmallECBMaxCount . The initial value is zero.

Table 9-1
LSL Configuration Table Field Descriptions

Field	Description
LSLCFG_MediumECBBelow16Count	This field contains the current number of ECBs below 16 meg that have been allocated for the medium ECB below 16 meg pool. The maximum value for this count is defined by LSLCFG_MediumECBMaxCount . The initial value is zero.
LSLCFG_LargeECBBelow16Count	This field contains the current number of ECBs below 16 meg that have been allocated for the large ECB below 16 meg pool. The maximum value for this count is defined by LSLCFG_LargeECBMaxCount . The initial value is zero.
LSLCFG_XLargeECBBelow16Count	This field contains the current number of ECBs below 16 meg that have been allocated for the extra large ECB below 16 meg pool. The maximum value for this count is defined by LSLCFG_XLargeECBMaxCount . The initial value is zero.
LSLCFG_HugeECBBelow16Count	This field contains the current number of ECBs below 16 meg that have been allocated for the huge ECB below 16 meg pool. The maximum value for this count is defined by LSLCFG_HugeECBMaxCount . The initial value is zero.
LSLCFG_SmallECBMinCount	The minimum number of ECBs allocated for the small EBC pool.
LSLCFG_MediumECBMinCount	The minimum number of ECBs allocated for the medium EBC pool.
LSLCFG_LargeECBMinCount	The minimum number of ECBs allocated for the large EBC pool.
LSLCFG_XLargeECBMinCount	The minimum number of ECBs allocated for the extra large EBC pool.
LSLCFG_HugeECBMinCount	The minimum number of ECBs allocated for the huge EBC pool.
LSLCFG_SmallECBMaxCount	The maximum number of ECBs allocated for the small EBC pool.
LSLCFG_MediumECBMaxCount	The maximum number of ECBs allocated for the medium EBC pool.
LSLCFG_LargeECBMaxCount	The maximum number of ECBs allocated for the large EBC pool.
LSLCFG_XLargeECBMaxCount	The maximum number of ECBs allocated for the extra large EBC pool.
LSLCFG_HugeECBMaxCount	The maximum number of ECBs allocated for the huge EBC pool.

Table 9-1
LSL Configuration Table Field Descriptions

Field	Description
LSLCFG_SmallECBSize	This field contains the maximum data size of the ECBs contained in the small ECB pool.
LSLCFG_MediumECBSize	This field contains the maximum data size of the ECBs contained in the medium ECB pool.
LSLCFG_LargeECBSize	This field contains the maximum data size of the ECBs contained in the large ECB pool.
LSLCFG_XLargeECBSize	This field contains the maximum data size of the ECBs contained in the extra large ECB pool.
LSLCFG_HugeECBSize	This field contains the maximum data size of the ECBs contained in the huge ECB pool.

LSL Statistics Table

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

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

LSL Statistics Table Structure Sample Code

```
typedef struct _LOG_BRD_STAT_TABLE_ENTRY_
{
    UINT32          LogBrd_TransmittedPackets;
    UINT32          LogBrd_ReceivedPackets;
    UINT32          LogBrd_UnclaimedPackets;
    UINT32          LogBrd_TxOverloaded;
} LOG_BRD_STAT_TABLE_ENTRY;

LOG_BRD_STAT_TABLE_ENTRY
LogicalBoardStatTablePtr[MaxNumberOfLogicalBoards];

typedef struct _LSL_STATS_TABLE_
{
    UINT16          LStatTableMajorVer;
    UINT16          LStatTableMinorVer;
    UINT32          LNumGenericCounters;
    STAT_TABLE_ENTRY (*LGenericCountersPtr)[];
    UINT32          LNumLogicalBoards;
    LOG_BRD_STAT_TABLE_ENTRY
    (*LogicalBoardStatTablePtr)[];
    UINT32          LNumCustomCounters;
    STAT_TABLE_ENTRY (*LCustomCountersPtr)[];
```

```
}  LSL_STATS_TABLE;
```

Table 9-2
LSL Statistics Table Field Descriptions

Field	Description
LStatTableMajorVer	This field has the major version number of the statistics table (2 for this specification).
LStatTableMinorVer	This field has the minor version number of the statistics table (0 for this specification).
LNumGenericCounters	This field has the total number of generic <i>STAT_TABLE_ENTRY</i> counters in this portion of this table. This field is set to 0x000A for this specification.
LGenericCountersPtr	Pointer to an array of <i>STAT_TABLE_ENTRY</i> counters [<i>LNumGenericCounters</i>].
LNumLogicalBoards	This field has the number of logical boards whose specific statistics are pointed to. Normally this value is the maximum number of boards the LSL can handle.
LogicalBoardStatTablePtr	Pointer to an array of <i>LOG_BRD_STAT_TABLE_ENTRY</i> counters [<i>LNumLogicalBoards</i>].
LNumCustomCounters	This field contains the total number of custom <i>STAT_TABLE_ENTRY</i> counters in this portion of this table.
LCustomCountersPtr	Pointer to an array of <i>STAT_TABLE_ENTRY</i> counters [<i>LCustomCounters</i>].

The following describes the *LOG_BRD_STAT_TABLE_ENTRY* fields.

Table 9-3
LOG_BRD_STAT_TABLE_ENTRY Field Descriptions

Size	Label	Description
UINT32	LogBrd_TransmittedPackets	Number packets that requested transmission by the LSL.
UINT32	LogBrd_ReceivedPackets	Number packets received by the LSL.
UINT32	LogBrd_UnclaimedPackets	Number packets not claimed by any protocol stack.
UINT32	LogBrd_TxOverloaded	This field is incremented when LSL MLID transmit checking detects that a device is overloaded and is consuming too many system resources, such as when MQDepth is too high.

9-8 ODI Specification: Protocol Stacks and MLIDs (C Language)

Example

```

#define          NUM_GENERIC_COUNTERS  10
UINT32          LTotalTxPackets,  LGetECBRequests,
                LGetECBFailures, LAESEventCount,
                LPostponedEvents, LCancelEventFailures,
                LValidBuffersReused, LReserved,
                LTotalRxPackets, LUnclaimedPackets;

STAT_TABLE_ENTRY  LGenericCounters [NUM_GENERIC_COUNTERS] =
{
    { ODI_STAT_UINT32,  &LTotalTxPackets,  NULL },
    { ODI_STAT_UINT32,  &LGetECBRequests,  NULL },
    { ODI_STAT_UINT32,  &LGetECBFailures,  NULL },
    { ODI_STAT_UINT32,  &LAESEventCount,  NULL },
    { ODI_STAT_UINT32,  NULL,  NULL },
    { ODI_STAT_UINT32,  &LCancelEventFailures,  NULL },
    { ODI_STAT_UINT32,  NULL,  NULL },
    { ODI_STAT_UINT32,  NULL,  NULL },
    { ODI_STAT_UINT32,  &LTotalRxPackets,  NULL },
    { ODI_STAT_UINT32,  &LUnclaimedPackets,  NULL },
};

LSL_STATS_TABLE  LSL_StatsTable = {2, 0,
                                     NUM_GENERIC_COUNTERS,
                                     LGenericCounters, MaxNumberOfLogicalBoards,
                                     LogicalBoardStatTablePtr, 0, NULL };

```

Table 9-4
Generic STAT_TABLE_ENTRY Counters Array Fields

Size	Label	Description
UINT32	LTOTALTxPackets	Total number of packets that requested transmission (whether they were actually transmitted or not).
UINT32	LGetECBRequests	Total number of transmit and receive ECB requests.
UINT32	LGetECBFailures	Number of get ECB requests that failed because of unavailable resources.
UINT32	LAESEventsCount	Number of completed AES events.
UINT32	LReserved1	This field is reserved.
UINT32	LCancelEventFailures	Number of cancel events that failed.
UINT32	LReserved2	This field is reserved.
UINT32	LReserved	This field is reserved.
UINT32	LTOTALRxPackets	This field has the total number of <i>CLSL_GetStackECB</i> requests made to the LSL.
UINT32	LUnclaimedPackets	Number of incoming packets that were not claimed by any protocol stack.


The **LSLAPI_ARRAY** function can be accessed through the LSL information block (INFO_BLOCK) using indexes. The location of the various LSL API services in the information block are as follows:

```
#define          LSL_NUM_API          49L
void (*LSLAPI_Array[])(void)=
{
    (void      *)(void))    CLSL_GetSizedECB,
    (void      *)(void))    CLSL_ReturnECB,
    (void      *)(void))    CLSL_CancelEvent,
    (void      *)(void))    CLSL_ScheduleAESEvent,
    (void      *)(void))    CLSL_CancelAESEvent,
    (void      *)(void))    CLSL_GetIntervalMarker,
    (void      *)(void))    CLSL_RegisterStack,
    (void      *)(void))    CLSL_DeRegisterStack,
    (void      *)(void))    CLSL_Reserved,
    (void      *)(void))    CLSL_Reserved,
    (void      *)(void))    CLSL_Reserved,
    (void      *)(void))    CLSL_GetStackECB,
    (void      *)(void))    CLSL_SendPacket,
    (void      *)(void))    CLSL_FastSendComplete,
    (void      *)(void))    CLSL_SendComplete,
    (void      *)(void))    CLSL_RegisterMLID,
    (void      *)(void))    CLSL_GetStackIDFromName,
    (void      *)(void))    CLSL_GetPIDFromStackIDBoard,
    (void      *)(void))    CLSL_GetMLIDControlEntry,
    (void      *)(void))    CLSL_GetProtocolControlEntry,
    (void      *)(void))    CLSL_GetLSLStatistics,
    (void      *)(void))    CLSL_BindStack,
    (void      *)(void))    CLSL_UnbindStack,
    (void      *)(void))    CLSL_AddProtocolID,
    (void      *)(void))    CLSL_GetBoundBoardInfo,
    (void      *)(void))    CLSL_GetLSLConfiguration,
    (void      *)(void))    CLSL_DeRegisterMLID,
    (void      *)(void))    CLSL_RegisterDefaultChain,
    (void      *)(void))    CLSL_RegisterPreScanChain,
    (void      *)(void))    CLSL_Reserved,
    (void      *)(void))    CLSL_DeRegisterDefaultChain,
    (void      *)(void))    CLSL_DeRegisterPreScanChain,
    (void      *)(void))    CLSL_Reserved,
    (void      *)(void))    CLSL_GetStartOfChain,
    (void      *)(void))    CLSL_ReSubmitDefault,
    (void      *)(void))    CLSL_ReSubmitPreScanRx,
    (void      *)(void))    CLSL_ReSubmitPreScanTx,
    (void      *)(void))    CLSL_HoldEvent,
    (void      *)(void))    CLSL_FastHoldEvent,
    (void      *)(void))    CLSL_GetMaxECBBufferSize,
    (void      *)(void))    CLSL_Reserved,
```

```
(void      *)(void))  CLSL_ServiceEvents,
(void      *)(void))  CLSL_ModifyStackFilter,
(void      *)(void))  CLSL_ControlStackFilter,
(void      *)(void))  CLSL_SendProtocolInfoToOtherEngine,
(void      *)(void))  CLSL_SendProtocolInfoToPartner,
(void      *)(void))  CLSL_BindProtocolToBoard,
(void      *)(void))  CLSL_GetMultipleECBs,
(void      *)(void))  CLSL_GetPhysicalAddressOfECB
};

INFO_BLOCK LSLAPIInfoBlock = {
    LSL_NUM_API,
    (void(**)(void))&LSLAPI_Array };

```

Note  For platforms with dynamic linkers, the above steps are unnecessary since the linker takes care of these items. With dynamic linking formats (for example, NLM and ELF) the LSL functions can be called directly by name. Only platforms that do not allow for dynamic linking in this manner must locate the LSL entry points through the underlying platform—for example, entry points are located in Windows NT by a call to the resource manager.

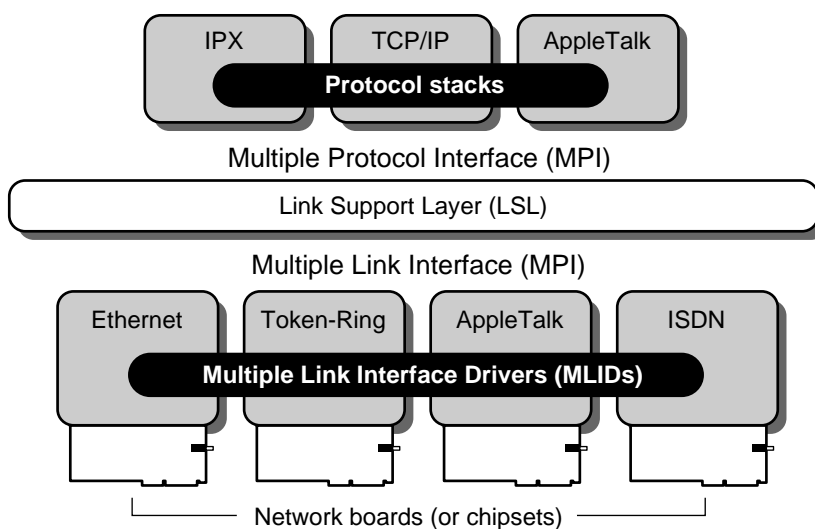
chapter 9

LSL Support Routines

Chapter Overview

This chapter describes the Link Support Layer (LSL) support routines that comprise the Multiple Protocol Interface (MPI) and the Multiple Link Interface (MLI). Figure 1.1 is a block diagram illustrating these interfaces. The routines in this chapter are available to both protocol stacks and MLIDs.

Figure 10-1
LSL Interfaces



LSL API Services

The LSL contains a number of services that are available to protocol stacks and MLIDs. You can invoke these services by calling the LSL support entry point

LSL Support Routines **10-1**

obtained when the protocol stack or MLID locates the LSL. This chapter defines the following functions available from the LSL:

CLSL_AddProtocolID
CLSL_BindProtocolToBoard
CLSL_BindStack
CLSL_CancelAESEvent
CLSL_CancelEvent
CLSL_ControlStackFilter
CLSL_DeRegisterDefaultChain
CLSL_DeRegisterMLID
CLSL_DeRegisterPreScanChain
CLSL_DeRegisterStack
CLSL_FastHoldEvent
CLSL_FastSendComplete
CLSL_GetBoundBoardInfo
CLSL_GetIntervalMarker
CLSL_GetLSLConfiguration
CLSL_GetLSLStatistics
CLSL_GetMaxECBBufferSize
CLSL_GetMLIDControlEntry
CLSL_GetMultipleECBs
CLSL_GetPhysicalAddressOfECB
CLSL_GetPIDFromStackIDBoard
CLSL_GetProtocolControlEntry
CLSL_GetSizedECB
CLSL_GetStackECB
CLSL_GetStackIDFromName
CLSL_GetStartofChain
CLSL_HoldEvent
CLSL_ModifyStackFilter
CLSL_RegisterDefaultChain
CLSL_RegisterMLID
CLSL_RegisterPreScanChain
CLSL_RegisterStack
CLSL_ReSubmitDefault
CLSL_ReSubmitPreScanRx
CLSL_ReSubmitPreScanTx

10-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

CLSL_ReturnECB
 CLSL_ScheduleAESEvent
 CLSL_SendComplete
 CLSL_SendPacket
 CLSL_SendProtocolInfoToOtherEngine (*NetWare Server only*)
 CLSL_SendProtocolInfoToPartner (*NetWare Server only*)
 CLSL_ServiceEvents
 CLSL_UnbindStack

The functions above are accessed through the information block using indexes. The location of the various CLSL functions in the information block are as follows:

Index	Function
0	CLSL_GetSizedECB
1	CLSL_ReturnECB
2	CLSL_CancelEvent
3	CLSL_ScheduleAESEvent
4	CLSL_CancelAESEvent
5	CLSL_GetIntervalMarker
6	CLSL_RegisterStack
7	CLSL_DeRegisterStack
8	Reserved
9	Reserved
10	Reserved
11	CLSL_GetStackECB
12	CLSL_SendPacket
13	CLSL_FastSendComplete
14	CLSL_SendComplete
15	CLSL_RegisterMLID
16	CLSL_GetStackIDFromName
17	CLSL_GetPIDFromStackIDBoard
18	CLSL_GetMLIDControlEntry
19	CLSL_GetProtocolControlEntry
20	CLSL_GetLSLStatistics
21	CLSL_BindStack
22	CLSL_UnbindStack
23	CLSL_AddProtocolID

24	CLSL_GetBoundBoardInfo
25	CLSL_GetLSLConfiguration
26	CLSL_DeRegisterMLID
27	CLSL_RegisterDefaultChain
28	CLSL_RegisterPreScanChain
29	Reserved
30	CLSL_DeRegisterDefaultChain
31	CLSL_DeRegisterPreScanChain
32	Reserved
33	CLSL_GetStartofChain
34	CLSL_ReSubmitDefault
35	CLSL_ReSubmitPreScanRx
36	CLSL_ResubmitPreScanTx
37	CLSL_HoldEvent
38	CLSL_FastHoldEvent
39	CLSL_GetMaxECBBufferSize
40	Reserved
41	CLSL_ServiceEvents
42	CLSL_ModifyStackFilter
43	CLSL_ControlStackFilter
44	CLSL_SendProtocolInfoToOtherEngine (Server only)
45	CLSL_SendProtocolInfoToPartner (Server only)
46	CLSL_BindProtocolToBoard
47	CLSL_GetMultipleECBs
48	CLSL_GetPhysicalAddressOfECB

Note



Access to the CLSL APIs, independent of the link method (dynamic or static), in the information block can be accomplished by using the macro definitions in ODI.H. The macros are listed below. The infoBlock parameter is returned by CLSL_InitEntryPoint. Refer to the API definitions for details on the rest of the parameters.

```

CLSLEntry_GetSizedECB (infoBlock, ecbDataSize, pResourceObj, Below16Meg)
CLSLEntry_ReturnECB (infoBlock, returnedECB)
CLSLEntry_CancelEvent (infoBlock, ecbBuffer)
CLSLEntry_ScheduleAESEvent (infoBlock, timerAESECB)
CLSLEntry_CancelAESEvent (infoBlock, timerAESECB)
CLSLEntry_GetIntervalMarker (infoBlock)
CLSLEntry_RegisterStack (infoBlock, protocolNode, protocolNumber)
CLSLEntry_DeRegisterStack (infoBlock, protocolNumber)

```

10-4 ODI Specification: Protocol Stacks and MLIDs (C Language)


```

CLSLEntry_GetStackECB (infoBlock, lookAheadBuf)
CLSLEntry_SendPacket (infoBlock, sendECB)
CLSLEntry_FastSendComplete (infoBlock, sendECB)
CLSLEntry_SendComplete (infoBlock, sendECB)
CLSLEntry_RegisterMLID (infoBlock, mlidHandlers, mlidConfigTable, boardNumber)
CLSLEntry_GetStackIDFromName (infoBlock, name, protocolNumber)
CLSLEntry_GetPIDFromStackIDBoard (infoBlock, protocolNumber, boardNumber,
    errorStatus)
CLSLEntry_GetMLIDControlEntry (infoBlock, boardNumber, errorStatus)
CLSLEntry_GetProtocolControlEntry (infoBlock, protocolNumber, errorStatus)
CLSLEntry_GetLSLStatistics (infoBlock)
CLSLEntry_BindStack (infoBlock, protocolNumber, boardNumber)
CLSLEntry_UnbindStack (infoBlock, protocolNumber, boardNumber)
CLSLEntry_AddProtocolID (infoBlock, protocolID, protocolName, frameTypeString)
CLSLEntry_GetBoundBoardInfo (infoBlock, boardNumber, stackBuffer)
CLSLEntry_GetLSLConfiguration (infoBlock)
CLSLEntry_DeRegisterMLID (infoBlock, boardNumber)
CLSLEntry_RegisterDefaultChain (infoBlock, stackChainNode)
CLSLEntry_RegisterPreScanChain (infoBlock, pStkChnPreRxNode, pStkChnTxNode)
CLSLEntry_DeRegisterDefaultChain (infoBlock, stackChainNode)
CLSLEntry_DeRegisterPreScanChain (infoBlock, pStkChainRxNode, pStkChainTxNode)
CLSLEntry_GetStartofChain (infoBlock, boardNumber, defaultChainStartNode,
    preScanRxChainStartNode, preScanTxChainStartNode)
CLSLEntry_ReSubmitDefault (infoBlock, stackChainNode, LookAheadBuf)
CLSLEntry_ReSubmitPreScanRx (infoBlock, stackChainNode, lookAheadBuf)
CLSLEntry_ResubmitPreScanTx (infoBlock, stackChainNode, transmitECB)
CLSLEntry_HoldEvent (infoBlock, holdECB)
CLSLEntry_FastHoldEvent (infoBlock, ecbBuffer)
CLSLEntry_GetMaxECBBufferSize (infoBlock)
CLSLEntry_ServiceEvents (infoBlock)
CLSLEntry_ModifyStackFilter (infoBlock, stackIdentifier, boardNumber, newMask,
    pCurrentMask)
CLSLEntry_ControlStackFilter (infoBlock, boardNumber, function, mask,
    parameter1, parameter2)
CLSLEntry_SendProtocolInfoToOtherEngine (infoBlock, protocolNumber,
    protocolInfo, length, infoSendCallBack)
CLSLEntry_SendProtocolInfoToPartner (infoBlock, protocolNumber, protocolinfo,
    length, infoSendCallBack)
CLSLEntry_BindProtocolToBoard (infoBlock, protocolNumber, boardNumber,
    userParmString)
CLSLEntry_GetMultipleECBs (infoBlock, ecbDataSize, pResourceObj, nECBs)
CLSLEntry_GetPhysicalAddressOfECB (info Block, ecb)

```

Locating the LSL

The LSL module must reside in the system before the user can load any Protocol Stack or MLID. On some platforms, such as the NetWare server, the LSL may already be preloaded.


The following is the process usually used to boot an ODI system:

```
LSL.xxx           ;Load Link Support Layer
CNE2000.xxx       ;Load MLID(s)
IPX.xxx           ;Load protocol suites
NW_whatever.xxx   ;Load Redirector/Shell/IFS etc.
```

MLIDs and Protocol Stacks must first obtain the LSL API entry points in order to initialize. Table 10-1 outlines the procedure to find these entry points.

Table 10-1
Finding LSL API Entry Points for an MLID

Actor/Agent	Action
MLID	1. The method used is platform dependent and implementation dependant. 2. Returns an error to the operating system if the MLID fails to find the API entry points.
LSL	3. Returns a pointer to the LSL's initialization entry point.
MLID	4. Calls the LSL's initialization entry point.
LSL	5. Returns a pointer to the start of the LSL's API array and the number of elements in the array.

Note  For platforms with dynamic linkers, the above steps are unnecessary since the linker takes care of these items. With dynamic linking formats (for example, NLM and ELF) the LSL functions can be called directly by name. Only platforms that do not allow for dynamic linking in this manner must locate the LSL entry points through the underlying platform—for example, entry points are located in Windows NT by a call to the resource manager.

The MLID or protocol stack calls the LSL initialization entry point to get the LSL Services entry points for the MLID and the protocol stack.

```
INFO_BLOCK*  CLSL_InitEntryPoint  ( void );
```

CLSL_InitEntryPoint in the above code returns a pointer to the LSL API INFO_BLOCK structure, which is defined as follows:

```
typedef struct _INFO_BLOCK_
{
    UINT32  NumberOfAPIs;
    void ( **SupportAPIArray ) ();
} INFO_BLOCK;
```

Field descriptions:

NumberOfAPIs

The number of elements in the *SupportAPIArray*.

*(**SupportAPIArray)()*

Pointer to an array of function pointers whose functions return voids.

Note



In Windows NT, a call to the resource manager is used to obtain the LSL's entry points.

CLSL_AddProtocolID

Index 23 (0x17)

Allows a protocol stack to register a Protocol ID (PID) for a given frame type and protocol stack combination.

Syntax

```
#include <odi.h>

ODISTAT CLSL_AddProtocolID (
    PROT_ID  *ProtocolID,
    MEON_STRING *ProtocolName,
    MEON_STRING *FrameTypeString );
```

Input Parameters

ProtocolID

Pointer to a byte area of size *PID_SIZE* containing the PID.

ProtocolName

Pointer to a NULL terminated MEON string containing the short name for the protocol stack that receives frames with the appropriate PID.

FrameTypeString

Pointer to a NULL terminated MEON string containing the name of the frame type that is to receive the frames with the specified PID; in other words, the contents of *FrameTypeString* have a value identifying the frame type—for example, *ETHERNET_II*.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The specified PID was successfully registered with the LSL.
ODISTAT_BAD_PARAMETER	The length of the specified protocol short name is equal to 0, is larger than the maximum length allowed, or the frame type identified by the FrameID has not been registered with the LSL. In other words, no board has registered with the LSL using that frame type.
ODISTAT_DUPLICATE_ENTRY	A PID for the specified frame type has already been registered with the LSL, possibly by this protocol stack, or has been registered for another protocol stack, which has yet to register with the LSL.
ODISTAT_NO_MORE_ITEMS	The specified PID and frame type combination could not be registered with the LSL for the named protocol stack, because the combination is already in use by a differently named protocol stack or the named protocol stack and frame type combination already has a different PID registered for it.
ODISTAT_OUT_OF_RESOURCES	The LSL has no resources to register another PID for the specified frame type.

Remarks

A protocol stack invokes the *CLSL_GetPIDFromStackIDBoard* function before it calls this function, because a PID might have been previously registered for the specified protocol and frame type combination.

CLSL_BindProtocolToBoard

Index 46 (0x2E)

Binds a protocol stack to an adapter and frame type (logical board) combination, which enables packet reception.

Syntax

```
#include <odi.h>

ODISTAT CLSL_BindProtocolToBoard (
    UINT32  ProtocolNumber,
    UINT32  BoardNumber
    MEON_STRING  *UserParmString );
```

Input Parameters

- ProtocolNumber*
The Stack ID (SID).
- BoardNumber*
The board number for the protocol stack to bind to.
- UserParmString*
Pointer to an optional user specified MEON parameter string that is implementation dependent; NULL if unused.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully bound to the board.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified SID does not exist.
ODISTAT_DUPLICATE_ENTRY	The specified binding already exists.
ODISTAT_FAIL	The protocol stack failed to bind to the specified MLID.
ODISTAT_ITEM_NOT_PRESENT	No Protocol ID (PID) has been registered for use by this protocol stack with the specified board's frame type; in other words, a PID must be registered by calling CLSL_AddProtocolID for the board's frame type.
ODISTAT_OUT_OF_RESOURCES	The call could not allocate enough memory.

Remarks

When this routine returns successfully, the specified binding has occurred. The bound protocol stack will receive the packets that contain the registered Protocol ID for that stack and that are received by the specified board.

This function differs from **CLSL_BindStack**, because it obtains the protocol stack control service point and executes the protocol stack's bind control service. This function allows a configuration entity to bind a protocol stack to a board in an easy manner.

We recommend that you use this function to bind a protocol stack to a board because of its ease of use, and because this function is NetWare SFTIII aware in NetWare SFTIII environments (it manages primary/secondary server issues of the function transparent to the user). In other words, when a protocol is bound to a board on the primary server, the SFTIII environment automatically causes the same operation to occur on the secondary server.

CLSL_BindStack

Index 21 (0x15)

Binds a protocol stack to an adapter and frame type (logical board) combination. This allows received packets to be passed from the logical board to the protocol stack's receive handler by way of the LSL.

Syntax

```
#include <odi.h>

ODISTAT CLSL_BindStack (
    UINT32  ProtocolNumber,
    UINT32  BoardNumber );
```

Input Parameters

- ProtocolNumber*
The Stack ID (SID).
- BoardNumber*
The board number for the protocol stack to bind to.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully bound to the board.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified SID does not exist.
ODISTAT_DUPLICATE_ENTRY	The specified binding already exists.
ODISTAT_FAIL	The protocol stack failed to bind to the specified MLID.
ODISTAT_ITEM_NOT_PRESENT	No Protocol ID (PID) has been registered for use by this protocol stack with the specified board's frame type. In other words, you must register a PID by calling CLSL_AddProtocolID for the board's frame type.
ODISTAT_OUT_OF_RESOURCES	The call could not allocate enough memory.

Remarks

When this routine returns successfully, the specified binding has occurred. The bound protocol stack will receive the packets that contain the registered Protocol ID for that stack and that are received by the specified board.

This function differs from **CLSL_BindProtocolToBoard**, in that it does not obtain the protocol control service point but is expected to be called from the protocol's control service function *Bind*.

CLSL_CancelAESEvent

Index 4 (0x04)

Cancels a previously scheduled AES event.

Syntax

```
#include <odi.h>

ODISTAT CLSL_CancelAESEvent (
    AES_ECB *TimerAESECB );
```

Input Parameters

TimerAESECB
A pointer to the AES ECB to be canceled.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The specified AES event was canceled.
ODISTAT_ITEM_NOT_PRESENT	The specified AES ECB is not currently scheduled.
ODISTAT_BAD_PARAMETER	The resource tag for the AES ECB is invalid.

Remarks

This function removes a scheduled ECB from the LSL’s event queue. If the AES ECB was canceled, the *ECB_Status* field is set to *ODISTAT_CANCELED* and cast as a UINT16. The defined ESR is not called.

CLSL_CancelEvent

Index 02 (0x02)

Cancels a previously scheduled event.

Syntax

```
#include <odi.h>

ODISTAT CLSL_CancelEvent (
    ECB *ECBBuffer );
```

Input Parameters

ECBBuffer

Pointer to the ECB to be canceled.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The specified event was canceled.
ODISTAT_ITEM_NOT_PRESENT	The specified ECB is not currently scheduled.

Remarks

This function removes an ECB from the LSL's event queue. If the ECB was canceled, the *ECB_Status* field is set to *ODISTAT_CANCELED* and is cast as a UINT16. The defined ESR is not called.

CLSL_ControlStackFilter

Index 43 (0x2B)

This routine allows an MLID or some other agent to call all stacks that have at least one of the mask bits set in the *Mask* parameter with a notification of a change of state of the MLID and that match the protocol stack's filter mask settings. These stacks must also be bound/registered to the board number specified to be notified.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ControlStackFilter (
    UINT32  BoardNumber,
    UINT32  Function,
    UINT32  Mask,
    void    *Parameter1,
    void    *Parameter2);
```

Input Parameters

BoardNumber

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

Function

The control handler function number to be called.

Mask

The filter mask for all stacks to be called.

Parameter1

A possible parameter to pass through to a control handler function.

Parameter2

A possible parameter to pass through to a control handler function.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	All stacks bound/registered with the logical board with their corresponding mask bit set have been notified.
ODISTAT_ITEM_NOT_PRESENT	The specified board does not exist.

Remarks

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



Note This function updates all stacks associated with the physical LAN adapter that the logical board is operating on; in other words, this function updates other protocol stacks that are operating on logical boards that have the same name and instance as the logical board specified by *BoardNumber*. Also, see the discussion of **CLSL_ModifyStackFilter** later in this chapter.

CLSL_DeRegisterDefaultChain

Index 30 (0x1E)

Deregisters a default protocol stack from the specified board.

Syntax

```
#include <odi.h>

ODISTAT CLSL_DeRegisterDefaultChain (
    PS_CHAINED_RX_NODE *StackChainNode );
```

Input Parameters

StackChainNode
Pointer to the node structure defining this chained stack.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully deregistered.
ODISTAT_BAD_PARAMETER	There is no MLID registered for the board number provided in <i>StackChainNode</i> .
ODISTAT_ITEM_NOT_PRESENT	There is no default chain stack registered for this MLID with the provided <i>StackChainNode</i> .

Remarks

After this call, the protocol stack will not receive any more incoming packets from the specified board (unless the protocol stack has an outstanding call to **CLSL_RegisterPrescanChain** or **CLSL_RegisterStack/CLSL_BindStack/CLSL_BindProtocolToBoard**) and must call **CLSL_RegisterDefaultChain** again to start receiving packets.

CLSL_DeRegisterMLID

Index 26 (0x1A)

Called by the MLID to notify the LSL of a board whose number is no longer available to the system.

Syntax

```
#include <odi.h>

ODISTAT CLSL_DeRegisterMLID (
    UINT32 BoardNumber );
```

Input Parameters

BoardNumber
The board number to deregister.

Output Parameters

None.

Return Values

- ODISTAT_SUCCESSFUL The MLID has been successfully deregistered.
- ODISTAT_BAD_PARAMETER The board number is invalid.

Remarks

Once the LSL has been notified that a board is no longer available, the LSL calls all protocol stacks using that board to notify them of the deregistration.

The MLID frees all LSL transmit control blocks and receive control blocks (ECBs) before invoking this function. Remember to check internal send/receive queues for ECBs.

CLSL_DeRegisterPreScanChain

Index 31 (0x1F)

Deregisters a prescan receive and/or a prescan transmit protocol stack from the specified board.

Syntax

```
#include <odi.h>

ODISTAT CLSL_DeRegisterPreScanChain (
    PS_CHAINED_RX_NODE    *PStkChainRxNode,
    PS_CHAINED_TX_NODE    *PStkChainTxNode );
```

Input Parameters

PStkChainRxNode

Pointer to the node structure defining the prescan receive chained stack. If NULL, no prescan receive stack is being deregistered.

PStkChainTxNode

Pointer to the node structure defining the prescan transmit chained stack. If NULL, no prescan transmit stack is being deregistered.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully deregistered.
ODISTAT_BAD_PARAMETER	There is no MLID registered for the board number provided in <i>PStkChainRxNode</i> or <i>PStkChainTxNode</i> ..
ODISTAT_ITEM_NOT_PRESENT	There is no prescan receive stack or there is no prescan transmit stack registered for this MLID with the provided <i>PStkChainRxNode</i> or <i>PStkChainTxNode</i> ..

Remarks

After this call, the protocol stack will not receive any more incoming and/or outgoing packets (unless the protocol stack has an outstanding **CLSL_RegisterDefaultChain** or **CLSL_RegisterStack/CLSL_BindStack/CLSL_BindProtocolToBoard**). The protocol stack must make a call to **CLSL_RegisterPreScanChain** to again start receiving packets.

When deregistering both a receive and transmit protocol stack, the receive stack is deregistered first.

The operation to deregister both the receive and transmit prescan chained stacks is executed monoatomically.

CLSL_DeRegisterStack

Index 07 (0x07)

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

Syntax

```
#include <odi.h>

ODISTAT CLSL_DeRegisterStack (
    UINT32 ProtocolNumber );
```

Input Parameters

ProtocolNumber
The Stack ID (SID).

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully deregistered.
ODISTAT_ITEM_NOT_PRESENT	Either the protocol stack is not registered, or the <i>ProtocolNumber</i> is greater than the maximum number of stacks supported by the LSL.

Remarks

After this call, the protocol stack will not receive any more incoming packets (unless the protocol stack has an outstanding **CLSL_RegisterDefaultChain** or **CLSL_RegisterPreScanChain**). The protocol stack must make the **CLSL_RegisterStack/CLSL_BindStack/CLSL_BindProtocolToBoard** calls to again start receiving packets.

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

10-24 ODI Specification: Protocol Stacks and MLIDs (C Language)

CLSL_FastHoldEvent

Index 38 (0x26)

This routine may improve the performance of drivers that call **CLSL_ServiceEvents** immediately after calling **CLSL_HoldEvent**. **CLSL_FastHoldEvent** may dispatch the ECB directly to the protocol stack.

Syntax

```
#include <odi.h>

void CLSL_FastHoldEvent (
    ECB *ECBBuffer);
```

Input Parameters

ECBBuffer

Pointer to an ECB to be processed immediately.



Note

If the ECB is a receive ECB, the following fields of the ECB must be set before calling this routine:

- ECB_PreviousLink
- ECB_Status
- ECB_DriverWorkSpace
- ECB_DataLength
- ECB_FragmentCount
- ECB_Fragment (addresses and lengths)

Any other fields (for example, *ECB_ProtocolID*) that the protocol stack requires must be filled in by the protocol stack from the LOOKAHEAD structure before returning the ECB buffer to be filled during the call to **CLSL_GetStackECB**.

Output Parameters

None.

Return Values

None.

Remarks

This process can cause the board’s receive handling routine to be called.

Normally, the ECB is placed on the LSL’s event queue for processing using *CLSL_HoldEvent*. When the adapter finishes servicing its hardware, it calls *CLSL_ServiceEvents* to inform the LSL to process any ECBs the adapter has placed there. *CLSL_FastHoldEvent* may allow intelligent adapters that have finished servicing the adapter hardware by the time the ECB is presented to the LSL to reduce latency by passing the ECB directly to the protocol stack for processing.

The event service routine that is called by **CLSL_FastHoldEvent** must not poll.

Note



This functionality is not possible on any multi-processor capable platforms.

CLSL_FastSendComplete

Index 13 (0x0D)

This routine improves the performance of drivers that call **CLSL_ServiceEvents** immediately after calling **CLSL_SendComplete**.

CLSL_FastSendComplete dispatches the ECB directly to the protocol stack.

Syntax

```
#include <odi.h>

void CLSL_FastSendComplete (
    ECB *SendECB );
```

Input Parameters

SendECB

Pointer to a transmit ECB, which is to be processed immediately.

Output Parameters

None.

Return Values

None.

Remarks

This process can cause the board’s send handling routine to be called .

Normally, the ECB is placed on the LSL’s event queue for processing using **CLSL_SendComplete**. When the adapter finishes servicing its hardware, it calls **CLSL_ServiceEvents** to inform the LSL to process any ECBs the adapter has placed there. **CLSL_FastSendComplete** allows intelligent adapters that have finished servicing the adapter hardware by the time the ECB is presented to the LSL to reduce latency by passing the ECB directly to the protocol stack for processing.

The event service routine that is called by **CLSL_FastSendComplete** must not poll.

Note



Passing the ECB directly to the protocol stack may not be possible on multi-processor platforms.

CLSL_GetBoundBoardInfo

Index 24 (0x18)

Allows a protocol stack to check the LSL to determine the registered protocol stacks and Protocol IDs that are bound or the boards that the protocol stacks are registered for.

Syntax

```
#include <odi.h>

ODISTAT CLSL_GetBoundBoardInfo (
    UINT32 BoardNumber,
    UINT32*StackBuffer);
```

Input Parameters

BoardNumber

The value of the logical board that is checked to determine the protocol stacks bound to it.

Input/Output Parameters

StackBuffer

Pointer to a array of UINT32 sized buffers where the number of bound stacks and their Stack IDs (SIDs) are returned.

On entry, the first entry in the stack buffer contains the total size of the stack buffer in bytes.



Note

This buffer must be large enough to contain the maximum number of stacks (the maximum number of stacks is available from the LSL configuration table) and a count field (buffer size = max stacks + 1).

On exit, the first entry in the buffer contains the number of bound stacks to the specified board, followed by the bound stacks' SIDs.

Return Values

ODISTAT_SUCCESSFUL	The command was successfully completed.
ODISTAT_NO_MORE_ITEMS	The board number does not exist and there are no boards at higher values.
ODISTAT_ITEM_NOT_PRESENT	The board number does not exist but there might be boards at higher values.
ODISTAT_OUT_OF_RESOURCES	The stack buffer provided is insufficient for the parameters to be returned. The first entry in the stack buffer contains the number of bytes required for the count of bound protocol stacks and their SIDs that the MLID wants to return.

CLSL_GetIntervalMarker

Index 05 (0x05)

Returns a timing marker in milliseconds.

Syntax

```
#include <odi.h>

UINT32 CLSL_GetIntervalMarker ( void );
```

Input Parameters

None.

Output Parameters

None.

Return Values

Milliseconds A time value in milliseconds.

Remarks

The timing marker is used for machine-independent time measurement. The actual value returned has no relation to any real-world absolute time. However, when time marker values are compared with each other, the difference is elapsed time in milliseconds.

Note



This function is intended for use in the timing of low resolution events only.

CLSL_GetLSLConfiguration

Index 25 (0x19)

Returns a pointer to the LSL configuration table

Syntax

```
#include <odi.h>

LSL_CONFIG_TABLE *CLSL_GetLSLConfiguration ( void );
```

Input Parameters

None.

Output Parameters

None.

Return Values

ConfigTable Pointer to the LSL configuration table.

Remarks

The LSL configuration table is normally used to obtain the LSL's current version number. The version number can be used to determine if certain LSL features are present.

See Also

See *Chapter 9: LSL Data Structures* for more on the LSL configuration table.

10-32 ODI Specification: Protocol Stacks and MLIDs (C Language)

CLSL_GetLSLStatistics

Index 20 (0x14)

Returns a pointer to the LSL statistics table.

Syntax

```
#include <odi.h>

LSL_STATS_TABLE *CLSL_GetLSLStatistics ( void );
```

Input Parameters

None.

Output Parameters

None.

Return Values

StatsTable Pointer to the LSL statistics table.

See Also

See *Chapter 9: LSL Data Structures* for a description of the LSL statistics table.

CLSL_GetMaxECBBufferSize

Index 39 (0x27)

Returns the maximum LSL ECB buffer size.

Syntax

```
#include <odi.h>

UINT32 CLSL_GetMaxECBBufferSize ( void );
```

Input Parameters

None.

Output Parameters

None.

Return Values

ECBBufferSize
The maximum LSL ECB buffer size.

0x00000000
No LSL ECB buffers defined.

Remarks

This size is the maximum amount of buffer space available, including one fragment to store data in after the ECB's defined fields have taken up their required space. In other words, the returned value allows for the fields defined in the ECB structure.

CLSL_GetMLIDControlEntry

Index 18 (0x12)

Returns a pointer to the specified MLID's information block, which describes the control handler routines.

Syntax

```
#include <odi.h>

INFO_BLOCK  *CLSL_GetMLIDControlEntry (
    UINT32   BoardNumber,
    ODISTAT  *ErrorStatus );
```

Input Parameters

BoardNumber

The logical board number whose information block is desired.

Output Parameters

ErrorStatus

If the returned value is a NULL pointer, *ErrorStatus* is one of the following:

ODISTAT_NO_MORE_ITEMS

The board number does not exist and there are no boards at higher values.

ODISTAT_ITEM_NOT_PRESENT

The board number does not exist but there might be boards at higher values.

Return Values

APIInfoBlock	A pointer to the MLID's information block, which describes the array of MLID control handler routines.
NULL	An error condition that is indicated by <i>*ErrorStatus</i> .

Remarks

The MLID control handler routines can be called directly by a protocol or an application to obtain configuration information and to issue defined commands.

See Also

See Chapter 18, "MLID Control Routines" for the defined MLID control functions.

CLSL_GetMultipleECBs

Index 47 (0x2F)

Returns a pointer to the first ECB on the linked list. MLIDs and protocol stacks use this call to get linked lists of transmit buffers and receive buffers.

Syntax

```
#include <odi.h>

ECB *CLSL_GetMultipleECBs (
    UINT32 ECBDataSize,
    void *pResourceObj,
    UINT32 *nECBs );
```

Input Parameters

ECBDataSize

The amount of data space required (in bytes) for the linked list of transmit and receive buffers.



Note

The size of the ECB is not included in *ECBDataSize*. The LSL adds the size of the ECB structure when generating an ECB of *ECBDataSize*.

pResourceObj

Pointer to a platform specific object used for resource management. This value is a pass-through value, and is not interpreted.

nECBs

Pointer to the number of ECBs requested.

Output Parameters

nECBs

Pointer to the number of ECBs allocated.

Return State

Pointer to the first ECB on the linked list. NULL if no ECBs are available.

Remarks

If the number of ECBs requested is larger than the number of ECBs available, this procedure allocates all available ECBs and returns a pointer to the first ECB on the list.

On return *nECBs* points to the number of ECBs allocated. If no ECBs are available, *nECBs* points to a value of zero.

All pointers are logical pointers, and the returned linked list of ECBs is linked according to the ECB structure.

Note



Communications buffers are critical. Protocol stacks must use only a minimal number of buffers concurrently and must return the buffers when they are finished with them, using **CLSL_ReturnECB**.

CLSL_GetPhysicalAddressOfECB

Index 48 (0x030)

Gets the physical address of an LSL ECB.

Syntax

```
#include <odi.h>

ECB *CLSL_GetPhysicalAddressOfECB
    (ECB *ecb);
```

Input Parameters

ecb

Pointer (logical address) to an LSL ECB.

Output Parameters

None.

Return Values

Pointer (physical address) of the ECB structure.

Remarks

This function can be only be used for ECBs obtained via **CLSL_GetSizedECB** or **CLSL_GetMultipleECBs**.

CLSL_GetPIDFromStackIDBoard

Index 17 (0x11)

Returns a pointer to a Protocol ID (PID) that corresponds to a protocol and frame type combination.

Syntax

```
#include <odi.h>

PROT_ID *CLSL_GetPIDFromStackIDBoard (
    UINT32 ProtocolNumber,
    UINT32 BoardNumber,
    ODI_STAT *ErrorStatus);
```

Input Parameters

ProtocolNumber
The Stack ID (SID).

BoardNumber
The board number.

Output Parameters

ErrorStatus
If the returned value is a NULL pointer, *ErrorStatus* is one of the following:

ODI_STAT_BAD_PARAMETER
The SID or the board number does not exist.

ODI_STAT_ITEM_NOT_PRESENT
A Protocol ID has not been registered for the specified board and frame type combination.

Return Values

ProtocolID	A pointer to a byte area of <i>PID_SIZE</i> size, which contains the PID.
NULL	An error condition that is indicated by <i>ErrorStatus</i> .

Remarks

Protocol stacks use the returned PID to fill in the *ECB_ProtocolID* field of all send ECBs. The returned PID is the value assigned to the protocol for the frame type (for example, *ETHERNET_II*) that is represented by the board number.

If a PID is not present, a protocol stack can add its own Protocol ID using **CLSL_AddProtocolID** (see *Chapter 10: LSL Support Routines*).

CLSL_GetProtocolControlEntry

Index 19 (0x13)

Returns a pointer to the specified protocol stack's information block, which describes the control handler routines.

Syntax

```
#include <odi.h>

INFO_BLOCK  *CLSL_GetProtocolControlEntry (
    UINT32   ProtocolNumber,
    ODISTAT  *ErrorStatus );
```

Input Parameters

ProtocolNumber

The Stack ID (SID) whose information block is desired.

Output Parameters

ErrorStatus

If the returned value is a NULL pointer, *ErrorStatus* is one of the following:

ODISTAT_NO_MORE_ITEMS

The SID does not exist and there are no others at higher values.

ODISTAT_ITEM_NOT_PRESENT

The SID does not exist but there might be others at higher values.

Return Values

APIInfoBlock	A pointer to protocol stack's information block, which describes the array of protocol stack control handler routines.
NULL	An error condition that is indicated by <i>ErrorStatus</i> .

Remarks

The protocol control handler routine can be called directly by a protocol or an application to obtain configuration information and to issue defined commands. (See *Chapter 7: Protocol Stack Control Routines* for the defined protocol control functions.)



In previous assembly versions of the ODI specification, the corresponding routine contained a board number parameter, which was used to get the non-chained default or prescan receive protocol stack's control handler routines (for example, it was used before the introduction of chained protocol stacks). Users must now use the **CLSL_GetStartofChain** command to get the control handler routines for default, prescan receive, and prescan transmit protocol stacks.

CLSL_GetSizedECB

Index 00 (0x00)

Called by the MLID or protocol stack for several purposes, such as obtaining transmit or receive buffers; returns a pointer to an ECB.

Syntax

```
#include <odi.h>

ECB *CLSL_GetSizedECB (
    UINT32  ECBDataSize,
    void *pResourceObj,
    BOOLEAN  Below16Meg );
```

Input Parameters

ECBDataSize

The amount of data space required (in bytes) for the transmit and receive buffers.



The size of the ECB is not included in *ECBDataSize*. The LSL adds the size of the ECB structure when generating an ECB of *ECBDataSize*.

pResourceObj

Pointer to a platform specific object used for resource management. This is a pass through value and is not interpreted.

Below16Meg

If set to TRUE, memory for the ECB is allocated from memory below the 16MB boundary. Normally, protocol stacks call with this value set to FALSE.

Output Parameters

None.

Return Values

ECBBuffer	A pointer to an ECB.
NULL	An error condition indicating that there are no free ECBs available.

Remarks

You must keep in mind that communications buffers are a critical resource, and a protocol stack must not use a large number of buffers concurrently.

The protocol stack returns the buffer using **CLSL_ReturnECB** when the protocol stack has finished with the buffer.



Note All buffers allocated by this function are physically and logically contiguous.

An ECB allocated by this function contains only one fragment. The address and the buffer size in the FRAGMENT_STRUCT element of the ECB fragment list must not be altered by the caller.

CLSL_GetStackECB

Index 11 (0x0B)

Called by the MLID to obtain communication buffers from a protocol stack through the LSL.

Syntax

```
#include <odi.h>

ODISTAT CLSL_GetStackECB (
    LOOKAHEAD *LookAheadBuf );
```

Input Parameters

LookAheadBuf

Pointer to a LOOKAHEAD structure, which defines the received packet. See the "Receive Lookahead" section in Chapter 5, "Protocol Stack Packet Reception".

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	<p>The protocol stack has returned a pointer in the LOOKAHEAD structure's ECB field to a receive ECB to be filled with the packet.</p> <p>If the protocol stack was able to get everything it needed from the LOOKAHEAD, the <i>LkAhd_ReturnedECB</i> field will be set to NULL, indicating that additional data does not need to be copied.</p>
ODISTAT_SUCCESS_TAKEN	<p>The protocol stack has accepted the packet and has taken the prefilled LSL ECB associated with the LOOKAHEAD structure's <i>LkAhd_PreFilledECB</i> field.</p>
ODISTAT_OUT_OF_RESOURCES	<p>The LSL was unable to obtain an ECB for this packet.</p>

Remarks

On entry, **CLSL_GetStackECB** requires a pointer to the LOOKAHEAD structure. The returned ECB's fragment count and fragment descriptor fields describe the buffers into which the packet is placed.

Regardless of whether the protocol stack is bound, prescan, or default, the protocol stack is passed LOOKAHEAD data whenever its receive handler is invoked. This data is used to determine into which receive buffers (if any) the data is placed. (Receive buffers can be fragmented.) If the protocol stack determines that it will consume the packet, it must build an ECB that describes the receive buffers and then return that ECB to the MLID through the LSL. The MLID uses the ECB's description of the receive buffers to move the data from the LAN adapter into the described protocol receive buffers. When the MLID has completed the data move, it passes the ECB to the LSL for event completion.

Alternatively, if the LOOKAHEAD structure has an LSL ECB associated with it (*LkAhd_PreFilledECB* is not NULL), the protocol stack can accept the packet by signaling that it has accepted the packet and taken the associated ECB, which it returns to the LSL later. If a protocol stack performs this operation, it must queue the ECB for processing at a later point, since the LOOKAHEAD indication is usually made at privileged time by an MLID. If

the protocol stack chooses not to make use of the provided ECB in the LOOKAHEAD structure and returns its own ECB to be filled, the LSL performs the prefilled ECB to stack ECB copy of data, and by so doing, simplifies the operation of the MLID.



MLIDs that provide a prefilled ECB in the LOOKAHEAD structure do not need to return the ECB to the LSL, since the LSL always returns the prefilled ECB to its buffer pool before returning to the caller of this function. When prefilled ECBs are used, the return value is *ODISTAT_SUCCESS_TAKEN* or *ODISTAT_OUT_OF_RESOURCES*.

If the protocol stack requires the ECB *ProtocolID*, *BoardNumber*, and *ImmediateAddress* fields to be filled in, the protocol stack must get them from the LOOKAHEAD structure and place them in the relevant ECB fields.

When *ODISTAT_SUCCESSFUL* is returned, the *LkAhd_ReturnedECB* may contain a NULL ECB. This allows an MLID to correctly maintain its statistics. This may occur if the protocol stack was able to retrieve all the necessary information from the LOOKAHEAD data.

CLSL_GetStackIDFromName

Index 16 (0x10)

Allows a protocol stack or an application to obtain its own or any other Stack ID (SID).

Syntax

```
#include <odi.h>

ODISTAT CLSL_GetStackIDFromName (
    MEON_STRING  *Name,
    UINT32       *ProtocolNumber );
```

Input Parameters

Name

Pointer to a NULL terminated MEON string containing the short name of the protocol stack.

Output Parameters

ProtocolNumber

Pointer to a UINT32 buffer where this function returns the SID if successful.

Return Values

ODISTAT_SUCCESSFUL	Function completed successfully; a SID has been returned.
ODISTAT_BAD_PARAMETER	The length of the stack name given is greater than the maximum allowed or is equal to 0.
ODISTAT_ITEM_NOT_PRESENT	The named protocol stack is not presently registered.

Remarks

The stack name is not case sensitive.

CLSL_GetStartofChain

Index 33 (0x21)

Returns pointers to pointers to the start of the chains for the prescan transmit, prescan receive, and default stack chains for the specified board.

Syntax

```
#include <odi.h>

ODISTAT CLSL_GetStartofChain (
    UINT32 BoardNumber,
    PS_CHAINED_RX_NODE
    **DefaultChainStartNode,
    PS_CHAINED_RX_NODE
    **PreScanRxChainStartNode,
    PS_CHAINED_TX_NODE
    **PreScanTxChainStartNode );
```

Input Parameters

BoardNumber

The board number for which to obtain the start of the chained protocol stacks.

Output Parameters

DefaultChainStartNode

Pointer to a pointer to the start of the default stack chain for the specified board number.

PreScanRxChainStartNode

Pointer to a pointer to the start of the prescan receive stack chain for the specified board number.

PreScanTxChainStartNode

Pointer to a pointer to the start of the prescan transmit stack chain for the specified board number.

Return Values

ODISTAT_SUCCESSFUL Command successfully executed.
ODISTAT_BAD_PARAMETER Invalid board number.

Remarks

If there are no stacks in the relevant chain, the pointer returned points to a NULL.

CLSL_HoldEvent

Index 37 (0x25)

Allows a protocol stack or MLID to place a previously allocated ECB buffer on the LSL's event queue to be processed at service events time.

Syntax

```
#include <odi.h>

void CLSL_HoldEvent (
    ECB *HoldECB );
```

Input Parameters

HoldECB

Pointer to an ECB to place on the LSL's event queue for processing when **CLSL_ServiceEvents** is executed.

If the ECB is a receive ECB, the following fields of the ECB must be set before calling this routine:

MLID

ECB_PreviousLink

ECB_Status

ECB_DriverWorkSpace

ECB_DataLength

Protocol Stack

ECB_FragmentCount

ECB_Fragment Addresses and Lengths



Note

If the protocol stack gets the ECB from the LSL (**CLSL_GetSizedECB**), the protocol stack must not modify *ECB_FragmentCount* or *ECB_Fragment Addresses and Lengths*.

Any other fields (for example, *ECB_ProtocolID*, *ECB_BoardNumber*, or *ECB_ImmediateAddress*) that the protocol stack requires must be filled in by the protocol stack from the LOOKAHEAD structure before returning the ECB buffer to be filled during the call to **CLSL_GetStackECB**.

Output Parameters

None.

Return Values

An ECB is always successfully placed on the LSL's event queue.

CLSL_ModifyStackFilter

Index 43 (0x2A)

Allows a stack of any type to modify its filter mask.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ModifyStackFilter (
    void *StackIdentifier,
    UINT32 BoardNumber,
    UINT32 NewMask,
    UINT32 *pCurrentMask );
```

Input Parameters

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, value is less than the maximum number of bound protocol stacks supported), or the pointer is a pointer to a stack chain node.

BoardNumber

The logical board number where the filtering of packets to the supplied SID is modified.

NewMask

The new filter mask for the stack and board combination (0 for query). Table 10-2 gives the bit definitions.

Output Parameters

pCurrentMask

Returns a pointer to the current filter mask setting. Table 10-2 gives the bit definitions.

Return Values

ODISTAT_SUCCESSFUL

The new filter mask is set for the stack and board combination, and/or if query, the current filter mask for the stack/board combination is returned.

ODISTAT_BAD_PARAMETER

Either the specified SID or the board does not exist.

ODISTAT_ITEM_NOT_PRESENT

The stack and board combination does not exist.

Remarks

When a bound stack registers with the LSL, it defaults to only receiving direct, supported multicast, and broadcast addressed packets. A chained protocol stack must specify the type of packets it wants to receive in its filter mask when it registers for a board.

A stack, after binding/registering with a board, can modify its filter to allow it to specify the type of packets that it now wishes to have passed to it by the LSL. Normally, bound protocol stacks use the default setting for the mask with the filter functions being used by chained protocol stacks for bridging, traffic monitoring, etc.

Note



A chained protocol stack can be configured as a network monitor by filtering for all direct, remote, and error packets.

Table 10-2
CLSL_ModifyStackFilter

Name	Description
DT_MULTICAST	Receive supported, group addressed packets such as multicast addressed packets that the MLID is configured to receive.
DT_BROADCAST	Receive broadcast packets.
DT_REMOTE_UNICAST	Receive directed packets whose addresses do not match that of the MLID's node address—for example, if this bit is set and the MLID is in promiscuous mode, it passes received, directed packets destined for another workstation (the group address bit is not set in the destination address of the packet's MAC header). Note, routing of receive packets can occur if source routing is enabled.
DT_REMOTE_MULTICAST	Receive unsupported, group addressed packets that the MLID is not configured to receive—for example, multicast packets.
DT_SOURCE_ROUTE	Receive source routed packets.
DT_ERROR	Receive packets that the MLID is configured to receive and that contain errors.
DT_MAC_FRAME	Receive frame packets that are non-data frames. Note, MAC frames do not contain source routing information.
DT_DIRECT	Receive directed packets.
DT_RX_PRIORITY	Receive priority packets.

CLSL_RegisterDefaultChain

Index 27 (0x1B)

Registers a default protocol stack for the specified board.

Syntax

```
#include <odi.h>

ODISTAT CLSL_RegisterDefaultChain (
    PS_CHAINED_RX_NODE *StackChainNode );
```

Input Parameters

StackChainNode
Pointer to the node for the chained protocol stack.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully registered.
ODISTAT_BAD_PARAMETER	Either the specified board does not exist or an invalid stack chain position was requested.
ODISTAT_DUPLICATE_ENTRY	Requested chain position already occupied.

Remarks

After this call, the protocol stack will receive incoming packets from the specified board.

On reception, the default chain stack is called with a pointer to a LOOKAHEAD structure.

PS_CHAINED_RX_NODE Structure

```
typedef struct _PS_CHAINED_RX_NODE_
{
    struct _PS_CHAINED_RX_NODE_ *StackChainLink;
    UINT32      StackChainBoardNumber;
    CHNPOS      StackChainPositionRequested;
    ODISTAT      (*StackRxChainHandler)(LOOKAHEAD*,
                                         struct _PS_CHAINED_RX_NODE_ *);
    INFO_BLOCK   *StackChainControl;
    UINT32      StackChainFilter;
    void         *StackChainContext;
    void         *StackChainResourceObj;
} PS_CHAINED_RX_NODE;
```

Field Descriptions

StackChainLink

Pointer to the next node in the chain, which is filled in by the LSL—NULL terminated.

StackChainBoardNumber

Logical board number to register for.

StackChainPositionRequested

Position in chain desired.

StackChainPositionRequested is defined by the following values:

CHNPOS_FIRST_MUST

Load at very first position in chain.

CHNPOS_FIRST_NEXT

Load at next available position at front of chain.

CHNPOS_LOAD_ORDER

Chain position dependent on load order.

CHNPOS_LAST_NEXT

Load at next available position at end of chain.

CHNPOS_LAST_MUST

Load at very end of chain.

StackRxChainHandler

Pointer to the default stack's receive handler. See "Protocol Receive Handler for Prescan and Default Stacks" in *Chapter 5: Protocol Stack Packet Reception*.

StackChainControl

Pointer to the default stack's information block for control handler routines.

StackChainFilter

The filter mask for the stack and board combination. Refer to **CLSL_ModifyStackFilter** for a description of the operation and values for this filter mask.

StackChainContext

Pointer to context used by the chained stack; in other words, a pointer to any context that the chained protocol stack desires when using the chained receive node structure.

StackChainResourceObj

Pointer to a platform specific object used for resource management. This value is a pass through value and is not interpreted.

CLSL_RegisterMLID

Index 15 (0x0F)

Called by the driver initialization procedure to register a logical board with the LSL.

Syntax

```
#include <odi.h>

ODISTAT CLSL_RegisterMLID (
    MLID_REG *MLIDHandlers,
    MLID_CONFIG_TABLE *MLIDConfigTable,
    UINT32 *BoardNumber );
```

Input Parameters

MLIDHandlers

Pointer to MLID's registration structure containing pointers to the MLID transmit and control handling interfaces.

MLIDConfigTable

Pointer to the MLID configuration table that is attempting to register with the LSL. This table is complete, except for values returned by the LSL. In other words, it has *MLIDCFG_MaxFrameSize*, *MLIDCFG_FrameID*, pointers to names, etc. filled except for the logical board number, which is returned by this call.

BoardNumber

Pointer to the logical board number returned for this MLID.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The MLID was successfully registered with the LSL.
ODISTAT_OUT_OF_RESOURCES	There is no more room to register another MLID with the LSL.

Remarks

The MLID uses the logical board number returned by the LSL when referring itself to the LSL and any other entity of the ODI specification—for example, protocol stacks.

MLID_REG Structure

```
typedef struct _MLID_REG_
{
    void (*MLIDSendHandler)(ECB*, void *);
    INFO_BLOCK *MLIDControlHandler;
    void *MLIDSendContext;
    void *MLIDResourceObj;
    void *MLIDModuleHandle;
} MLID_REG;
```

Field Descriptions

- MLIDSendHandler*
Pointer to the MLID’s transmit function.
Note: **MLIDSendHandler** is called with a transmit ECB and a pointer to the **MLIDSendContext** that it provided to the LSL when it registered successfully with the LSL.
- MLIDControlHandler*
Pointer to the MLID’s information block (INFO_BLOCK) for it’s control functions.
- MLIDSendContext*
Pointer to a MLID defined context to be passed to the MLID when it’s send handler is called.

MLIDResourceObj

Pointer to a platform specific object that is used for resource management.
This value is a pass through value and is not interpreted.

MLIDModuleHandle

Pointer to the module handle provided by the loader to the MLID when the loader loaded the MLID.

CLSL_RegisterPreScanChain

Index 28 (0x1C)

Registers a prescan receive protocol stack and/or a prescan transmit protocol stack for the specified board.

Syntax

```
#include <odi.h>

ODISTAT CLSL_RegisterPreScanChain (
    PS_CHAINED_RX_NODE *PStkChnPreRxNode,
    PS_CHAINED_TX_NODE *PStkChnPreTxNode );
```

Input Parameters

- *PStkChnPreRxNode*
Pointer to the node for the chained received protocol stack. If NULL, no prescan receive chain stack is being registered.
- *PStkChnPreTxNode*
Pointer to the node for the chained transmit protocol stack. If NULL, no prescan transmit chain stack is being registered.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully registered.
ODISTAT_BAD_PARAMETER	Either the specified board does not exist or an invalid stack chain position was requested.
ODISTAT_DUPLICATE_ENTRY	Requested chain position already occupied.



Important If any value other than *ODISTAT_SUCCESSFUL* is returned, neither input stack is registered; either both input stacks register or neither one does.

Remarks

- The operation to register both the receive and transmit prescan chained stacks is executed monoatomically.
- When deregistering both a receive and transmit protocol stack, the receive stack is deregistered first.
- The mechanism for denoting whether the chained receive protocol stack accepts or rejects a received packet is defined by the protocol receive handler. For information on the protocol receive handler, see “Protocol Receive Handler for Prescan and Default Stacks” in *Chapter 5: Protocol Stack Packet Reception*.
- The mechanism for denoting whether the chained transmit protocol stack accepts or rejects an ECB is defined by the protocol transmit handler. For information on the protocol transmit handler, see “Protocol Transmit Handler for Prescan and Default Stacks” in *Chapter 6: Protocol Stack Packet Transmission*.

PS_CHAINED_RX_NODE Structure

```
typedef struct _PS_CHAINED_RX_NODE_
{
    struct _PS_CHAINED_RX_NODE_ *StackChainLink;
    UINT32 StackChainBoardNumber;
    CHNPOS StackChainPositionRequested;
    ODISTAT (*StackRxChainHandler)(LOOKAHEAD*,
                                   struct _PS_CHAINED_RX_NODE_ *);
}
```

```

        INFO_BLOCK      *StackChainControl;
        UINT32           StackChainFilter;
        void             *StackChainContext;
        void             *StackChainResourceObj;
    }    PS_CHAINED_RX_NODE;

```

Field Descriptions

StackChainLink

Pointer to the next node in the chain. This field is filled in by the LSL and is NULL terminated.

StackChainBoardNumber

Logical board number to register for.

StackChainPositionRequested

Position in chain desired.

StackChainPositionRequested is defined by the following values:

CHNPOS_FIRST_MUST

Load at very first position in chain.

CHNPOS_FIRST_NEXT

Load at next available position at front of chain.

CHNPOS_LOAD_ORDER

Chain position dependent on load order.

CHNPOS_LAST_NEXT

Load at next available position at end of chain.

CHNPOS_LAST_MUST

Load at very end of chain.

StackRxChainHandler

Pointer to the prescan stack's receive handler. See *Chapter 5: Protocol Stack Packet Reception* for information on the receive handler.

StackChainControl

Pointer to the prescan receive information block for control handler routines.

StackChainFilter

The filter mask for the stack and board combination. Refer to **CLSL_ModifyStackFilter** for a description of the operation and values for this filter mask.

StackChainContext

Pointer to context used by the chained stack; in other words, a pointer to any context that the chained protocol stack desires when using the chained receive node structure.

StackChainResourceObj

Pointer to a platform specific object used for resource management. This value is a pass through value and is not interpreted.

PS_CHAINED_TX_NODE Structure

```
typedef struct _PS_CHAINED_TX_NODE_
{
    struct _PS_CHAINED_TX_NODE_ *StackChainLink;
    UINT32      StackChainBoardNumber;
    CHNPOS      StackChainPositionRequested;
    ODISTAT      (*StackTxChainHandler)(ECB*,
        struct _PS_CHAINED_TX_NODE_ *);
    INFO_BLOCK   *StackChainControl;
    UINT32      StackChainFilter;
    void         *StackChainContext;
    void         *StackChainResourceObj;
} PS_CHAINED_TX_NODE;
```

Field Descriptions*StackChainLink*

Pointer to the next node in the chain. This field is filled in by the LSL and is NULL terminated.

StackChainBoardNumber

Logical board number to register for.

StackChainPositionRequested

Position in chain desired.

StackChainPositionRequested is defined by the following values:

CHNPOS_FIRST_MUST

Load at very first position in chain.

CHNPOS_FIRST_NEXT

Load at next available position at front of chain.

CHNPOS_LOAD_ORDER

Chain position dependent on load order.

CHNPOS_LAST_NEXT

Load at next available position at end of chain.

CHNPOS_LAST_MUST

Load at very end of chain.

StackTxChainHandler

Pointer the prescan stack's transmit handler. See *Chapter 6: Protocol Stack Packet Transmission* for information on packet transmission.

StackChainControl

Pointer to the prescan transmit information block for control handler routines.

StackChainFilter

This field is not used for prescan transmit chained protocol stacks.

StackChainContext

Pointer to context used by the chained stack; in other words, a pointer to any context that the chained protocol stack desires when using the chained transmit node structure.

StackChainResourceObj

Pointer to a platform specific object used for resource management. This value is a pass through value and is not interpreted.

CLSL_RegisterStack

Index 06 (0x06)

Registers a bound protocol stack with the LSL and returns an LSL assigned handle for the stack (the Stack ID or SID).

Syntax

```
#include <odi.h>

ODISTAT CLSL_RegisterStack (
    PS_BOUND_NODE *ProtocolNode,
    UINT32 *ProtocolNumber );
```

Input Parameters

ProtocolNode

Pointer to a bound protocol stack node structure.

Input/Output Parameters

ProtocolNumber

On entry, pointer to a buffer used to return the SID for the bound protocol stack.

On exit, pointer to the SID.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was successfully registered.
ODISTAT_BAD_PARAMETER	The stack name length is greater than the maximum allowed or is equal to 0.
ODISTAT_DUPLICATE_ENTRY	The specified stack is already registered.
ODISTAT_OUT_OF_RESOURCES	The maximum number of stacks is already registered.

Remarks

The protocol stack will not start receiving packets from the MLID until it has been bound to the MLID by a call to **CLSL_BindStack** or **CLSL_BindProtocolToBoard**.

When a bound stack registers with the LSL, its filter mask defaults to only receiving direct, supported multicast, and broadcast addressed packets. A bound stack can, after registering, modify its filter to allow it to specify the type of packets that it now wishes to have passed to it by the LSL. Normally, bound protocol stacks use the default setting for the mask with the filter functions being used by chained protocol stacks for bridging, traffic monitoring, etc.

PS_BOUND_NODE Structure

```
typedef struct _PS_BOUND_NODE_
{
    MEON_STRING*ProtocolName;
    ODISTAT      (*ProtocolReceiveHandler)(LOOKAHEAD*);
    INFO_BLOCK   *ProtocolControlHandler;
    void         *ProtocolResourceObj;
} PS_BOUND_NODE;
```

Field Descriptions

ProtocolName

Pointer to a NULL terminated MEON string containing the short name for the protocol stack that is to receive the frames with the appropriate Protocol ID.

ProtocolReceiveHandler

Pointer the protocol stack's receive handler.

ProtocolControlHandler

Pointer to the protocol stack's information block for control handler routines.

ProtocolResourceObj

Pointer to a platform specific object used for resource management. This value is a pass through value and is not interpreted.

CLSL_ReSubmitDefault

Index 34 (0x22)

Allows default chained protocol stacks to pass received packets back to the LSL for further processing. The packets were originally queued and later processed at process time by a chained protocol stack. The LSL will pass these packets to the next protocol stack in the chain.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ReSubmitDefault (
    PS_CHAINED_RX_NODE *StackChainNode,
    LOOKAHEAD *LookAheadBuf );
```

Input Parameters

- StackChainNode*
Pointer provided by the LSL to the node structure, which defines this chained stack.
- LookAheadBuf*
Pointer to a LOOKAHEAD structure, which defines the received packet.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	<p>The protocol stack has returned a pointer to a receive ECB to be filled with the packet in the LOOKAHEAD structure's <i>LkAhd_ReturnedECB</i> field.</p> <p>If the protocol stack was able to get everything it needed from the LOOKAHEAD, the <i>LkAhd_ReturnedECB</i> field will be set to NULL, indicating that additional data does not need to be copied.</p>
ODISTAT_SUCCESS_TAKEN	<p>The protocol stack accepted the packet and has taken the prefilled LSL ECB associated with the LOOKAHEAD structure's <i>LkAhd_PreFilledECB</i> field.</p>
ODISTAT_OUT_OF_RESOURCES	<p>Reports an error condition—for example, the LSL was unable to obtain an ECB for this packet. The default stack that performs a <i>CLSL_ReSubmitDefault</i> function copies the contents of its LOOKAHEAD structure <i>LkAhd_FrameDataStartCopyOffset</i> field to the original callers LOOKAHEAD structure <i>LkAhd_FrameDataStartCopyOffset</i> field. This is because the LSL passes state information in the above field that it uses in the processing of this received packet.</p>

Remarks

The chained stack node passed to this routine is that of the chained stack calling this routine. The LSL calls the next appropriate stack in the chain to continue processing of this packet.

CLSL_ReSubmitPreScanRx

Index 35 (0x23)

Allows prescan receive chained protocol stacks to pass received packets back to the LSL for further processing. The packets were originally queued and later processed at process time by a chained protocol stack. The LSL will pass these packets to the next protocol stack in the chain.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ReSubmitPreScanRx (
    PS_CHAINED_RX_NODE *StackChainNode,
    LOOKAHEAD *LookAheadBuf );
```

Input Parameters

StackChainNode

Pointer provided by the LSL to the node structure that defines this chained stack.

LookAheadBuf

Pointer to a LOOKAHEAD structure that defines the received packet.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	<p>The protocol stack has returned a pointer to a receive ECB to be filled with the packet in the LOOKAHEAD structure's <i>LkAhd_ReturnedECB</i> field.</p> <p>If the protocol stack was able to get everything it needed from the LOOKAHEAD, the <i>LkAhd_ReturnedECB</i> field will be set to NULL, indicating that additional data does not need to be copied.</p>
ODISTAT_SUCCESS_TAKEN	<p>The protocol stack has accepted the packet and has taken the prefilled LSL ECB associated with the LOOKAHEAD structure's <i>LkAhd_PreFilledECB</i> field.</p>
ODISTAT_OUT_OF_RESOURCES	<p>Reports an error condition, such as the LSL was unable to obtain an ECB for this packet. The prescan receive stack that performs a CLSL_ReSubmitPreScanRx function copies the contents of its LOOKAHEAD structure <i>LkAhd_FrameDataStartCopyOffset</i> field to the original callers LOOKAHEAD structure <i>LkAhd_FrameDataStartCopyOffset</i> field, since the LSL will pass state information in the above field that it uses in the processing of this received packet.</p>

Remarks

The chained stack node passed to this routine is that of the chained stack calling this routine. The LSL calls the next appropriate stack in the chain to further process this packet.

CLSL_ReSubmitPreScanTx

Index 36 (0x24)

Allows prescan transmit chained protocol stacks to pass transmit ECBs back to the LSL for further processing. The ECBs were originally queued and later processed at process time by a prescan transmit chain stack. The LSL will pass these ECBs to the next protocol stack in the chain.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ReSubmitPreScanTx (
    PS_CHAINED_TX_NODE *StackChainNode,
    ECB *TransmitECB);
```

Input Parameters

StackChainNode

Pointer provided by the LSL to the node structure defining this chained stack.

TransmitECB

Pointer to a transmit ECB.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	Command executed successfully.
ODISTAT_BAD_PARAMETER	The board number defined by the node structure and pointed to by <i>StackChainNode</i> does not exist.

Remarks

The chained stack node passed to this routine is that of the chained stack calling this routine. The LSL will call the next appropriate stack in the chain to further process this packet.

CLSL_ReturnECB

Index 01 (0x01)

Enables a protocol stack to return a previously allocated LSL ECB buffer to the LSL's buffer pool.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ReturnECB (
    ECB *ReturnedECB );
```

Input Parameters

ReturnedECB

Pointer to an ECB to return to the LSL.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL The ECB was successfully returned to the LSL.

ODISTAT_BAD_PARAMETER The ECB did not originate from the LSL.
ER

CLSL_ScheduleAESEvent

Index 03 (0x03)

Schedules an asynchronous event scheduler (AES) event.

Syntax

```
#include <odi.h>

ODISTAT CLSL_ScheduleAESEvent (
    AES_ECB *TimerAESECB );
```

Input Parameters

TimerAESECB
Pointer to an AES ECB to be scheduled.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The specified AES event was scheduled.
ODISTAT_BAD_PARAMETER	The resource tag for the AES ECB was invalid.

Remarks

The defined event service routine (ESR) is called after the specified number of milliseconds. The ESR can reschedule itself after it resets *AES_MSecondValue*, thus creating a simple polling function.

An AES ECB that is already in use by the LSL AES system must not be passed again to **CLSL_ScheduleAESEvent**. To reset the AES event time, use

CLSL_CancelAESEvent and then issue a new **CLSL_ScheduleAESEvent** call.

On entry to the **AES_ESR** routine:

```
AES_ESR ( AES_ECB *TimerAESECB );
```

AES_ECB Structure

```
typedef struct _AES_ECB_
{
    struct _AES_ECB_ *AES_Link;
    UINT32             AES_MSecondValue;
    UINT16             AES_Status;
    void               (*AES_ESR)(struct _AES_ECB_
        *);
    UINT32             AES_Reserved;
    void               *AES_ResourceObj;
    void               *AES_Context;
} AES_ECB;
```

Field Descriptions

AES_Link

This field is used by the LSL for list management.

AES_MSecondValue

This field specifies the number of milliseconds to wait before invoking the defined **AES_ESR** routine. This field must be initialized each time the *AES_ECB* is passed to **CLSL_ScheduleAESEvent**.

AES_Status

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

AES_ESR

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

AES_Reserved

This field is reserved for use by the LSL.

AES_ResourceObj

Pointer to a platform specific object used for resource management. This is a pass through value and is not interpreted.

AES_Context

This optional field specifies a pointer to context parameters that can be passed to the AES ESR routine upon completion of the AES event.

CLSL_SendComplete

Index 14 (0x0E)

Must be called by the MLID every time a packet is transmitted with an ECB.

Syntax

```
#include <odi.h>

void CLSL_SendComplete (
    ECB *SendECB );
```

Input Parameters

SendECB

Pointer to an ECB associated with a completed send event.

Output Parameters

None.

Return Values

None.

Remarks


The MLID must call this routine any time the driver is finished using a send ECB (also called a Transmit Control Block). The MLID is usually done with the send ECB after it sends the data to the network interface card.

After making this call, the MLID must call **CLSL_ServiceEvents** to process the send.

The MLID sets the completion code in the *ECB_Status* field before making this call.

The completion codes for the *ECB_Status* field are as follows:

ODISTAT_SUCCESSFUL	The MLID determined that the transmit was successful. Because the transmit was connectionless, this completion code does not mean that the destination received the packet.
ODISTAT_MLID_SHUTDOWN	The MLID specified in the <i>ECB_BoardNumber</i> field cannot be found. This usually means that the MLID has been removed from memory or is shut down (temporarily or permanently).
ODISTAT_BAD_PARAMETER	The ECB contains bad parameters—for example, the amount of data to transmit exceeds the maximum possible for the MLID. Note, the ECB will not have been transmitted.
ODISTAT_CANCELED	The ECB is being returned without being transmitted. This usually occurs if the ECB was held in an MLID's queues, then the MLID clears its queues due to a shut down request. This can also occur if the MLID was unable to transmit the packet.

Note  The ODISTAT type is cast to a UINT16 for the *ECB_Status* field.

See Also

CLSL_SendPacket

CLSL_SendPacket

Index 12 (0x0C)

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

Syntax

```
#include <odi.h>

ODISTAT CLSL_SendPacket (
    ECB *SendECB );
```

Input Parameters

SendECB
Pointer to an ECB to be sent.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The ECB has been handed to the MLID.
ODISTAT_ITEM_NOT_PRESENT	The board number in the <i>ECB_BoardNumber</i> field does not correspond to a registered MLID.
ODISTAT_FAIL	The ECB was invalid or already in use elsewhere in the system.

Remarks

When the MLID has transmitted the packet and is finished with the ECB, it calls **CLSL_SendComplete** to return the ECB.

If the MLID calls **CLSL_SendComplete** before this function returns, the ECB's ESR ((*ECB_ESR)(ECB*)) can be invoked.

See Appendix A, "Event Control Blocks (ECBs)" for the ECB requirements.

See Also

CLSL_SendComplete

CLSL_SendProtocolInfoToPartner

Index 44 (0x2C) NetWare Server only

Called (when using SFTIII) whenever protocol information needs to be sent to the other IOEngine, such as after a bind or unbind operation.


Syntax

```
#include <odi.h>

SFTIII_STAT CLSL_SendProtocolInfoToPartner (
    UINT32 ProtocolNumber,
    UINT8 *ProtocolInfo,
    UINT32 Length,
    void ( *InfoSendCallBack )
    ( UINT32 Reserved, UINT8 *ProtocolInfo
      ) );
```

Input Parameters

- ProtocolNumber*
The Stack ID (SID).
 - ProtocolInfo*
Pointer to information to be sent.
 - Length*
Number of bytes (UINT8) pointed to by *ProtocolInfo*.
 - InfoSendCallBack*
Pointer to a function that is called when the information pointed to by *ProtocolInfo* has been sent.
- Note



Information sent does not imply that the destination received the information.

Output Parameters

None.

Return Values

SFTIII_STAT_SUCCESSFUL	Operation completed successfully.
SFTIII_STAT_MIRROR_NOT_ACTIVE	Engine not mirrored.
SFTIII_STAT_NO_PARTNER	Engine does not have a partner.
SFTIII_STAT_NOT_SUPPORTED	Function not supported.

Remarks

The operating system will signal your protocol stack at certain times to give all needed protocol information to the other IOEngine.

CLSL_SendProtocolInfoToOtherEngine

Index 45 (0x2D) NetWare Server only

Called (when using SFTIII) whenever protocol information needs to be sent to the other engine, such as after a bind or unbind operation.

Syntax

```
#include <odi.h>

SFTIII_STAT CLSL_SendProtocolInfoToOtherEngine (
    UINT32  ProtocolNumber,
    UINT8   *ProtocolInfo,
    UINT32  Length,
    void ( *InfoSendCallBack ) ( UINT8
        *ProtocolInfo ) );
```

Input Parameters

ProtocolNumber

The Stack ID (SID).

ProtocolInfo

Pointer to information to be sent.

Length

Number of bytes (UINT8) pointed to by *ProtocolInfo*.

InfoSendCallBack

Pointer to a function to be called when the information pointed to by *ProtocolInfo* has been sent.

Output Parameters

None.

Return Values

SFTIII_STAT_SUCCESSFUL	Operation completed successfully. Because the transmit was connectionless, this completion code does not mean that the destination received the packet.
SFTIII_STAT_MIRROR_NOT_ACTIVE	Mirrored server engine not active.
SFTIII_STAT_NO_PARTNER	Engine does not have a partner.
SFTIII_STAT_OUT_OF_RESOURCES	No memory available to queue request.
SFTIII_STAT_NOT_SUPPORTED	Function not supported.

Remarks

The operating system signals your protocol stack at certain times to give all needed protocol information to the other engine.

Note



CLSL_SendProtocolInfoToOtherEngine and *CLSL_SendProtocolInfoToPartner* are available in all three server types (IOEngine, MEngine, and server). If this API is unsupported, the engine will return with all bits set.

CLSL_ServiceEvents

Index 41 (0x29)

Causes the LSL to service any events placed on its hold queue by **CLSL_HoldEvent**.

Syntax

```
#include <odi.h>

void CLSL_ServiceEvents ( void );
```

Input Parameters

None.

Output Parameters

None.

Return Values

None.

Remarks

CLSL_ServiceEvents processes all of the ECBs on its hold queue by calling the ESR ((*ECB_ESR)(ECB*)) routine for each ECB, as defined in the ECB's *ECB_ESR* field. These ESR routines must not poll for transmit or receive events.

CLSL_UnbindStack

Index 22 (0x16)

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

Syntax

```
#include <odi.h>

ODISTAT CLSL_UnbindStack (
    UINT32  ProtocolNumber,
    UINT32  BoardNumber );
```

Input Parameters

ProtocolNumber
The Stack ID (SID).

BoardNumber
The board number.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The protocol stack was unbound from an adapter and frame type (logical board) combination.
ODISTAT_BAD_PARAMETER	The specified SID or the board number is invalid.
ODISTAT_ITEM_NOT_PRESENT	The specified binding does not exist.

Remarks

After this routine successfully returns, packet reception between the specified protocol stack and logical board is disabled (unless the protocol stack has an outstanding **CLSL_RegisterPrescanChain** or **CLSL_RegisterDefaultChain**).

10-94 ODI Specification: Protocol Stacks and MLIDs (C Language)

chapter 10 *Overview of the MLID*

Chapter Overview

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

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



Note

As an alternative to writing a complete MLID, you may want to write a C language Hardware Specific Module (CHSM) by getting the *LAN Driver Developer's Guide* kit. This kit provides many of the pieces of the MLID and only requires you to write the HSM, which, depending upon your needs, may be easier to do than writing a complete MLID.

Multiple Operating System Support

MLID development depends on the operating system under which the MLID will run. An MLID developed to the NetWare operating system must be developed differently than an MLID developed to the DOS, OS/2, or Windows NT operating systems.

If you wish to develop a single MLID that runs under multiple operating systems, we strongly recommend that (instead of writing an MLID) you develop a C language Hardware Specific Module (CHSM) using the *LAN Driver Developer's Guide* kit to help you.

Overview of the MLID **11-1**

NetWare MLID

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

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

MLID Procedures

The ODI specification defines the following procedures:

- MLID initialization routine (required)
- Board service routine (one or both are required)
 - Interrupt Service Routine (ISR)
 - Driver polling routine
- Packet transmission routine(required)

The MLID also supports the following control procedures:

- Control procedures for ODI IOCTLs
 - AddMulticastAddress (required if hardware supports multicast addressing)
 - DeleteMulticastAddress (required if hardware supports multicast addressing)
 - GetMLIDConfiguration (required)
 - GetMLIDStatistics (required)
 - DriverPromiscuousChange(recommended)
 - SetLookAheadSize (required)
 - DriverManagement (optional)
 - MLIDReset (required)
 - MLIDShutdown (required)

11-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

- GetMulticastInfo (required if hardware supports multicast addressing)
- RegisterMonitor (required)
- RemoveNetworkInterface (required)
- ShutdownNetworkInterface (required)
- ResetNetworkInterface (required)
- Timeout detection (some LAN adapters do not need to provide these procedures)
 - Interrupt call back routine (optional)
 - AES call back routine (optional)
- MLID removal routine (required)

The specific hardware requirements of a LAN adapter can require that you write additional procedures; however, the procedures listed above represent the generic code elements found in every MLID.

A brief description of each procedure is presented throughout the rest of this chapter. These descriptions are high-level generalizations only and are not true in every case, nor do they describe every possible case. More detailed descriptions of each procedure is provided in chapters 12 through 15 of this specification.

MLID Initialization

In general terms, the MLID's initialization routine must perform the following actions:

- Allocate memory for the MLID's variables and structures.
- Parse the standard LOAD command line options.
- Process custom command line parameters and custom firmware.
- Register the hardware configuration with the operating system.
- Initialize the LAN adapter.
- Register the MLID with the LSL.

Overview of the MLID 11-3

- Provide a hook for the MLID's board service routine by allocating an interrupt or by establishing a polling procedure.
- Schedule callback events for timeout detection and recovery.

Board Service Routine

The board service routine generally needs to detect and handle the following events on the LAN adapter:

- Received packet
- Packet receiving error
- Completed transmission
- Packet transmission error

The MLID can be notified of these events by an interrupt service routine (ISR), a polling procedure, or a polling procedure with interrupt backup.

Packet Transmission

The MLID's packet transmission routine is called whenever a packet needs to be transmitted onto the wire. The MLID must build the necessary frame and media headers and then send the packet.

Note



The MLID's transmit handler is passed both the transmit ECB and a pointer to its send context, which it provided to the LSL when it registered. For more information, see **CLSL_RegisterMLID** in *Chapter 10: LSL Support Routines*.

Control Routines

Among the control procedures that the MLID must provide are control procedures to support multicast addressing (if the hardware supports it) and procedures to reset and shut down the hardware. The MLID can also supply a control procedure to support promiscuous mode.

MLIDs that support the hub management interface implement the driver management support routine.

11-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

Timeout Detection

The MLID can schedule an AES event that it uses at specified intervals. For example, the MLID might need to be called regularly to inspect the LAN adapter and determine if the adapter has failed to complete a transmission. If a timeout error had occurred, the procedure discards the packet being sent, resets the board, and begins transmitting the next packet in the send queue.

Driver Remove

Every MLID must have a remove procedure that allows the user to unload the MLID from the operating system. This procedure must shut down the LAN adapter and return any resources that the MLID allocated from the operating system.

MLID Data Structures and Variables

In addition to the procedures discussed above, the MLID must also contain certain data structures and variables. The following are the primary structures:

- MLID configuration table
- MLID statistics table

MLID Configuration Table

The MLID configuration table is a data structure that defines the configuration of the LAN adapter and MLID. The fields in this table are referred to by the LSL, Protocol Stack, MLID, and other components in the system. The requirements for MLID configuration tables are described in detail in Chapter 12, "MLID Data Structures".

MLID Statistics Table

The MLID statistics table is a data structure that contains data on the operation of the LAN adapter and the MLID. Chapter 12, "MLID Data Structures" contains a detailed description of this data structure.

Overview of the MLID 11-5

Network management and other components in the system reference the information in the MLID statistics table via the **GetMLIDStatistics** control procedure.

MLID Functionality

We strongly recommend that your MLID provides the following functionality.

- Reentrancy
- Multiple frame support (supports all frame types defined by a specific topology)
- Source routing support when supported by the topology
- Promiscuous mode support
- Multicast addressing support

Note



In some instances this specification makes recommendations on how to implement certain functionality, but these are only recommendations and it is up to you to implement the functionality the way you choose.

Reentrancy

We strongly recommend that your MLID support reentrancy. When you link your LAN driver, you can declare your driver reentrant. This allows the operating system to use a single code image of the MLID to run multiple LAN adapters (of the same type) or to run multiple frame types (logical boards) on the same LAN adapter. A non-reentrant driver requires the operating system to load an additional code image of the driver each time it uses another LAN adapter or supports another logical board.

To illustrate the advantage of reentrant code, consider the following example. Suppose you want to configure a server to drive two Novell CNTR2000 cards. You enter the following commands at the server console:

```
load cntr2000
load cntr2000
```

If you have written reentrant code, the first **load** command loads the code image of the driver into the server's memory and then calls the MLID's initialization routine. The second **load** command merely calls the MLID's initialization routine again. If you have not written reentrant code, two copies of the CNTR2000 LAN driver are loaded into memory.

Multiple Frame Support

If the LAN adapter runs on a topology that supports multiple frame types, we strongly recommend that the MLID support all the frame types for that particular topology. You can implement multiple frame support by using logical boards.

Multiple Frame Support and Logical Boards

To illustrate how logical boards are used, consider the preceding example of loading a CNTR2000 twice. When you enter the **load** command the second time, you are indicating one of two things:

- You want the MLID to run a second LAN adapter.
- You want the MLID to run a second frame type on the LAN adapter that is already loaded.

Whichever is the case, your MLID creates a "logical board" in response to this command. (A fuller description of logical boards is provided below.) The operating system does not concern itself with distinguishing between logical boards that have exclusive use of a LAN adapter and logical boards that share the same LAN adapter with other logical boards. Only the MLID makes this distinction.

Multiple Frame Support in Reentrant Code

If you are writing reentrant code, each logical board uses the same code image of the MLID that was loaded into the operating system with the first **load**

Overview of the MLID 11-7

command. However, the MLID must maintain a separate adapter data space for each physical board or each separate channel (see the MLID configuration table for more information) and a separate frame data space for each logical board.

Adapter Data Space

When a second load command is issued, an ambiguous situation arises. The MLID resolves this ambiguity by asking the following question:

Do you want to add another frame type for a previously loaded board?

If your response to the operating system's question is no, the MLID must allocate an adapter data space to drive a second adapter. The adapter data space is a structure that contains the hardware specific information that the MLID needs to drive the LAN adapter (interrupt number, beginning memory address, etc.). The statistics table required by the ODI specification is contained in this adapter data space. The MLID allocates one adapter data space for each LAN adapter, regardless of the number of logical boards (frame types) it supports.

Note



The MLID must create an adapter data space for every LAN adapter of the same type that is loaded.

Frame Data Space

Every logical board has a frame data space associated with it. The frame data space is a structure that contains the frame-specific information the MLID needs to support that frame type. The MLID allocates a frame data space for each logical board. The MLID then copies the configuration table template for that logical board into its frame data space.

Note



The MLID must create a frame data space for every frame type that is loaded.

Implementing Multiple Frame Support

Figure 1.1 illustrates how you might implement multiple frame support in a CNE2000 driver. In order to use the first CNE2000 adapter, you would enter:

```
load cne2000
```

In response to this command, the MLID creates logical board 1, which uses Frame Data Space 1 and Adapter Data Space A to run Adapter A. By default,

Frame Data Space 1 contains the information necessary to support Ethernet 802.2.

Now suppose you wanted to use a second CNE2000 adapter that supported both SNAP and 802.2 frames. You start by entering the following command:

```
load cne2000 frame=ETHERNET_SNAP
```

Afterwards, you are asked the following question:

Do you want to add another frame type for a previously loaded board?

In order to use the second CNE2000 adapter, you need to enter **n**. This causes the MLID to create Logical Board 2, which uses Frame Data Space 2 and Adapter Data Space B to run Adapter B. The following command then tells the MLID that Frame Data Space 2 will support Ethernet SNAP:

```
frame=ETHERNET_SNAP
```

In order for Adapter B to also support 802.2, you need to load the CNE2000 driver a third time:

```
load cne2000 frame=ETHERNET_802.2
```

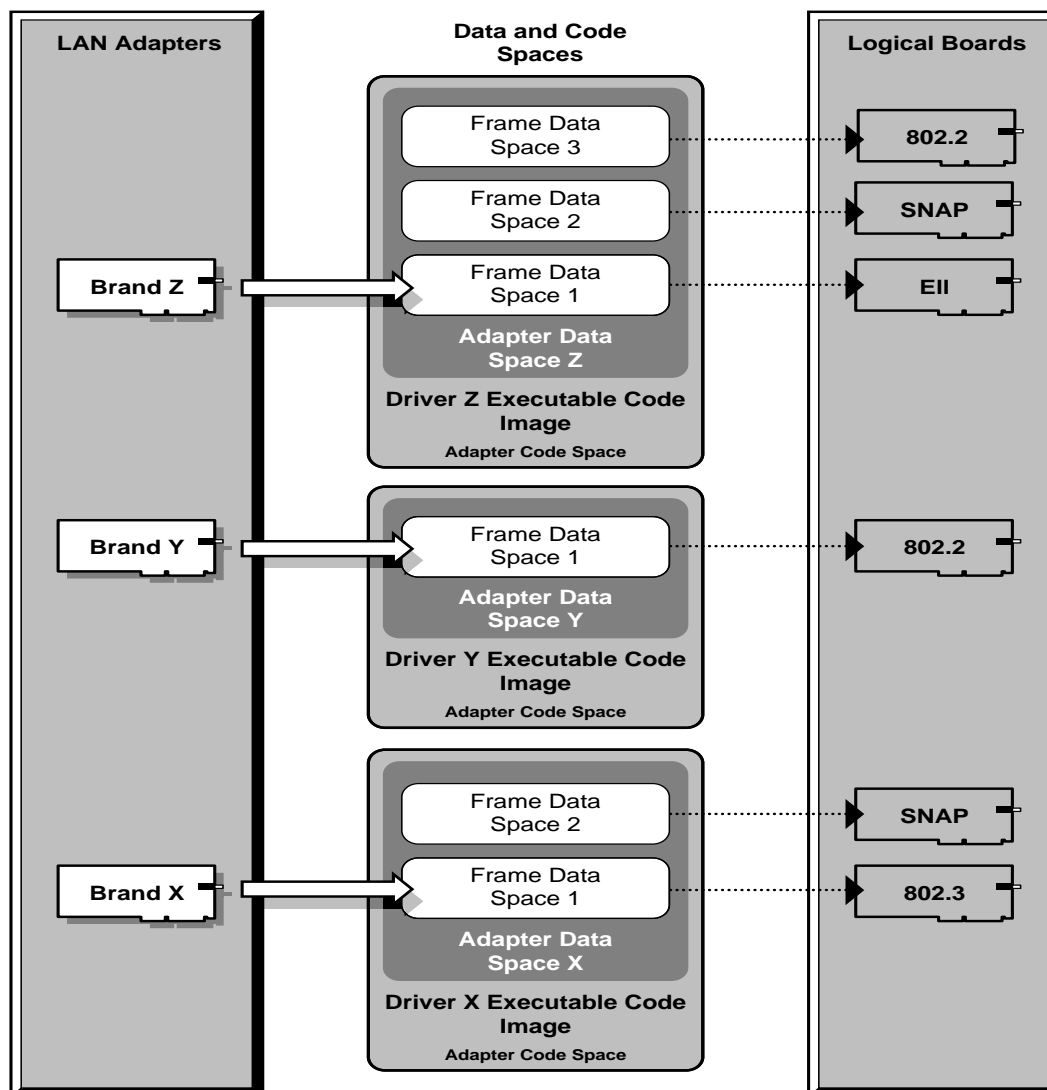
This time, however, you enter **y** in response to the following question:

Do you want to add another frame type for a previously loaded board?

The operating system then lets you indicate the LAN adapter that you want to add additional frame support to. If you were to specify Adapter B, the MLID would create logical board 3, which uses Frame Data Space 3 and Adapter Data Space B to communicate with Adapter B.

Overview of the MLID 11-9

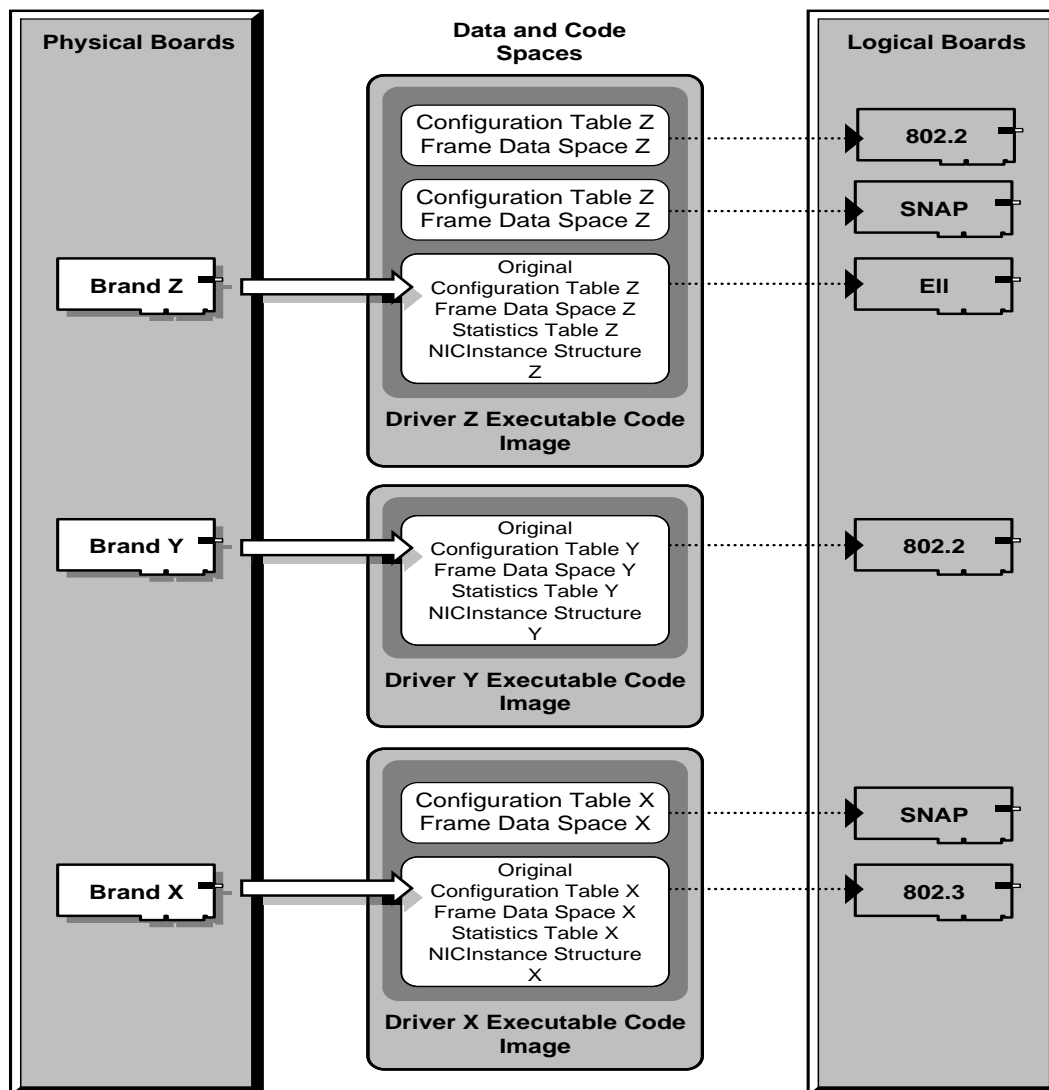
Figure 11-1
Implementations of
Multiple Frame Support Using Ethernet



11-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

Figure 1.2 shows that when the boards are not all the same type, each board has its own executable code image and adapter data space.

Figure 11-2
Implementation of
Multiple Boards/Frame Support



Overview of the MLID 11-11

Other Functionality

An MLID can support source routing (Token-Ring and FDDI topologies only, not Ethernet), promiscuous mode, and multicast addressing. We recommend that your MLID support all of these options, if the LAN adapter is capable of supporting them.

Source Routing Support

The *ODI Specification Supplement: Source Routing* describes how to add and configure source routing in the MLID.

Promiscuous Mode Support

When MLIDs operate in promiscuous mode, they pass all packets they receive to the upper layers. This includes bad packets, if possible. Because various monitoring functions operate in promiscuous mode, we strongly recommend that your MLID support promiscuous mode if your adapter is capable of such support. The MLID enables or disables promiscuous mode upon request by using the **PromiscuousChange** control routine described in Chapter 15, "MLID Control Routines".

Multicast Addressing Support

If your LAN adapter is capable of supporting multicast addressing, your MLID must support it. The **AddMulticastAddress**, **GetMulticastInfo**, and **DeleteMulticastAddress** control routines implement multicast support. These control procedures are discussed in more detail in Chapter 15, "MLID Control Routines".

MLID Design Considerations

The following section discusses hardware and coding issues you must consider when developing the MLID.

Hardware Issues

Every type of LAN adapter, such as the CNTR2000 and CNE2000, have different hardware and data transfer characteristics. A thorough understanding of your LAN adapter and LAN topology will allow you to create a more efficient driver. Keep in mind that the board and chip manufacturer's support

11-12 ODI Specification: Protocol Stacks and MLIDs (C Language)

engineers can provide you with up-to-date information regarding their hardware.

Data Transfer Mode

The LAN adapter's mode of data transfer is a primary consideration in MLID development. To achieve the highest performance, you must select support procedures matched to the data transfer mode. The data transfer modes are:

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

Bus Type

You must also consider the LAN adapter's bus type and size. The MLID's initialization process must register its bus type with the LSL. The following are common bus types:

- Industry Standard Architecture (ISA)
- Micro Channel Architecture
- Extended Industry Standard Architecture (EISA)
- Personal Computer Memory Card International Association (PCMCIA)
- Peripheral Component Interconnect (PCI)

Overview of the MLID 11-13

11-14 ODI Specification: Protocol Stacks and MLIDs (C Language)

chapter **11** *MLID Data Structures*

Chapter Overview

This chapter describes the data structures and variables that the MLID must define. All the data structures defined in this chapter must be present in the OSDATA segment of the MLID.

Frame Data Space


The ODI specification requires that every MLID have a configuration table as part of the frame data space. The MLID keeps a copy of the configuration table template in the OSDATA segment. The MLID uses the configuration table in the OSDATA segment as the working configuration table for the default logical board and as a template for the configuration tables it must copy for each loaded logical board. When the MLID allocates the frame data space for each logical board (frame type) that loads, it copies the configuration table template for that logical board into that logical board's frame data space. Because external processes can also access this table, the ODI specification defines this table's format strictly.

MLID Configuration Table


The MLID configuration table contains information about the MLID and its configuration. The MLID must define one configuration structure for each logical board number assigned by the LSL. Variables in this structure include the interrupt number, port I/O address, node address, and other MLID specific parameters.

The MLID must define the configuration table to contain the LAN adapter's default configuration and any other information about that configuration. The table must be defined by the fields described in this chapter, with each entry filled in accordingly. Certain variables in the configuration table are specific to your MLID. Other variables are specific to the LAN adapter the MLID is running.

- Note



All data strings in the configuration table consist of NULL terminated MEON strings.
- Note



Protocol stacks and other system modules treat the MLID configuration table as read only!

MLID Configuration Table Structure Sample Code

```
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_IORangel;
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;

```

MLID Data Structures 12-3

```
        UINT8                MLIDCFG_DIOConfigMinorVer;  
    }    MLID_CONFIG_TABLE
```

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_Signature	MEON [26]	This field contains a string that indicates the start of the configuration table. The string is “HardwareDriverMLID” followed by exactly eight spaces. It must be included in the table.
MLIDCFG_MajorVersion	UINT8	This field must be set to the major version number of the configuration table. The current major version number is 1.
MLIDCFG_MinorVersion	UINT8	This field must be set to the minor version number of the configuration table. The current minor version number is 21.
MLIDCFG_NodeAddress	NODE_ADDR	This field holds the card's node address. The MLID sets this field during its initialization routine. (See <i>ODI Specification Supplement: Canonical and Noncanonical Addressing</i> for information regarding octet bit reversal.)
MLIDCFG_ModeFlags	UINT16	See <i>MLIDCFG_ModeFlags</i> field description (Table 12-2). Unused bits are reserved and set to 0.
MLIDCFG_BoardNumber	UINT16	During initialization, the MLID sets this field to the board number that is returned by the CLSLRegisterMLID .
MLIDCFG_BoardInstance	UINT16	The MLID sets this field to the physical board instance. For example, if two CNE2000 boards were installed in the system, the first CNE2000 driver loaded would have this field set to 1; the second CNE2000 driver would have this field set to 2. Note: Each controller on a multi-channel adapter is treated as a separate adapter if access to the controller is independent of the other controllers.

12-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_MaxFrameSize	UINT32	<p>This value defines the largest possible packet size that can be transmitted and/or received by the driver and physical card combination. This value includes all headers.</p> <p>Ethernet drivers set this field to 1514 decimal. Since Token-Ring drivers can send and receive a number of different packet sizes, a Token-Ring driver must determine during its <i>DriverInit</i> routine the appropriate packet size and place that value in this field. Token-Ring drivers support 4K (4096+74+48 = 4218) packet sizes whenever it is possible and practical. The value in this field cannot be less than 618 decimal. See Table 12-5 for more details.</p>
MLIDCFG_BestDataSize	UINT32	<p>The MLID sets this field after returning from <i>DriverInit</i>. The MLID 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 MLID sets this field to 1500 decimal (1514 - 14 [MAC] = 1500) if the MLID is running the Ethernet_II packet type. If a Token-Ring MLID sets this field, it sets this field to <i>MLIDCFG_MaxFrameSize</i> - 14 [MAC] - 3 [802.2 UI] if the MLID's packet type is Token-Ring. See Table 12-5 for more details.</p>
MLIDCFG_WorstDataSize	UINT32	<p>The MLID sets this field after returning from <i>DriverInit</i>. The MLID subtracts the length of the largest media headers(s) from the <i>MLIDCFG_MaxFrameSize</i> field.</p> <p>Note, protocol stacks use the value in this field to determine the largest packet size this driver can send or receive.</p> <p>For example, of a Token-Ring MLID sets this field, it sets this field to <i>MLIDCFG_MaxFrameSize</i> - 14 [MAC] - 30 [source routing] - 4 [802.2 UI] if the MLID's frame type is Token-Ring. An Ethernet_II MLID sets this field to 1500. See Table 12-5 for more details.</p>

MLID Data Structures **12-5**

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_CardName	MEON_STRING *	This field holds a pointer to a NULL terminated MEON string that is identical to the description string in the linker definition file. For example, "Novell Ethernet NE2000". (See Appendix C, Platform Specific Information).
MLIDCFG_ShortName	MEON_STRING *	This field holds a pointer to a NULL terminated MEON string that contains a shortened version of the long name. This string cannot contain more than 8 MEON characters. The string is usually the MLID's file name. For example, "CNE2000".
MLIDCFG_FrameTypeString	MEON_STRING *	This field holds a pointer to a NULL terminated MEON string describing 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.)
MLIDCFG_Reserved0	UINT16	This field is reserved for future use and must be set to 0.
MLIDCFG_FrameID	UINT16	<p>This field contains the frame type ID being used by this MLID.</p> <p>For more information on frame types, see <i>ODI Specification Supplement: Frame Types and Protocol IDs</i>.</p>
MLIDCFG_TransportTime	UINT16	<p>This field indicates the number of milliseconds it takes the adapter to transmit a 586-byte packet. Most MLIDs set this field to 1. This field cannot be set to 0.</p> <p>If the MLID is used with a slow asynchronous line, the value is set accordingly to a representative value.</p>

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description															
MLIDCFG_SourceRouting	UINT32 (*) (UINT32, void *, void **, boolean)	<p>This field contains a pointer used by a Token-Ring or FDDI MLID and a source routing module, such as SROUTE.NLM. See the <i>ODI Specification Supplement: Source Routing</i> for a discussion of dynamic source routing.</p> <p>Note: This function should use the ANSI C calling convention.</p>															
MLIDCFG_LineSpeed	UINT16	<p>This field holds the data rate used by the physical card's media. 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.</p> <p>If the line speed can be selected, as with Token-Ring, the MLID must determine the selected line speed and place that value in this field. Below are some common values:</p> <table> <tr> <td>Ethernet</td><td>10 Mbps</td><td>0x000A</td></tr> <tr> <td>Token-Ring</td><td>4 Mbps</td><td>0x0004</td></tr> <tr> <td>Token-Ring</td><td>16 Mbps</td><td>0x0010</td></tr> <tr> <td>FDDI</td><td>100 Mbps</td><td>0x0064</td></tr> <tr> <td>ISDN</td><td>64 Kbps</td><td>0x8040</td></tr> </table> <p>For example, if the speed of the line MLID is 10 Mbps (Ethernet for example) put 10 (decimal) in this field.</p>	Ethernet	10 Mbps	0x000A	Token-Ring	4 Mbps	0x0004	Token-Ring	16 Mbps	0x0010	FDDI	100 Mbps	0x0064	ISDN	64 Kbps	0x8040
Ethernet	10 Mbps	0x000A															
Token-Ring	4 Mbps	0x0004															
Token-Ring	16 Mbps	0x0010															
FDDI	100 Mbps	0x0064															
ISDN	64 Kbps	0x8040															

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_LookAheadSize	UINT16	<p>This field holds the configured lookahead size as set by protocol stacks. The MLID sets this to a default value of 18 bytes. However, a protocol stack can dynamically override this value using the SetLookAheadSize MLID control function. The maximum value is 128 bytes when receiving a packet, the MLID uses this value and the maximum possible media header when determining the amount of lookahead data the MLID must pass to the <i>CLSL_GetStackECB</i> routine for every packet that the MLID receives. The value in this field can be changed at any time. Therefore, the MLID must reference this field for every packet that the MLID receives.</p> <p>Note: Once set, this value never decreases. See SetLookAheadSize in Chapter 15 of this document.</p>
MLIDCFG_SGCount	UINT8	<p>The maximum number of Scatter/Gather elements that the adapter is capable of handling. This is only valid if the MM_FRAGS_PHYS_BIT in MLIDCFG_ModeFlags is set by the MLID. The minimum value for this field is 2 (1 MAC header fragment and 1 data header fragment). The maximum value for this field is 17 (1 MAC header fragment and 16 ECB data fragments).</p>
MLIDCFG_Reserved1	UINT8	<p>This field is reserved for future use and must be set to 0.</p>
MLIDCFG_PrioritySup	UINT16	<p>This field contains the number of priority levels that the MLID supports. This field has a maximum of 7 priorities (1-7). Zero (0) indicates a non-priority packet. The MLID can set this field to 0 through 7. Seven (7) is the highest priority.</p>
MLIDCFG_Reserved2	void *	<p>This field is reserved for future use and must be set to 0.</p>

12-8 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_DriverMajorVer	UINT8	This field defines the current revision level of the MLID and matches the revision level found in the linker definition file and displayed by the MLID. For example, if the MLID's current major version is 2, this field's value is 2.
MLIDCFG_DriverMinorVer	UINT8	This field defines the current revision level of the MLID and matches the revision level found in the linker definition file and displayed by the MLID. For example, if the MLID's current minor version is .32, this field's value is 32. (If the current major and minor version level displayed by the MLID is 2.32, these fields reflect that version of 2.32.)
MLIDCFG_Flags	UINT16	See the <i>MLIDCFG_Flags</i> field description (Table 12-3). This field is set by the MLID and reflects whether the MLID supports hub management and whether the adapter has specialized hardware to support group addressing—for example, CAM. Unused bits must be set to 0. See Table 12-3 for more information.
MLIDCFG_SendRetries	UINT16	This field is initialized by the MLID to an appropriate value that represents the number of times the MLID will retry a transmission operation with an error before giving up. See the "RETRIES Load" key word in Appendix C, 'Platform Specific Information'.
MLIDCFG_DriverLink	void *	This field is set to NULL and is not modified by the MLID.

MLID Data Structures 12-9

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_SharingFlags	UINT16	<p>The MLID sets this variable. See the <i>MLIDCFG_SharingFlags</i> field description (Table 12-4).</p> <p>This field informs the system the hardware resources that a driver/physical card can share with other driver/physical cards. If the MLID supports shareable interrupts, the MLID must set the <i>MS_SHARE_IRQx_BIT</i> bit. The first bit is used to indicate when the MLID is shutdown. The MLID is responsible for setting and clearing this bit. The bit definitions for this field are listed in Table 12-4.</p>
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 GetInstanceNumber after calling SearchAdapter. This value normally corresponds to the number silk screened on the mother board or stamped on the chassis of the compter. The instances are assinged a unique value in the following cases:</p> <ul style="list-style-type: none">Integrated Motherboard DevicesPCI BIOS v2.0 DevicesPCI BIOS v2.1 Adapter<ul style="list-style-type: none">(with Multiple Devices or Functions)PnP ISA DevicesConfilcts 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 default I/O port base address. The user can override this value input from a configuration entry that is operating system dependent. If the MLID is self-configurable, it determines the appropriate value for the physical card and place that value into this field before before registering the hardware options and returning from initialization. If the MLID does not use I/O ports, this field is set to NULL.</p>

12-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_IORange0	UINT16	This field defines the number of UINT8 I/O ports decoded by the physical card at <i>MLIDCFG_IOPort0</i> . Set this field to <i>UNUSED_IO_RANGE</i> if the physical card does not use I/O ports.
MLIDCFG_IOPort1	UINT16	This field allows the MLID to define two I/O port base addresses. The definition is the same as <i>MLIDCFG_IOPort0</i> . Set this to <i>UNUSED_IO_PORT</i> if the physical card does not have a second range of I/O ports.
MLIDCFG_IORange1	UINT16	The number of UINT8 I/O ports starting at <i>MLIDCFG_IOPort1</i> . If this field is not used, set it to <i>UNUSED_IO_RANGE</i> .
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 MLID sets this variable, but it can be changed by the configuration.
MLIDCFG_MemorySize0	UINT16	<p>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>.</p> <p>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.</p>

MLID Data Structures 12-11

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_MemoryAddress1	void *	This field allows the MLID to define a second memory address range used by the MLID's adapter. For example, <i>MLIDCFG_MemoryAddress1</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 <i>UNUSED_MEMORY_ADDRESS</i> if the adapter does not define a second memory range.
MLIDCFG_MemorySize1	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_MemoryAddress1</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_MemoryAddress1</i> . If <i>MLIDCFG_MemoryAddress1</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.
MLIDCFG_Interrupt0	UINT8	Primary interrupt vector number. This field is initialized to the physical adapter's default interrupt request line (IRQ). If the adapter does not use an interrupt line, set this field to <i>UNUSED_INTERRUPT</i> . If the MLID's adapter supports IRQ 2 or 9, the MLID sets the value to be consistent with the adapter's documentation. This field is set to the adapter's default base interrupt vector number. 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	Secondary interrupt vector number. This field is set to the adapter's second interrupt vector number. Set this field to <i>UNUSED_INTERRUPT</i> if it is not used.

12-12 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_DMALine0	UINT8	This field is 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> .
MLIDCFG_DMALine1	UINT8	This field is 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 used.
MLIDCFG_ResourceTag	void *	This field contains a pointer to a resource tag.
MLIDCFG_Config	void *	This field contains a pointer to the LSL's copy of the configuration table. The MLID does not use this field.
MLIDCFG_CommandString	void *	Pointer to a structure containing two fields. The first field is a forward link to the next structure. 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 will be the concatenation of all of them. Bits 9 and 10 of the <i>MLID_SharingFlags</i> field are used in conjunction with this field. The MLID sets this field.
MLIDCFG_LogicalName[18]	MEON_STRING	MLIDs do not use this field. It contains the NULL terminated logical name of the LAN MLID if a name exists.
MLIDCFG_LinearMemory0	void *	<p>The operating system fills in this field with the linear address of <i>MLIDCFG_MemoryAddress0</i> during the MLID's initialization routine.</p> <p>Do not convert <i>MLIDCFG_MemoryAddress0</i> to the logical address using the operating system conversion routines.</p>

MLID Data Structures **12-13**

Table 12-1
MLID Configuration Table Field Descriptions

Name	Type	Description
MLIDCFG_LinearMemory1	void *	<p>The operating system fills in this field with the linear address of <i>MLIDCFG_MemoryAddress1</i> during the MLID's initialization routine.</p> <p>Do not use the operating system conversion routines to convert <i>MLIDCFG_MemoryAddress1</i> to the logical address.</p>
MLIDCFG_ChannelNumber	UINT16	This field is used in multichannel adapters. It holds the channel number of the NIC to use. Set this field to 0 if multichannel adapters are not in use.
MLIDCFG_DBusTag	void *	<p>Pointer to an architecture dependent value which specifies the bus on which the adapter is found. The MLID must enter the value returned by SearchAdapter in this field.</p>
MLIDCFG_DIOConfigMajorVer	UINT8	The major version of the IO_CONFIG_TABLE structure (the bottom half of the MLID_CONFIG_TABLE structure). The MLID sets this field to 1.
MLIDCFG_DIOConfigMinorVer	UINT8	The minor version of the IO_CONFIG_TABLE structure (the bottom half of the MLID_CONFIG_TABLE structure). The MLID sets this field to 0.

MLIDCFG_ModeFlags Field

Replace with new sentence.

Figure 12-1
MLIDCFG_ModeFlags Field Default Values

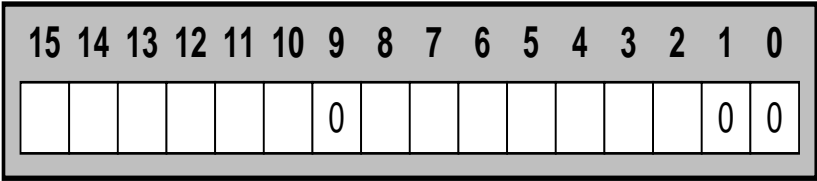


Table 12-2 explains.....

Table 12-2
MLIDCFG_ModeFlags Bits Description

Bit #	Name	Description
0	Reserved	This bit is reserved and must be set to zero.
1	Reserved	This bit is reserved and must be set to zero.
2	MM_DEPENDABLE_BIT	The function of this bit has been rendered obsolete by recent changes to NetWare (specifically, the NetWare Link Services Protocol). Therefore, 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 MLID, 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	Set this bit if the MLID and physical board support multicasting. Multicast support is required for all media that have multicast capability.

Table 12-2
MLIDCFG_ModeFlags Bits Description

Bit #	Name	Description
4	MM_CSL_COMPLIANT_BIT	The MLID 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, etc.) 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 interface, see the <i>WAN ODI Specification</i> .
5	MM_PREFILLED_ECB_BIT	Set this bit if the MLID always supplies prefilled (LSL) ECBs in the LOOKAHEAD structure's <i>LkAhd_PreFilledECB</i> field.
6	MM_RAW_SENDS_BIT	The MLID sets this bit to 1 if it supports raw send.
7	MM_DATA_SZ_UNKNOWN_BIT	Set this bit if the MLID is capable of setting the <i>LkAhd_FrameDataSize</i> field in the LOOKAHEAD structure to a -1, frame size and/or receive status unknown—for example, pipelined LAN adapter.
8	MM_SMP_BIT	Set this bit if the MLID supports symmetrical multiprocessing.
9	Reserved	Reserved.
10	MM_FRAG_RECEIVES_BIT	Set this bit to zero. This field is used only if the MLID was developed with Novell's <i>LAN Driver Developer's Kit for HSMs</i> .
11	MM_C_HSM_BIT	Set this bit to zero. This field is used only if the MLID was developed with Novell's <i>LAN Driver Developer's Kit for HSMs</i> .
12	MM_FRAGS_PHYS_BIT	Set this bit to zero. This field is used only if the MLID was developed with Novell's <i>LAN Driver Developer's Kit for HSMs</i> .
13	MM_PROMISCUOUS_BIT	The MLID sets this bit if it supports promiscuous mode.
14	MM_NONCANONICAL_BIT	See Bit 15.

12-16 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-2
MLIDCFG_ModeFlags Bits Description

Bit #	Name	Description
15	MM_PHYS_NODE_ADDR_BIT	<p>The MLID sets or clears bits 14 and 15 to indicate whether the MLID's configuration table <i>NodeAddress</i> field contains a canonical or noncanonical address.</p> <p>Bit 14 indicates when the configuration table is using the noncanonical format.</p> <p>Bit 15 indicates whether the MLID supports the use of a <i>PhysicalNodeAddress</i>.</p> <p>The following are the bit 15 and 14 combinations:</p> <p>00 = MLIDCFG_NodeAddress format is unspecified. The node address is assumed to be in the physical layer's native format; octet bit reversal is not supported.</p> <p>01 = This is an illegal value and must not occur.</p> <p>10 = MLIDCFG_NodeAddress is canonical and octet bit reversal is supported.</p> <p>11 = MLIDCFG_NodeAddress is noncanonical and octet bit reversal is supported.</p>

MLID Data Structures 12-17

MLIDCFG_Flags Field

Figure 12-2
MLIDCFG_Flags Field Default Values

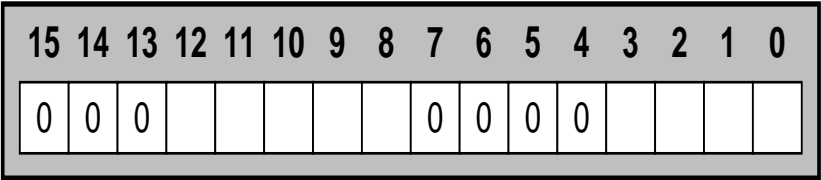


Table 12-3 describes.....

Table 12-3
MLIDCFG_Flags Bit Description

Bit #	Name	Description
8	MF_HUB_MANAGEMENT_BIT	Set to 1 if the MLID 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 MLID supports multicast addressing.</p> <p>The MLID sets bit 10 if it has specialized adapter hardware (such as hardware that utilizes CAM memory).</p> <p>If an MLID that usually defaults to functional addressing also supports group addressing sets bit 10, 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 MLID does not need to perform any checking. The MLID can dynamically set and clear bit 9. For example, if the adapter utilizes CAM memory, but has temporarily run out memory, the MLID must temporarily filter the group addresses. In this case, the MLID must reset bit 9.</p>

12-18 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-3
MLIDCFG_Flags Bit Description

Bit #	Name	Description
9/10		<p>Bits 9 and 10 combinations are as follows:</p> <p>00= The format of the multicast address defaults to that of the topology:</p> <p>Ethernet => Group Addressing (Multicast Addressing)</p> <p>Token-Ring => Group Addressing and Functional Addressing</p> <p>FDDI => Group Addressing</p> <p>01= Illegal value which must not occur.</p> <p>10= Filter group address in MLID. Group addressing is supported by the specialized adapter hardware.</p> <p>11= Adapter filtered group address. MLID software checking is not required. Group addressing is supported by the specialized adapter hardware.</p> <p>See also ODI Specification Supplement: Canonical and Noncanonical Addressing for information regarding octet bit reversal</p>
11	MF_RECONFIG_BIT	<p>This bit indicates to an MLID that indirect (such as file based) configuration information for the associated interface instance may have changed. This bit can be set by any caller prior to calling the MLIDReset function. It is to be examined by the MLIDReset and cleared upon completion of the reset processing. This bit has no meaning for MLIDs which do not support use of indirect (such as file based) configuration information.</p>
12	MF_PRIORITYSUP_BIT	<p>The MLID sets this bit during initialization if the MLID has set the MLIDCFG_PrioritySup field to something other than 0.</p> <p>Note: The MLID may temporarily clear this bit to disable priority support.</p>

MLIDCFG_SharingFlags Field

Figure 12-3
MLIDCFG_SharingFlags Field Default Values

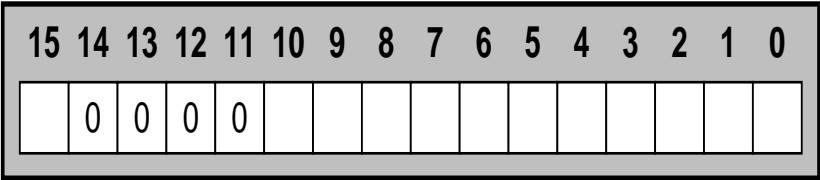


Table 12-4
MLIDCFG_SharingFlags Bits Description

Bit #	Name	Description
0	MS_SHUTDOWN_BIT	Set to 1 if the logical board is currently shut down. This bit must 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 adaptor can share I/O port 0.
2	MS_SHARE_PORT1_BIT	Set to 1 if the adaptor can share I/O port 1.
3	MS_SHARE_MEMORY0_BIT	Set to 1 if the adaptor can share memory range 0.
4	MS_SHARE_MEMORY1_BIT	Set to 1 if the adaptor can share memory range 1.
5	MS_SHARE_IRQ0_BIT	Set to 1 if the adaptor can share interrupt 0
6	MS_SHARE_IRQ1_BIT	Set to 1 if the adaptor can share interrupt 1.
7	MS_SHARE_DMA0_BIT	Set to 1 if the adaptor can share DMA channel 0.
8	MS_SHARE_DMA1_BIT	Set to 1 if the adaptor can share DMA channel 1.
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 MLID sets this bit if the command line passed to DriverInit is not empty.

12-20 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-4
MLIDCFG_SharingFlags Bits Description

Bit #	Name	Description
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.
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 other values, except for combinations of the above values, are invalid unless stated otherwise.

Table 12-5
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

12-22 ODI Specification: Protocol Stacks and MLIDs (C Language)

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

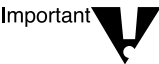
Adapter Data Space

The MLID must allocate and initialize a structure called *DriverAdapterDataSpaceTemplate*. This structure must contain the data that is specific to a particular LAN adapter. You must determine what hardware-specific fields the MLID needs in this structure in order to drive its particular LAN adapter. But keep in mind that this structure must also contain the MLID statistics table.

MLID Statistics Table

This section describes the MLID statistics table in detail for MLIDs that interface directly to the LSL. This section includes a sample of the statistics table code and a description of each of the statistics table’s fields.

All MLIDs must keep a statistics table for the purpose of network management. The following is the format of an MLID statistics table.



A protocol stack treats this table as read only!

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

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



When the statistics counters reach their maximum value, they wrap back to their beginning value.

MLID Statistics Table Structure

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

Field Descriptions

StatUseFlag

Values are defined as follows:

ODI_STAT_UNUSED	<i>StatCounter</i> entry not in use.
ODI_STAT_UINT32	<i>StatCounter</i> is a pointer to an UINT32 counter.
ODI_STAT_UINT64	<i>StatCounter</i> is a pointer to an UINT64 counter.
ODI_STAT_MEON_STRING	<i>StatCounter</i> is a pointer to a Null terminated string of MEON. The maximum string length is 256, including the NULL termination.
ODI_STAT_UNTYPED	<i>StatCounter</i> is a pointer to a UINT8 array preceded by its length (UINT32). This value is generally used for debugging and is displayed in hexadecimal bytes.

StatString

Pointer to a NULL terminated MEON string that describes the statistics counter.

StatCounter

Defined by *StatUseFlag*.

Statistics Table Structure

```
typedef struct  _MLID_STATS_TABLE_  
{  
    UINT16          MStatTableMajorVer;  
    UINT16          MStatTableMinorVer;  
    UINT32          MNumGenericCounters;  
    STAT_TABLE_ENTRY (*MGenericCountsPtr)[ ];  
    UINT32          MNumMediaCounters;  
    STAT_TABLE_ENTRY (*MMediaCountsPtr)[ ];  
    UINT32          MNumCustomCounters;  
    STAT_TABLE_ENTRY (*MCustomCountsPtr)[ ];  
} MLID_STATS_TABLE;
```

Table 12-6
MLID Statistics Table Fields

Name	Type	Description
MStatTableMajorVer	UINT16	This field contains the major version number of the statistics table. The current major number is 4.
MStatTableMinorVer	UINT16	This field contains the minor version number of the statistics table. The current minor version number is 0.
MNumGenericCounters	UINT32	This field has the total number of generic <i>STAT_TABLE_ENTRY</i> counters in this portion of this table. This field is set to 20 for this specification.
MGenericCountsPtr	STAT_TABLE_ENTRY*	Pointer to an array of <i>STAT_TABLE_ENTRY</i> counters [<i>MNumGenericCounters</i>].
MNumMediaCounters	UINT32	This field has the total number of media specific <i>STAT_TABLE_ENTRY</i> counters in this portion of this table. This field is set to the following values: Token-Ring 13 Ethernet 8 FDDI 10
MMediaCountsPtr	STAT_TABLE_ENTRY*	Pointer to an array of <i>STAT_TABLE_ENTRY</i> counters [<i>MNumMediaCounters</i>].
MNumCustomCounters	UINT32	The total number of custom <i>STAT_TABLE_ENTRY</i> counters in this portion of this table. This field is variable (dependent on the MLID).
MCustomCountersPtr	STAT_TABLE_ENTRY*	Pointer to an array of <i>STAT_TABLE_ENTRY</i> 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;

MEON_STRING     MTotalTxPacketStr[]      "Total Tx Packet Count";
MEON_STRING     MTotalRxPacketStr[]      "Total Rx Packet Count";
MEON_STRING     MNoECBAvailableStr[]     "No ECB Available Count";
MEON_STRING     MPacketTxTooBigStr[]     "Packet Tx Too Big Count";
MEON_STRING     MPacketTxTooSmallStr[]   "Packet Tx Too Small Count";
MEON_STRING     MPacketRxOverflowStr[]   "Packet Rx Overflow Count";
MEON_STRING     MPacketRxTooBigStr[]     "Packet Rx Too Big Count";
MEON_STRING     MPacketRxTooSmallStr[]   "Packet Rx Too Small Count";
MEON_STRING     MTotalTxMiscStr[]        "Total Tx Misc Count";
MEON_STRING     MTotalRxMiscStr[]        "Total Rx Misc Count";
MEON_STRING     MRetryTxStr[]            "Retry Tx Count";
MEON_STRING     MChecksumErrorStr[]      "Checksum Error Count";
MEON_STRING     MHardwareMismatchStr[]   "Hardware Mismatch Count";
MEON_STRING     MTotalTxOKByteStr[]      "Total Tx OK Byte Count";
MEON_STRING     MTotalRxOKByteStr[]      "Total Rx OK Byte Count";
MEON_STRING     MTotalGroupAddrTxStr[]   "Total Group AddrTx Count";
MEON_STRING     MTotalGroupAddrRxStr[]   "Total Group Addr Rx Count";
MEON_STRING     MAdapterResetStr[]       "Adapter Reset Count";
MEON_STRING     MAdapterOprTimeStampStr[] "Adapter Opr Time Stamp";
MEON_STRING     MQDepthStr[]             "Tx Queue Depth";

```

12-28 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

STAT_TABLE_ENTRY      MGenericCounters  [NUM_GENERIC_COUNTERS] =
{
    { ODI_STAT_UINT32, &MTotalTxPacketCount,      &MTotalTxPacketStr },
    { ODI_STAT_UINT32, &MTotalRxPacketCount,      &MTotalRxPacketStr },
    { ODI_STAT_UINT32, &MNoECBAvailableCount,     &MNoECBAvailableStr },
    { ODI_STAT_UINT32, &MPacketTxTooBigCount,      &MPacketTxTooBigStr },
    { ODI_STAT_UINT32, &MPacketTxTooSmallCount,    &MPacketTxTooSmallStr },
    { ODI_STAT_UINT32, &MPacketRxOverflowCount,    &RMPacketRxOverflowStr },
    { ODI_STAT_UINT32, &MPacketRxTooBigCount,      &MPacketRxTooBigStr },
    { ODI_STAT_UINT32, &MPacketRxTooSmallCount,    &MPacketRxTooSmallStr },
    { ODI_STAT_UINT32, &MTotalTxMiscCount,         &MTotalTxMiscStr },
    { ODI_STAT_UINT32, &MTotalRxMiscCount,         &MTotalRxMiscStr },
    { ODI_STAT_UINT32, &MRetryTxCount,             &M&MRetryTxStr },
    { ODI_STAT_UINT32, &MChecksumErrorCount,       &MChecksumErrorStr },
    { ODI_STAT_UINT32, &MHardwareRxMismatchCount, &MHardwareMismatchStr },
    { ODI_STAT_UINT64, &MTotalTxOKByteCount,       &MTotalTxOKByteStr },
    { ODI_STAT_UINT64, &MTotalRxOKByteCount,       &MTotalRxOKByteStr },
    { ODI_STAT_UINT32, &MTotalGroupAddrTxCount,    &MTotalGroupAddrTxStr },
    { ODI_STAT_UINT32, &MTotalGroupAddrRxCount,    &MTotalGroupAddrRxStr },
    { ODI_STAT_UINT32, &MAdapterResetCount,        &MAdapterResetStr },
    { ODI_STAT_UINT32, &MAdapterOprTimeStamp,     &MAdapterOprTimeStampStr },
    { ODI_STAT_UINT32, &MQDepth,                  &MQDepthStr }
};

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

```

MLID Data Structures 12-29

Table 12-7
MLID Statistics Table Generic Counters

Name	Type	Description
MTotalTxPacketCount	UINT32	Number of packets successfully transmitted onto the media. Mandatory.
MTotalRxPacketCount	UINT32	Number of packets reported as successfully received without errors. This counter is independent of whether the packet is accepted. Mandatory.
MNoECBAvailableCount	UINT32	Number of times an incoming packet was discarded due to lack of host receive buffers or the host not wanting the packet. Mandatory.
MPacketTxTooBigCount	UINT32	Number of times a send packet was too big for transmission. 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. Optional
MPacketRxOverflowCount	UINT32	Number of times the adapter's receive buffer pool was exhausted, which caused subsequent incoming packets to be discarded. 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. Mandatory.
MPacketRxTooSmallCount	UINT32	Number of times a packet was received that was too small for media definitions. Optional.
MTotalTxMiscCount	UINT32	This counter is incremented if the MLID failed to transmit and has no appropriate generic counter to increment. Mandatory.
MTotalRxMiscCount	UINT32	This counter is incremented if the MLID receives a packet with errors and has no appropriate generic counter to increment. Mandatory.
MRetryTxCount	UINT32	Number of times the MLID retried a transmit operation because of a failure. Optional.
MChecksumErrorCount	UINT32	Number of times the MLID received a packet with corrupt data (for example, CRC errors). Optional.

12-30 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table 12-7

MLID Statistics Table Generic Counters

Name	Type	Description
MHardwareRxMismatchCount	UINT32	Number of times the MLID received a packet that did not pass the length consistency checks. Optional.
MTotalTxOKByteCount	UINT64	Number of bytes (including low-level headers) the MLID successfully transmitted onto the media. Mandatory.
MTotalRxOKByteCount	UINT64	Number of bytes (including low-level headers) the MLID successfully received. Mandatory.
MTotalGroupAddrTxCount	UINT32	Number of packets the MLID transmitted with a group destination address. Mandatory.
MTotalGroupAddrRxCount	UINT32	Number of packets the MLID received with a group destination address. Mandatory.
MAdapterResetCount	UINT32	Number of times the adapter was reset due to an internal failure or a call to the MLIDReset function. Mandatory.
MAdapterOprTimeStamp	UINT32	This counter contains the time, (platform dependent clock, for example number of ticks), the adapter last changed operational state, such as Loaded, MLID Shutdown and Reset Control Service functions. Mandatory
MQDepth	UINT32	Number of Transmit ECBs that are queued for the adapter. The MLID maintains this field. Mandatory.

MLID Statistics Table Media Specific Counters

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

Token-Ring

Media specific counters array *STAT_TABLE_ENTRY* for Token-Ring are as follows:

Table 12-8
Media Specific Counters for Token-Ring

Size	Label	Description
UINT32	TRN_ACErrCounter	This counter is incremented when 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. Mandatory.
UINT32	TRN_AbortDelimiterCounter	This counter is incremented when a station transmits an abort delimiter while transmitting. Mandatory.
UINT32	TRN_BurstErrCounter	This counter is incremented when a station detects the absence of transitions for five half-bit times (burst-five error). Note that only one station detects a burst-five error, because the first station to detect it converts it to a burst-four. Mandatory.
UINT32	TRN_FrameCopiedErrCounter	This counter is incremented when 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. Mandatory.
UINT32	TRN_FrequencyErrCounter	This counter is incremented when the frequency of the incoming signal differs by more than that specified in Section 7 (IEEE Std 802.5-1989) from the expected frequency. Mandatory.
UINT32	TRN_InternalErrCounter	This counter is incremented when a station recognizes a recoverable internal error. This can be used for detecting a station in marginal operating condition. Mandatory.

Table 12-8
Media Specific Counters for Token-Ring

Size	Label	Description																								
UINT32	TRN_LastRingStatus	<p>This value contains 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>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>This counter is incremented when a frame or token is copied or repeated by a station. The E bit is 0 in the frame or token and one of the following conditions 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>Mandatory.</p>																								
UINT32	TRN_LostFrameCounter	<p>This counter is incremented when a station is transmitting and its TRR timer expires. This counts how often frames transmitted by a particular station fails to return to it, thus causing the active monitor to issue a new token. Mandatory.</p>																								
UINT32	TRN_TokenErrorCounter	<p>This counter is incremented when a station acting as the active monitor recognizes an error condition that needs a token transmitted. This occurs when the TVX timer expires. Mandatory.</p>																								
UINT64	TRN_UpstreamNodeAddress	<p>This contains the upstream neighbor node address, right justified with leading zeros. Mandatory.</p>																								
UINT32	TRN_LastRingID	<p>This contains the value of the local ring. Mandatory.</p>																								

MLID Data Structures 12-33

Table 12-8
Media Specific Counters for Token-Ring

Size	Label	Description
UINT32	TRN_LastBeaconType	This contains the value of the last beacon type. Mandatory.

Ethernet

Media specific counters array *STAT_TABLE_ENTRY* for Ethernet are as follows:

Table 12-9
Media Specific Counters for Ethernet

Size	Label	Description
UINT32	ETH_TxOKSingleCollisionsCount	This counter contains a 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. Mandatory.
UINT32	ETH_TxOKMultipleCollisionsCount	This counter contains a count of frames that are involved in more than one 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 greater than 2 and less than or equal to the attempt limit of the network controller used by the MLID. (The attempt limit is specified by MLIDCFG_SendRetries.) Mandatory.
UINT32	ETH_TxOKButDeferred	This counter contains a count of frames whose transmission was delayed on its first attempt because the medium was busy. Mandatory.
UINT32	ETH_TxAbortLateCollision	This counter contains a 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. Mandatory.
UINT32	ETH_TxAbortExcessCollision	This counter contains a count of frames that, due to excessive collisions, did not transmit successfully. This counter is incremented when the value of attempts during a transmission equals the attempt limit specified by MLIDCFG_SendRetries. Mandatory.

Table 12-9
Media Specific Counters for Ethernet

Size	Label	Description
UINT32	ETH_TxAbortCarrierSense	This counter contains a count of frames that the <i>carrierSense</i> variable was not asserted or was deasserted during the transmission of a frame without collision. Mandatory.
UINT32	ETH_TxAbortExcessiveDeferral	This counter contains a count of frames that were deferred for an excessive period of time. This counter must only be incremented once per LLC transmission. Mandatory.
UINT32	ETH_RxAbortFrameAlignment	This counter contains a count of frames that are not an integral number of bytes in length and do not pass the FCS check. Mandatory.

FDDI

Media specific counters array *STAT_TABLE_ENTRY* for FDDI are as follows:

Table 12-10
Media Specific Counters for FDDI

Size	Label	Description
UINT32	FDI_ConfigurationState	(ANSI fddiSMTCFState) This field contains attachment configuration for the station or concentrator. 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 Mandatory.
UINT64	FDI_UpstreamNode	(ANSI fddiMACUpstreamNbr) This counter contains the MAC's upstream neighbor's long individual MAC address; 0 if unknown. Mandatory.
UINT64	FDI_DownstreamNode	(ANSI fddiMACDownstreamNbr) This field contains the MAC's downstream neighbor's long individual MAC address; 0 if unknown. Mandatory.
UINT32	FDI_FrameErrorCount	This field contains a count of the number of frames that were detected in error by this MAC that had not been detected in error by another MAC. Mandatory.
UINT32	FDI_FramesLostCount	This field contains a count of the number of instances that this MAC detected a format error during frame reception such that the frame was stripped. Mandatory.

Table 12-10
Media Specific Counters for FDDI

Size	Label	Description
UINT32	FDI_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>Mandatory.</p>
UINT32	FDI_LCTFailureCount	<p>This counter contains the count of consecutive times the link confidence test (LCT) has failed during connection management.</p> <p>Mandatory.</p>
UINT32	FDI_LemRejectCount	<p>This counter contains a link error monitoring count of the times that a link has been rejected. Mandatory.</p>
UINT32	FDI_LemCount	<p>This counter contains the aggregate link error monitor error count; this field is set to 0 only on station power-up. Mandatory.</p>
UINT32	FDI_LConnectionState	<p>This field contains 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 = Maintenance</p> <p>Mandatory.</p>

12-38 ODI Specification: Protocol Stacks and MLIDs (C Language)

chapter **12** *MLID Initialization*

Chapter Overview

This chapter explains the issues involved in the initialization and registration of the MLID, including the following topics:

- Determining hardware options
- Parsing the command line
- Allocating frame and adapter data space
- Setting up the board service routine

S

The MLID Initialization Routine

When the MLID is loaded, the MLID initialization routine is called.

The MLID initialization routine must do the following tasks.

- Locate the LSL
- Allocate the frame and adapter data spaces
- Parse the LOAD command line
- Process the custom command line keywords and custom firmware
- Register hardware options
- Initialize the adapter hardware
- Register the MLID with the LSL
- Set up a board service routine
- Start timeout checks

If the MLID is unsuccessful in these initialization tasks, it should return `ODISTAT_FAIL`.

MLID Initialization **13-1**

Initialization Parameters Passed on the Stack

The initialization routine is called as follows:.

Syntax

```

ODISTAT DriverInit (
MODULE_HANDLE      *ModuleHandle,
SCREEN_HANDLE      *ScreenHandle,
MEON_STRING        *CommandLine,
MEON_STRING        *ModuleLoadPath,
UINT32UninitializedDataLength,
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.

CommandLine

Pointer to the command line that was used to load the driver. This parameter is used when parsing for the hardware configuration information from the command line.

ModuleLoadPath

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

UninitializedDataLength

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

CustomDataFileHandle

The custom data file is appended to the end of your MLID. Because the MLID was opened during loading, this handle points to a structure that the

13-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

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 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

<code>ODISTAT_SUCCESSFUL</code>	The MLID initialized successfully.
<code>ODISTAT_FAIL</code>	The MLID failed to initialize successfully.

Locating the LSL

The LSL module must reside in the system before the any MLIDs can be loaded. An MLID must first obtain the LSL API entry points to initialize. (See “Locating the LSL” in Chapter 10, “LSL Support Routines”.)

Frame and Adapter Data Spaces

Each time the operating system calls the initialization routine, the MLID creates a logical board. The initialization routine must allocate the frame data space for the logical board to use, and the MLID must create a copy of the configuration table in this space. If the MLID is being loaded for the first time for a given adapter, the initialization routine must also allocate the adapter data space for the adapter.

Determining Hardware Options

The MLID must determine the hardware configuration of the LAN adapter. This includes parameters such as:

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

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

For EISA and Micro Channel busses, 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 busses 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 slot parameter is used to contain the Hardware Instance Number (HIN). The HIN is a system-wide, bus-independent, unique handle for a device. The HIN enables the MLID 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 multi-bus systems.

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

13-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

- 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).

If the MLID supports an adapter with a product ID that is retrievable according to a standard (such as EISA, PCI, Micro Channel, or PC Card), the MLID must perform the following steps:

1. Scan for the adapter for each supported bus type using **SearchAdapter**. **SearchAdapter** gets the bus tag and the unique identifier for each hardware instance found.
2. Call **GetInstanceNumber** once for each hardware instance found. **GetInstanceNumber** uses the bus tag and the unique identifier to get the HIN for each adapter. The MLID uses this HIN when parsing to determine which hardware instance the MLID is being loaded for. The MLID then places the selected HIN in the *MLIDCFG_Slot* field of the configuration table.
3. Call **GetInstanceNumberMapping** with the configuration table *MLIDCFG_Slot* field as an input parameter. The corresponding bus tag and unique identifier will be returned. The bus tag returned from **GetInstanceNumberMapping** must be placed in the *MLIDCFG_DBusTag* field of the configuration table.
4. Call **GetCardConfigInfo** to get the adapter's configuration and fill out the I/O portion of the configuration table. (The bus tag and the unique identifier for the selected HIN are used as input parameters for **GetCardConfigInfo**.)

If the MLID supports a bus adapter whose product ID is not retrievable according to a standard (such as Legacy ISA), the MLID must parse the command line to get the hardware configuration information, using the following steps:

Note



If the MLID needs to get more than one parameter to determine the value(s) of other parameters, it should parse the command line more than once.

1. Parse the command line for all *standard* keywords.
2. Parse the command line for all *custom* keywords.

Note



In an advanced installation environment, custom keywords should be restricted to ensure optimal performance of automatic driver selection and loading.

Note



For family drivers that support adapters of more than one bus type (including legacy ISA), the custom 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.

3. After parsing the command line, fill in the I/O portion of the configuration table in the frame data space for the logical board. The *MLIDCFG_Slot* field of the configuration table will contain the selected adapter HIN, and the *MLIDCFG_DBusTag* field will contain the busTag.
4. When the MLID has all the needed information for the configuration table, **DriverInit** calls **RegisterHardwareOptions**.

Important



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

Registering Hardware Options

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

The MLID must be able to process the following three conditions on return from **RegisterHardwareOptions**:

- If the return value indicates that a new adapter was successfully registered, the MLID must proceed with the hardware initialization.
- If the return value indicates that a new frame type for an existing adapter was successfully registered, the initialization is complete.
- If the returned value indicates that the MLID was unable to register the hardware options, **DriverInit** should clean up and return with an error code.

Initializing the Adapter

At this point, the MLID initializes the adapter hardware.

When the MLID initializes the adapter hardware, it must include all software controlled configuration of hardware, and may also include hardware tests and diagnostics such as RAM testing. **DriverReset** can be called to do this part of the **DriverInit** procedure since it performs all of the steps necessary to initialize hardware.

During initialization, the MLID may adjust MLIDCFG_MaxFrameSize down if necessary.

Important



If firmware is to be installed on the adapter, it must be done when the MLID is initializing the adapter.

If an error occurs during hardware initialization, **DriverInit** should generate an error message, return the allocated resources, and return ODISTAT_FAIL to the operating.

When hardware initialization is successful, the MLID must do the following tasks:

- Set the MLIDCFG_SharingFlags MS_SHUTDOWN_BIT to zero

- Call **NESLProduceMLIDEvent** to produce a NESL Service Resume MLID Card Insertion Complete event
- Return **ODISTAT_SUCCESSFUL** to the operating system

Registering with the LSL

DriverInit calls **CLSL_RegisterMLID** to register the MLID with the LSL.

During registration, the MLID passes the LSL a pointer to the MLID's registration structure, which contains pointers to the MLID's transmit and control handling interfaces.

When registration is successful, the LSL assigns a logical board number to the adapter, and the MLID places it in the configuration table.

At this point, MLIDs for intelligent bus master adapters can pass the board number and the frame ID information to the adapter.

Setting up a Board Service Routine

At this point, the MLID registers its board service routine(s) with the operating system.

If the adapter is interrupt driven, it must be ready to process interrupts before it calls the operating system.

Polling MLIDs can use **GetPollSupportLevel** to determine the level of polling support provided on the platform.

Scheduling Timeout Callbacks

If an adapter is interrupt driven, the MLID can schedule a timer event to determine when a board is unable to complete a transmission.

To schedule a timer event, the MLID must call **CLSL_ScheduleAESEvent**, which schedules periodic calls to the MLID's **DriverAES** routine. However, the MLID cannot call **CLSL_ScheduleAESEvent** until after it has called **CLSL_RegisterMLID**.

If an adapter is not interrupt driven, the MLID should use polling routines to determine when a board is unable to complete a transmission.

13-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

chapter **13** *MLID Packet Reception*

Chapter Overview

This chapter is a brief overview of the various MLID reception methods available, and describes how to provide a LOOKAHEAD buffer.

You should review this chapter before writing the MLID's board service routine.

S

Reception Methods

When a physical board gets a packet from the network, the board service routine processes the packet.

Usually, the board service routine requests an ECB from the LSL and fills it in with information about the incoming packet. The MLID then passes the address of the ECB to the LSL, and the LSL transfers the information to the appropriate protocol stack.

The board service routine may also receive transmission complete interrupts. In which case, the board service routine must be able to handle the additional overhead involved in completing and re-issuing the transmission requests.

There are several different methods used for packet reception. The method selected is typically dependent on the adapter's data transfer method.

MLID Packet Reception **14-1**

Reception Method - Option 1

Option 1 is the simplest and most preferred reception method for host DMA adapters and bus master adapters.

For Reception Method - Option 1, the MLID must perform the following tasks:

1. Save the processor state and disable the system interrupts if they are not already disabled.
2. Start an internal critical section.
3. Enable the system interrupts so that external processes can execute. To do this, the MLID must perform the following tasks:
 - a. If the adapter is interrupt driven, disable the physical board's interrupts and dismiss the interrupt using the appropriate operating system call.
 - b. Enable system interrupts.
4. Get an ECB from the LSL using **CLSL_GETSizedECB**.
5. If an ECB is returned, fill in the ECB. Refer to Appendix A, "Event Control Blocks (ECBs)" for detailed descriptions of the ECB fields.

If an ECB is not available, discard the packet and increment the appropriate statistics counters.



The data buffer described by the ECB can be fragmented.

If the packet contains errors, set the GlobalError bit in the *ECB_DriverWorkspace* field of the ECB structure and fill out the appropriate error bit in the *ECB_PreviousLink* field.

6. Filter the packet and the frame header, and set up a LOOKAHEAD buffer. See the "Receive Lookahead" section in Chapter 5 "Protocol Stack Packet Reception".



The *LkAhd_PreFilledECB* field in the LOOKAHEAD buffer is set to point to the ECB.

If the packet contains errors, set the GlobalError bit in the *LkAhd_DestType* field of the LOOKAHEAD structure and fill out the appropriate error bit in the *LkAhd_PktAttr* field.

7. Call **CLSL_GetStackECB** with a pointer to the LOOKAHEAD buffer.

8. Increment the appropriate statistic counters.
9. Check whether the adapter has received another packet. If the adapter has received another packet, initiate the packet reception process again.
10. Set the interrupts to their original state by taking the following steps in the order given:
 - a. Disable system interrupts.
 - b. If the adapter is interrupt driven, enable the physical board's interrupts.
 - c. Enable the system interrupts.
11. Terminate the internal critical section and call **CLSL_ServiceEvents**. **CLSL_ServiceEvents** must be called before exiting the procedure to process ECBs in the hold queue.
12. Return control to the calling routine by restoring the registers and doing an IRET or another return as required by the platform that the MLID is executing on.

MLID Packet Reception **14-3**

Reception Method - Option 2

Option 2 is the preferred reception method for shared RAM adapters and programmed I/O adapters.

Option 2 uses a LOOKAHEAD process, where the frame header information is confirmed before the packet is transferred from the adapter to the ECB.

The adapter's data transfer mode determines how the LOOKAHEAD process is handled.

For shared RAM adapters, the LOOKAHEAD buffer starts at the beginning offset of the packet in shared RAM.

For programmed I/O adapters, the LOOKAHEAD buffer must be the size (in bytes) of the maximum frame header size plus the value in the *MLIDCFG_LookAheadSize* field of the MLID configuration table.

For Reception Method - Option 2, the MLID must perform the following tasks:

1. Save the processor state and disable the system interrupts if they are not already disabled.
2. Start an internal critical section.
3. Enable the system interrupts so that external processes can execute. To do this, the MLID must perform the following tasks:
 - a. If the adapter is interrupt driven, disable the physical board's interrupts and dismiss the interrupt using the appropriate operating system call.
 - b. Enable system interrupts.
4. Filter the packet and the frame header, and set up a LOOKAHEAD buffer. See the "Receive Lookahead" section in Chapter 5 "Protocol Stack Packet Reception".

Note



If the packet contains errors, set the GlobalError bit in the *LkAhd_DestType* field of the LOOKAHEAD structure and fill out the appropriate error bit in the *LkAhd_PktAttr* field.

5. Call **CLSL_GetStackECB** with a pointer to the LOOKAHEAD buffer.

If a protocol stack wants the packet, **CLSL_GetStackECB** returns an ECB.

If an ECB is not available, **CLSL_GetStackECB** discards the packet and increments the appropriate statistics counters.

6. Fill in the ECB. Refer to Appendix A, “Event Control Blocks (ECBs)” for detailed descriptions of the ECB fields.
7. Increment the appropriate statistics counters.

Note



The data buffer described by the ECB can be fragmented.

If the packet contains errors, set the GlobalError bit in the *ECB_DriverWorkspace* field of the ECB structure and fill out the appropriate error bit in the *ECB_PreviousLink* field.

8. Call **CLSL_HoldEvent**. This call places the specified ECB on the LSL's holding queue for processing.
9. Increment the appropriate statistic counters.
10. Determine if the adapter has received another packet. If the adapter has received another packet, initiate the packet reception process again.
11. Set the interrupts to their original state by taking the following steps in the order given:
 - a. Disable system interrupts.
 - b. If the adapter is interrupt driven, enable the physical board's interrupts.
 - c. Enable the system interrupts.
12. Terminate the internal critical section and call **CLSL_ServiceEvents**. **CLSL_ServiceEvents** must be called before exiting the procedure to process ECBs in the hold queue.
13. Return control to the calling routine by restoring the registers and doing an IRET or another return as required by the platform that the MLID is executing on.

Reception Method - Option 3

Pipelined adapters can be configured to interrupt before they receive the complete packet.

MLID Packet Reception 14-5

At driver initialization time, the adapter must be configured to wait until it has received at least the maximum header size plus the value in the *MLIDCFG_LookAheadSize* field of the MLID configuration table before it interrupts.

For Reception Method - Option 3, the MLID must perform the following tasks:

1. Save the processor state and disable the system interrupts if they are not already disabled.
2. Start an internal critical section.
3. Enable the system interrupts so that external processes can execute. To do this, the MLID must perform the following tasks:
 - a. If the adapter is interrupt driven, disable the physical board's interrupts and dismiss the interrupt using the appropriate operating system call.
 - b. Enable system interrupts.
4. Set up a LOOKAHEAD buffer. See the "Receive Lookahead" section in Chapter 5 "Protocol Stack Packet Reception".

Note



The *LkAhd_FrameDataSize* field in the LOOKAHEAD buffer must be set to UNUSED. This indicates that the entire packet has not been received and that its error status is unknown at this time.

5. Call **CLSL_GetStackECB** with a pointer to the LOOKAHEAD buffer.

If a protocol stack wants the packet, **CLSL_GetStackECB** returns an ECB.

If an ECB is not available, **CLSL_GetStackECB** discards the packet and increments the appropriate statistics counters.

6. Fill in the ECB and call **CLSL_HoldEvent/CLSL_ServiceEvents**. Refer to Appendix A, "Event Control Blocks (ECBs)" for detailed descriptions of the ECB fields.
7. Increment the appropriate statistic counters.

Note



The data buffer described by the ECB can be fragmented.

If the packet contains errors, set the GlobalError bit in the *ECB_DriverWorkspace* field of the ECB structure and fill out the appropriate error bit in the *ECB_PreviousLink* field.

8. Call **CLSL_HoldEvent** to place the specified ECB on the LSL's holding queue for processing.
9. Increments the appropriate statistic counters.
10. Determine whether the adapter has received another packet. If the adapter has received another packet, initiate the packet reception process again.
11. Set the interrupts to their original state by taking the following steps in the order given:
 - a. Disable system interrupts.
 - b. If the adapter is interrupt driven, enable the physical board's interrupts.
 - c. Enable the system interrupts.
12. Terminate the internal critical section and call **CLSL_ServiceEvents**. **CLSL_ServiceEvents** must be called before exiting the procedure to process ECBs in the hold queue.
13. Return control to the calling routine by restoring the registers and doing an IRET or another return as required by the platform that the MLID is executing on.

MLID Packet Reception **14-7**

Using Shared Interrupts

MLIDs 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:

- PCI and Micro Channel buses always use level-triggered interrupts and can support shared interrupts.
- ISA buses normally use edge-triggered interrupts and will not support shared interrupts unless external logic exists on the adapters for sharing interrupts.
- EISA buses normally use edge-triggered interrupts, but each interrupt can be individually configured to the level-triggered mode to support shared interrupts.
- Other buses vary in their use of edge and level triggered interrupts.
- The MLID must indicate that the adapters are sharing interrupts by setting the appropriate bit in the *MLIDCFG_SharingFlags* field of the configuration table.

chapter **14** *MLID Packet Transmission*

Chapter Overview

This chapter is a brief overview of the MLID Packet Transmission routine.

You should review this chapter before writing an MLID Packet Transmission routine.

S

MLID Packet Transmission Routine

The MLID transmits packets through the physical board. When a packet is ready to be sent, the protocol stack prepares an Event Control Block (ECB) and calls the LSL's send packet routine (**CLSL_SendPacket**). The LSL inspects the ECB board number and calls the associated MLID send handler. (The send handler entry point is exchanged with the LSL during initialization time.)

Generally, to prepare a packet for transmission when called by the LSL, the MLID must perform the following steps; however, the steps and their order are dependent on the operating system platform.

1. Start an internal critical section.
2. Enable the system interrupts so that external processes can execute. To do this, the MLID must perform the following tasks in the order given:
 - a. Disable system interrupts.
 - b. If the adapter is interrupt driven, disable the physical board's interrupts
 - c. Enable system interrupts.
3. Determine if the hardware is busy with a transmission. If it is, queue the send.

MLID Packet Transmission **15-1**

4. When the hardware is not busy with a transmission, set the busy flag and inspect the ECB for raw sends.
5. Determine whether the packet is a raw send.
 - a. If the packet is a raw send, do not generate the media header.
 - b. If the packet is not a raw send, generate a media header.
6. Begin transmission of the header and data by sending a request to the hardware.
7. Increment the appropriate counter(s).

8. The MLID must perform one of the following tasks, depending on whether the transmission is a lying send or a non-lying send:

- Lying send

Immediately following the send request to the hardware, set the *ECB_Status* field to 0 as if the send had completed successfully.

- Non-lying send

After the send operation has completed, set the *ECB_Status* field to 0 if the send was successful or to an error code if the send was unsuccessful. (This usually takes place in the board service routine after receiving a send complete interrupt.)

9. If a transmit monitor is registered, pass the completed TCB to the transmit monitor for inspection.
10. Return the ECB by calling **CLSL_SendComplete**.
11. Check the MLID internal queue for pending transmissions. If any are found, start sending them.
12. Set the interrupts to their original state by taking the following steps in the order given:
 - a. Disable system interrupts.
 - b. If the adapter is interrupt driven, enable the physical board's interrupts.
 - c. Enable the system interrupts.
13. Terminate the internal critical section and call **CLSL_ServiceEvents**. **CLSL_ServiceEvents** must be called before exiting the procedure to process ECBs in the hold queue.
14. Return control to the calling routine.

Note



The entity that made the transmit request should not poll for completion; instead, the entity should wait for the transmit request ESR in the transmit ECB to be called. Polling for completion of a transmit request can cause a dead-lock and system failure.

Priority Transmission Support

The following algorithm is used for priority transmission support.

1. During **DriverInit**, the CHSM must set the following parameters:
 - The MF_PRIORITY_BIT in the *MLIDCFG_Flags* field of the MLID configuration table.
 - The *MLIDCFG_PrioritySup* field of the MLID configuration table to indicate the number of levels available.

The MLID can set or reset the MF_PRIORITYSUP_BIT as the MLID 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 must set the *ECB_StackID* field to a value greater than or equal to 0x0FFF0. Refer to Appendix A, "Event Control Blocks (ECBs)" for the valid values for the *ECB_StackID* field:
3. The MLID must follow the steps given in the previous section of this chapter for the MLID Packet Transmission routine, with the exception that step five must include a check for priority packets and take the appropriate action.

chapter **15** *MLID Timeout Routine*

Chapter Overview

This chapter gives an overview of an MLID Timeout routine. This chapter discusses:

- Scheduling the timeout check
- Determining the wait interval
- Identifying a timeout error,
- Reinitializing the LAN adapter

S

You should review this chapter before writing the MLID timeout routine.

Establishing a Timeout Routine

Depending on the hardware capabilities of the LAN adapter, the MLID might need to establish a timeout check or “dead man timer” that regularly checks the LAN adapter to determine if the LAN adapter is blocked by an unfinished transmission. If a transmission has failed to complete after a reasonable period of time, the timeout procedure should perform the following tasks:

- Reinitialize the LAN adapter
- Increment statistics counters

Scheduling a Timeout Check

You can establish a timer function for your MLID’s timeout check using **CLSL_ScheduleAESEvent**. This call does not create a perpetual timer. **CLSL_ScheduleAESEvent** must be called again to reschedule the next check each time the MLID’s timeout procedure is called.

MLID Timeout Routine **16-1**

Determining the Wait Interval

You might need to experiment with the interval you set between timeout checks to determine the optimal wait interval. This value is affected by the LAN adapter's hardware, the network topology, and the network load. Generally, you will start working with an interval of 1 or 2 seconds.

Identifying a Timeout Error

Immediately after the MLID sends a packet, the send procedure should save a time stamp for later inspection. The timeout procedure compares the time stamp with the current time. If the difference between the two values is less than the established wait interval, the timeout procedure simply reschedules itself. If the wait interval has expired and the LAN adapter is still trying to transmit, a timeout condition has occurred.

Reinitializing the LAN Adapter

After identifying a timeout condition, the MLID should try to reinitialize the LAN adapter without destroying the send event in progress. If the maximum number of retries allowed for the LAN adapter has not been exceeded, the MLID should increment the retry counter and tell the LAN adapter to resend the packet.

If the maximum number of retries has been exceeded, the MLID increments the transmission error statistics counter, resets and clears the transmission bits and the buffers on the adapter, and then sends the next packet on the send queue (if one is waiting to be sent).

16 *MLID Remove Routine*

Chapter Overview

This chapter discusses the MLID Remove procedure. The MLID remove procedure is a routine that allows the operating system to dynamically unload the MLID.

This chapter discusses how the MLID should be shut down. In particular, it covers deregistering, canceling events, shutting down the LAN adapter, and removing the data spaces.

You should read this chapter before you write the MLID timeout and removal routines.

S

Removing the MLID

This routine is called whenever the MLID is unloaded. It gives the MLID a chance to clean up and return resources before being removed.

DeRegistering Logical Boards

When unloading the MLID, the remove procedure must deregister all of the MLID's logical boards with the LSL by calling **CLSL_DeRegisterMLID**. In addition, the remove procedure must deregister the hardware options the MLID registered with the operating system.

MLID Remove Routine **17-1**

Canceling Timeout Check and Polling Routines

The MLID must be sure to cancel any Timeout Check routines it may have scheduled before returning from this procedure by calling **CLSL_CancelAESEvent**. If the MLID's remove procedure does not cancel all AES events, the operating system will try to call the MLID's AES procedure at a memory address that is no longer valid. If the MLID is Polled it needs to deregister the polling routine with the operating system.

Shutting Down the LAN Adapter

The remove procedure must disable the LAN adapter. If the MLID is driving an interrupt-driven LAN adapter or using interrupt backup for a polling procedure, the remove procedure must also deregister the interrupt with the operating system.

Remove Data Spaces

The MLID must also free all the memory that has been allocated for each adapter data space and frame data space. However, the MLID should be careful not to try to remove any adapter data space or frame data space for a logical board that has been completely shut down by the MLID Shutdown control routine.

chapter **17** *MLID Control Routines*

Chapter Overview

This chapter describes the control routines that ODI requires the MLID to provide.

MLID Control Routine Overview

S

The ODI specification requires MLIDs to provide control functions to the LSL for use by protocol stacks and applications. When an MLID registers with the LSL, the LSL passes a pointer to the MLID information block (INFO_BLOCK) for control functions. Applications and protocol stacks use these pointers as entry points to get configuration information and statistics about an MLID (see **CLSL_GetMLIDControlEntry**).

All reserved and unsupported control functions must have pointers in the information block (INFO_BLOCK), which, when called, will return ODISTAT_BAD_COMMAND as the completion code.

MLID Control Routines **18-1**

The following functions are currently defined for these entry points:

AddMulticastAddress
DeleteMulticastAddress
GetMLIDConfiguration
GetMLIDStatistics
GetMulticastInfo
MLIDManagement
MLIDShutdown
MLIDReset
PromiscuousChange
RegisterMonitor
RemoveNetworkInterface
ShutdownNetworkInterface
ResetNetworkInterface
SetLookAheadSize

18-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

The functions above are accessed through the MLID information block (INFO_BLOCK) using indexes. The location of the various MLID control functions in the information block are as follows:

Index	Function
0	GetMLIDConfiguration
1	GetMLIDStatistics
2	AddMulticastAddress
3	DeleteMulticastAddress
4	Reserved
5	MLIDShutdown
6	MLIDReset
7	Reserved
8	Reserved
9	SetLookAheadSize
10	PromiscuousChange
11	RegisterMonitor
12	Reserved
13	Reserved
14	MLIDManagement
15	GetMulticastInfo
16	RemoveNetworkInterface
17	ShutdownNetworkInterface
18	ResetNetworkInterface

Note



Access to the MLID APIs, independent of the link method (dynamic or static), in the information block can be accomplished by using the macro definitions in ODI.H. The macros are listed below. The *infoBlock* parameter is returned by **CLSL_GetMLIDControlEntry**. Refer to the API definitions for details on the rest of the parameters.

```

MLIDCntl_GetConfig (infoBlock, boardNumber, boardConfig, pAsyncECB)
MLIDCntl_GetStats (infoBlock, boardNumber, boardStats, pAsyncECB)
MLIDCntl_AddMulti (infoBlock, boardNumber, addMulticastAddr, pAsyncECB)
MLIDCntl_DelMulti (infoBlock, boardNumber, delMulticastAddr, pAsyncECB)
MLIDCntl_Shutdown (infoBlock, boardNumber, shutdownType, pAsyncECB)
MLIDCntl_Reset (infoBlock, boardNumber, pAsyncECB)
MLIDCntl_PromisChange (infoBlock, boardNumber, PromiscuousState,
                      PromiscuousMode, pAsyncECB)
MLIDCntl_RegMon (infoBlock, boardNumber, txMonRoputine, pAsyncECB,
                monitorState)
MLIDCntl_Management (infoBlock, boardNumber, managementECB)
MLIDCntl_GetMulticastInfo (infoBlock, boardNumber, multicastinfoECB)
MLIDCntl_RemoveNetworkInterface (infoBlock, boardNumber, pAsyncECB)
MLIDCntl_ShutdownNetworkInterface (infoBlock, boardNumber, pAsyncECB)
MLIDCntl_ResetNetworkInterface (infoBlock, boardNumber, pAsyncECB)

```

MLID Control Routines 18-3

Adhere to the following procedures for MLID control services that complete asynchronously and for which an asynchronous ECB has been provided.

If the MLID can not finish the request in a reasonable amount of time (less than a millisecond), it does the following:

1. Queue the provided asynchronous ECB.
2. Return *ODISTAT_RESPONSE_DELAYED*.
3. Start servicing requests and schedule an AES event instead of polling hardware for completion.

If another application calls the same MLID control service with an ECB, before the MLID has finished with the first one, the MLID should delay until the first request has completed and then initiate the request.

Note



If an ECB has been provided, the ESR is only called if *ODISTAT_RESPONSE_DELAYED* is returned.

MLID Control Service Completed

For the following control services:

- GetMLIDConfiguration
- GetMLIDStatistics
- GetMLIDMulticastInfo
- MLIDShutdown
- MLIDReset
- AddMulticastAddress
- DeleteMulticastAddress
- SetLookAheadSize
- PromiscuousChange
- RegisterMonitor
- MLID Management
- RemoveNetworkInterface
- ShutdownNetworkInterface
- ResetNetworkInterface

MLID Control Routines **18-5**

The MLID does the following for each ECB queued for this control service.

1. Unlink the ECB from the MLID queue.
2. Fill in the *ECB_Status* field with the appropriate *ODISTAT* return status.
3. If **GetMLIDConfiguration** is used:

Fill the *ECB_PreviousLink* field with the pointer to the MLID configuration table; NULL if error.

If **GetMLIDStatistics** is used:

Fill the *ECB_PreviousLink* field with the pointer to the MLID statistics table; NULL if error.

If *PromiscuousChange* is used:

Fill the *ECB_PreviousLink* field with the current promiscuous mode.

4. Call the ECB's Event Service Routine (**(*ECB_ESR) (ECB*)**).

AddMulticastAddress

Index 2 (0x02)

Adds the specified node address to the multicast (group) address table.

Syntax

```
#include <odi.h>

ODISTAT AddMulticastAddress (
    UINT32 BoardNumber,
    NODE_ADDR *AddMulticastAddr,
    ECB *pAsyncECB );
```

Input Parameters

BoardNumber

The board number, which indicates the MLID to add the multicast (group) address for.

AddMulticastAddr

Pointer to an *ADDR_SIZE* size byte area, which contains the multicast (group) address. *ADDR_SIZE* is defined in ODI.H.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is to be called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

MLID Control Routines **18-7**

Return Values

ODISTAT_SUCCESSFUL	Multicasting was successfully enabled.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_OUT_OF_RESOURCES	The MLID has insufficient resources to enable the multicast (group) address.
ODISTAT_BAD_PARAMETER	The requested multicast (group) address is not valid for the MLID's media type, or the specified board number is invalid.
ODISTAT_BAD_COMMAND	Multicast (group) addressing is not supported by the MLID and/or the underlying hardware device.
ODISTAT_MLID_SHUTDOWN	The MLID is temporarily shutdown.

Remarks

Protocol stacks that enable multicast reception should first check *MM_MULTICAST_BIT* in the MLID configuration table's *MLIDCFG_ModeFlags* field. Some LAN media (for example, RX-Net) do not support multicasting. When an underlying MLID/adaptor does not support multicasting, the protocol stack should use broadcast transmission instead. (The destination address is 0xFF FF FF FF FF FF for a broadcast transmission.)

The MLID keeps a count of the number of times the specified address is added. When the address is deleted by **DeleteMulticastAddress**, the count is decremented. When the count is 0, the address is disabled. This behavior allows two or more protocols to safely use the same multicast (group) address.

The MLID manages enabled multicast (group) addresses according to the physical adapter. The format of a multicast (group) address is LAN medium dependent. The two most common formats are for Ethernet (Ethernet_II/IEEE 802.3) and Token-Ring (802.5), which are summarized below. Proprietary LAN media that support multicasting can have alternate address encoding methods. Therefore, a protocol stack should allow a multicast (group) address that can be configured by the user. This allows the protocol stack to work correctly on proprietary LAN media.



unctional addresses should never be sent on the medium with more than one function bit set. If more than one function bit is set, the address will not work on all media. For example, Token-Ring accepts a functional address that has more than one function bit set but PCN_II does not.



f a Token-Ring based adapter is operating in canonical mode (that is, bit 14 is clear and bit 15 is set in the *MLIDCFG_ModeFlags* field), the Token-Ring MLID accepts functional addresses in canonical format, that is, instead of C0-00, it accepts 03-00

Maximum Number of Multicast (Group) Addresses

The number of multicast (group) addresses supported by an underlying MLID/LAN medium is not specified by ODI. In the case of Token-Ring, the maximum number supported is specified by the definition of the address, with 28 being the maximum. Ethernet, however, has an almost infinite number of functional addresses. The maximum number supported is defined in ODI.H as MAXMULTICASTS.

Group Addressing Hardware

Most adapters for Ethernet, FDDI, etc. create a multicast (group) address hash table to filter incoming packets destined to a multicast group. Hashing is not usually a guaranteed filter; therefore, more than one multicast (group) address might be received by the adapter. This causes the underlying MLID to receive unwanted multicast packets. The MLID will complete the filtering so that only addresses enabled through this command are actually passed to protocol stacks.

A class of adapters have been developed that contain specialized hardware to support multicast (group) addressing—for example, Content Addressable Memory (CAM) memory, which ensures that only supported multicast (group) addresses get passed up from the hardware to the software of the MLID for processing. The MLID configuration table *MF_SOFT_GRP_BIT* and *MF_MF_GRP_ADDR_SUP_BIT* bits of the *MLIDCFG_Flags* field allow for such adapters. They allow for this hardware support to be recognized and for it to indicate when multicast (group) addresses need to be checked by software. See *Chapter 12: MLID Data Structures* for the MLID's configuration table and a discussion of the format of these bits.

Examples

MLID Control Routines 18-9

Ethernet Multicasts

Ethernet multicast (group) addresses must have bit 0 of byte 0 set to 1 (for example, x1 xx xx xx xx xx). The address is value based; each value is unique and separate from other values.

Most Ethernet adapters create a multicast hash table to filter incoming packets destined to a multicast group. Hashing is not usually a guaranteed filter; therefore, more than one multicast (group) address might be received by the adapter. This causes the underlying MLID to receive unwanted multicast packets. The MLID will complete the filtering so that only addresses enabled through this command are actually passed to protocol stacks.

Token-Ring Multicasts

Token-Ring multicast addresses in an ODI system can be Token-Ring functional addresses or group addresses as defined by the Token-Ring topology. These addresses are bit based. Each bit position in the address signifies a unique address, and more than one address can be specified by simply setting multiple bits. Addresses always begin with C0-00, which leaves 32 bits (4 bytes) for functional addresses. However, four of the 32 possible bits are reserved by IBM, which leaves 28 unique multicast (group) addresses available.

More than one multicast (group) address can be added when you invoke the **AddMulticastAddress** command. For example, if C0 00 00 01 00 00 and C0 00 00 02 00 00 need to be enabled, **AddMulticastAddress** can be called twice (once for each address) or simply called once with C0 00 00 03 00 00. Both methods are equivalent. Token-Ring MLIDs keep a use count for each functional address bit.

Note



The media specification produced by the standards committees should be referenced for complete descriptions on multicast addresses.

DeleteMulticastAddress

Index 3 (0x03)

Disables reception of a previously enabled multicast (group) address.

Syntax

```
#include <odi.h>

ODISTAT DeleteMulticastAddress (
    UINT32 BoardNumber,
    NODE_ADDR *DelMulticastAddr,
    ECB *pAsyncECB );
```

Input Parameters

BoardNumber

The board number describing which MLID to delete the multicast (group) address for.

DelMulticastAddr

Pointer to an *ADDR_SIZE* size byte area containing the multicast (group) address. *ADDR_SIZE* is defined in ODI.H.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

MLID Control Routines **18-11**

Return Values

ODISTAT_SUCCESSFUL	One instance of the address was successfully deleted.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_BAD_PARAMETER	The requested multicast (group) address is not valid for the MLID's media type, or the specified board number is invalid.
ODISTAT_ITEM_NOT_PRESENT	The specified address is not presently enabled in the MLID.
ODISTAT_BAD_COMMAND	Multicast (group) addressing is not supported by the MLID and/or the underlying hardware device.
ODISTAT_MLID_SHUTDOWN	The MLID is temporarily shutdown.

Remarks

This command decrements the MLID's use count for the specified address. When the use count becomes 0, response of that address is disabled. (See **AddMulticastAddress** for a discussion of multicast (group) address formats.)

GetMLIDConfiguration

Index 0 (0x00)

Returns a pointer to a pointer to the MLID configuration table for the specified logical board.

Syntax

```
#include <odi.h>

ODISTAT GetMLIDConfiguration (
    UINT32 BoardNumber,
    MLID_CONFIG_TABLE **BoardConfig,
    ECB *pAsyncECB);
```

Input Parameters

BoardNumber

The board number to obtain the MLID configuration table for.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

BoardConfig

Pointer to a buffer where a pointer to the specified board's MLID configuration table can be returned.

MLID Control Routines **18-13**

Return Values

ODISTAT_SUCCESSFUL	The pointer to the specified board's MLID configuration table was successfully returned.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not be completed in a reasonable amount of time and will complete asynchronously.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.

Remarks

This command is supported by all MLIDs. A separate configuration table is maintained by the MLID for each adapter and frame type combination. (See *Chapter 12: MLID Data Structures* for the format of the MLID configuration table.)

This control service usually completes synchronously.

GetMLIDStatistics

Index 1 (0x01)

Returns a pointer to a pointer to the MLID statistics table for the specified board.

Syntax

```
#include <odi.h>

ODISTAT GetMLIDStatistics (
    UINT32 BoardNumber,
    MLID_STATS_TABLE **BoardStats,
    ECB *pAsyncECB );
```

Input Parameters

BoardNumber

The board number to obtain the MLID statistics table for.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

BoardStats

Pointer to a buffer where a pointer to the specified board's statistics table can be returned.

MLID Control Routines **18-15**

Return Values

ODISTAT_SUCCESSFUL	The pointer to the specified board's MLID statistics table was successfully returned.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not be completed in a reasonable amount of time and will complete asynchronously.
ODISTAT_MLID_SHUTDOWN	The MLID is temporarily shutdown.

Remarks

All MLIDs support this command. The MLID maintains one statistics table for each physical adapter. Each frame type (logical board) present for that physical adapter uses the same table. The board number can be any of the logical board values present for a physical adapter. Regardless of the logical board number, **GetMLIDStatistics** returns the same table for each logical board associated with the MLID. (See *Chapter 12: MLID Data Structures* for the format of the MLID statistics table.)

Usually, this control service completes synchronously.

GetMulticastInfo

Index 15 (0x0F)

Allows management entities to get group addresses or functional addresses that the MLID is using.

Syntax

```
#include <odi.h>
ODISTAT GetMulticastInfo (
    UINT32 BoardNumber,
    ECB *MulticastInfoECB );
```

Input Parameters

BoardNumber

The logical board number.

Output Parameters

MulticastInfoECB

Pointer to an ECB in which to return Group (Multicast) and/or Functional address information. The MulticastInfoECB contains a single fragment into which to copy Group address (Multicast) information:

On Entry:

ECB_FragmentCount

Is equal to 1.

ECB_Fragment.FragmentAddress

Points to the buffer to copy Group address structures.

ECB_Fragment.FragmentLength

The size of the buffer to copy Group address structures.

MLID Control Routines **18-17**

On Exit:

ECB_ImmediateAddress

Contains the Functional address currently used by the MLID, zero if the MLID does not use Functional Addresses (eg. Ethernet, FDDI), or the Function Address is currently unused.

ECB_DataLength

Contains the number of bytes transferred into the buffer indicated by *ECB_Fragment.FragmentAddress*. If the MLID does not use Group addresses the *ECB_DataLength* field will be set to zero.

This field will contain the size of the buffer required to return the Multicast address information, if the buffer is insufficient in size, that is when *ODISTAT_OUT_OF_RESOURCES* is returned.

ECB_Fragment.FragmentAddress

Points to a buffer containing a series of Group address structures used by the MLID. If the MLID does not use Group addresses the *ECB_DataLength* field will be set to zero. The MLID returns in the buffer a series of repeating structures defined as follows:

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

GRP_ADDR
    Multicast address in the MLID configuration
    table. A value of zero indicates an entry
    that has never been used.

GRP_ADDR_COUNT
    This count is the number of times the address
    has been added. A value of zero indicates
    that the address is currently inactive.
```

ECB_Fragment.FragmentLength

Unchanged.

Return Values

ODISTAT_SUCCESSFUL	The command was successfully executed. The ECB is returned to the caller.
ODISTAT_RESPONSE_DELAYED	The requested operation was successfully started, but will complete asynchronously. The MulticastInfoECB event service routine will be called with the completion code when the command has finished execution.
ODISTAT_BAD_COMMAND	GetMulticastInfo functions are not supported by the MLID.
ODISTAT_OUT_OF_RESOURCES	The buffer presented in MulticastInfoECB to transfer the Group (Multicast) addresses used by the MLID into was insufficient. The field MulticastInfoECB.ECB_DataLength contains the size the buffer should be to transfer Group (Multicast) addresses used by the MLID.
ODISTAT_MLID_SHUTDOWN	The MLID is temporarily shutdown.

Remarks

The MulticastInfoECB ESR ((***ECB_ESR**)(**ECB***)) will only be called if the function returns ODISTAT_RESPONSE_DELAYED, any other return code implies that the processing of the ECB is complete and the ECB contains valid information, eg. ODISTAT_OUT_OF_RESOURCES returned implies ECB_DataLength contains the size of the buffer required to return appropriate information.

If this function is not implemented, a function which returns ODISTAT_BAD_COMMAND must be placed here in lieu of the **GetMulticastInfo** function.

The list of Group (Multicast) address returned are in canonical/noncanonical format as described by the MLID Configuration *MLID_ModeFlags* fields's bit 14 and 15.

MLID Control Routines 18-19

MLIDManagement

Index 14 (0x0E)

Allows various management entities to access management information—for example, hub management, SMT management for FDDI, protocol stacks off-loading routing duties to intelligent adapters, etc.

Syntax

```
#include <odi.h>

ODISTAT MLIDManagement (
    UINT32 BoardNumber,
    ECB *ManagementECB );
```

Input Parameters

BoardNumber

The logical board number.

ManagementECB

Pointer to an ECB containing management information. The first byte of the *ECB_ProtocolID* field is greater than 0x40 and less than 0x7F.

Output Parameters

None.

18-20 ODI Specification: Protocol Stacks and MLIDs (C Language)

Return Values

ODISTAT_SUCCESSFUL	Command was executed successfully.
ODISTAT_RESPONSE_ DELAYED	Command will complete asynchronously. The management ECB event service routine will be called with the completion code when the command has finished execution.
ODISTAT_BAD_COMMAND	Management functions are not supported by the MLID.
ODISTAT_BAD_PARAMETER	First byte of the <i>ECB_ProtocolID</i> field containing the management handle provided in the management ECB is not greater than 0x40 and less than 0x7F, or the specified board number is invalid.
ODISTAT_NO_SUCH_HANDLER	Management entity for the management handle provided in the management <i>ECB_ProtocolID</i> field does not exist.
ODISTAT_FAIL	The command was recognized, but could not be executed.

Remarks

The MLID management ECB ESR ((***ECB_ESR**)(**ECB***)) could be called before returning from this function.

MLIDReset

Index 6 (0x06)

Causes the MLID to totally reinitialize the physical adapter. This command also brings an MLID back into active operation if it was temporarily shut down.

Syntax

```
#include <odi.h>

ODISTAT MLIDReset (
    UINT32 BoardNumber,
    ECB *pAsyncECB );
```

Input Parameters

BoardNumber

The logical board number.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

18-22 ODI Specification: Protocol Stacks and MLIDs (C Language)

Return Values

ODISTAT_SUCCESSFUL	The physical card has been reactivated.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_FAIL	The MLID was unable to reset its hardware. This can indicate a hardware failure or system corruption.
ODISTAT_BAD_COMMAND	The MLID does not support this command.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.

Remarks

The *MS_SHUTDOWN_BIT* bit in each logical board's configuration table *MLIDCFG_SharingFlags* field will be reset to 0 when this function returns successfully.

This function leaves previously enabled multicast (group) addresses enabled.

MLIDShutdown

Index 5 (0x05)

Allows an application to shut down a physical adapter.

Syntax

```
#include <odi.h>

ODISTAT MLIDShutdown (
    UINT32 BoardNumber,
    UINT32 ShutDownType,
    ECB *pAsyncECB);
```

Input Parameters

BoardNumber

The logical board number.

ShutDownType

The form of shut down desired:

SHUTDOWN_PERMANENT

Shut down hardware and deregister with the LSL (permanent shutdown).

SHUTDOWN_PARTIAL

Shut down hardware only (temporary shutdown).

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The MLID was successfully shutdown.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_FAIL	The MLID was unable to shut down its hardware. This can indicate a hardware failure or system corruption.
ODISTAT_BAD_COMMAND	The MLID does not support this command.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.

Remarks

If the MLID is permanently shutdown, a subsequent call to **MLIDReset** will not be successful. Permanent shutdowns are normally used only to completely disable the hardware and return any interrupt vectors and system resources.

MLIDs that are temporarily shutdown can be brought back into operation by invoking the **MLIDReset** control command.

All the adapter's logical boards that share the same physical adapter and are represented by the logical board number in **MLIDCFG_BoardNumber** are affected by this command. All logical board MLID configuration tables that are affected by this command will have *MS_SHUTDOWN_BIT* set in the *MLIDCFG_SharingFlags* field.

Any outstanding protocol transmit and receive ECBs will be returned before this command is completed.

PromiscuousChange

Index 10 (0x0A)

Used by protocol stacks to enable or disable promiscuous mode on the MLID's adapter.

Syntax

```
#include <odi.h>

ODISTAT PromiscuousChange (
    UINT32 BoardNumber,
    UINT32 PromiscuousState,
    UINT32 *PromiscuousMode,
    ECB *pAsyncECB );
```

Input Parameters

BoardNumber

The logical board number.

PromiscuousState

Promiscuous state is set or reset according to the following values:

Prom_State_Off	Disable promiscuous mode
Prom_State_On	Enable promiscuous mode

PromiscuousMode

Pointer to the promiscuous mode mask, which indicates the type of frames the MLID is to promiscuously receive. The following defines the promiscuous mode mask:

PROM_MODE_QUERY	Query as to promiscuous mode
PROM_MODE_MAC	MAC frames
PROM_MODE_NON_MAC	NonMAC frames
PROM_MODE_MACANDNON	Both MAC & nonMAC frames
PROM_MODE_SMT	All Station Management Frames SMT for FDDI
PROM_MODE_RMC	Remote multicast frames are to be received.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is to be called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters*PromiscuousMode*

Pointer to the current promiscuous mode mask for the MLID at the completion of this function. See *PromiscuousMode* above for definitions of the promiscuous mode mask values.

Return Values

ODISTAT_SUCCESSFUL	Command was executed successfully.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_BAD_COMMAND	Promiscuous mode is not supported by the MLID.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.
ODISTAT_NO_SUCH_DRIVER	This usually means the driver is temporarily shutdown.

Remarks

A protocol stack can enable promiscuous mode multiple times without error. The current value of *PromiscuousMode* determines the type of frames that are promiscuously received. *PromiscuousState* determines whether promiscuous mode is enabled or disabled, unaffected the current *PromiscuousMode* settings. If the LAN medium or adapter does not distinguish between MAC and nonMAC frames (for example, Ethernet does not differentiate between MAC or nonMAC frames), both MAC and nonMAC frames are assumed for the *PromiscuousMode* mask.

Setting the Remote Multicast Frames bit causes the MLID to activate all multicast frame reception. For example, if an adapter utilizes a hash table for filtering active multicast frame, then the adapter sets the hash table to accept all multicast frames. Filtering of active multicast entries will be disabled while this bit is set. MLIDS that can filter must also disabled filtering while this bit is set. Protocols that enable the MLID to receive remote multicast frames must also remember to set the RxPkt_RemoteMulticast_Bit in their stack filter to receive remote multicast frames.

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

The driver must keep a counter for each promiscuous mode. Each time it is enabled, the counter(s) should be incremented, and each time it is disabled, they should be decremented. When the counters reach 0, promiscuous mode should be disabled on the adapter.

All packets are presented to the LSL.



MLIDs that support promiscuous mode set the *MM_PROMISCUOUS_BIT* in the *MLIDCFG_ModeFlags* field of the MLID's configuration table. Pad bytes are passed up.

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

All adapters that have promiscuous mode and raw send ability should not affect the source address of a raw send transmit when in promiscuous mode; in other words, you should not insert the adapters node address as the source address in the media layer header but transmit the provided media layer header unaltered.

MLID Control Routines **18-29**

RegisterMonitor

Index 11 (0x0B)

Invoked by protocol stacks to monitor the packets the adapter is transmitting.

Syntax

```
#include <odi.h>

ODISTAT RegisterMonitor (
    UINT32 BoardNumber,
    void (*TxMonRoutine)(CTCB*),
    ECB *pAsyncECB,
    BOOLEAN MonitorState );
```

Input Parameters

BoardNumber

The logical board number.

TxMonRoutine

Pointer to the transmit monitor routine.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

MonitorState

Boolean value to enable or disable the monitor transmit routine.

TRUE Enable monitor transmit routine.

FALSE Disable monitor transmit routine.

18-30 ODI Specification: Protocol Stacks and MLIDs (C Language)

Output Parameters

None.

Return Values

ODISTAT_SUCCESSFUL	The commands completed successfully.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_OUT_OF_RESOURCES	The monitor is already attached.
ODISTAT_BAD_COMMAND	This function is not supported.
ODISTAT_NO_SUCH_HANDLER	The value of the pointer to TxMonRoutine from an attempt to disable the monitor differs from that used when enabling the monitor transmit routine. In other words, the value for TxMonRoutine ensures that only the entity that enabled the monitor can disable it.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.
ODISTAT_MLID_SHUTDOWN	The MLID is temporarily shutdown.

Remarks

If a transmit monitor is registered, packets are passed to the monitor regardless of whether the MLID is in promiscuous mode or not. In other words, promiscuous mode has no effect on a transmit monitor.

The transmit monitor is registered on a per logical board basis.

Transmit Monitor

The transmit monitor is passed a pointer to a CTCB . The transmit monitor can copy part or all of the packet described by the CTCB but cannot modify it.

Syntax

```
void TxMonRoutine(CTCB *TxMon_CTCB);
```

Input Parameters

TxMon_CTCB

Pointer to a CTCB.

Output Parameters

None.

Return Values

None.

Remarks

The transmit monitor differs from the prescan transmit chain stack in that the transmit monitor presents the MAC layer header generated for the transmit packet to whoever is registered as a transmit monitor. A prescan transmit chain stack is only presented with the data that is to be transmitted.

CTCB Structure

```

typedef struct _CTCB_FRAGMENT_BLOCKSTRUCT_
{
    UINT32 CTCB_FragmentCount;
    FRAGMENT_STRUCT CTCB_Fragment[1];
}CTCB_FRAGMENT_BLOCK;

typedef struct _CTCB_
{
    void *CTCB_Reserved;
    UINT32 CTCB_BoardNumber;
    UINT32 CTCB_DriverWS[3];
    UINT32 CTCB_DataLen;
    CTCB_FRAGMENT_STRUCT *CTCB_FragBlockPtr;
    UINT32 CTCB_MediaHeaderLen;
    UINT8 CTCB_MediaHeader[MAX_MEDIA_HEADER_SIZE];
}CTCB;

```

The CTCB will always contain logical addresses when presented to the transmit monitor.

RemoveNetworkInterface

Index 16 (0X10)

Allows an application to remove (unload) a logical board.

Syntax

```
#include <odi.h>

ODISTAT RemoveNetworkInterface
    (UINT32 boardNumber,
     ECB *pAsyncECB );
```

Input Parameters

boardNumber

The logical board number.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns ODISTAT_RESPONSE_DELAYED). If NULL, it is assumed that the caller does not need to be informed of the completion of the request; however, ODISTAT_RESPONSE_DELAYED will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

18-34 ODI Specification: Protocol Stacks and MLIDs (C Language)

Return Values

ODISTAT_SUCCESSFUL	The logical board was successfully removed.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will be completed asynchronously.
ODISTAT_FAIL	The MLID was unable to remove the logical board.
ODISTAT_BAD_COMMAND	The MLID does not support this command.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.

Remarks

This function permanently removes the logical board associated with *boardNumber*.

ResetNetworkInterface

Index 18 (0X12)

Allows an application to reset a logical board.

Syntax

```
#include <odi.h>

ODISTAT ResetNetworkInterface
    (UINT32 boardNumber,
     ECB *pAsyncECB );
```

Input Parameters

boardNumber

The logical board number.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns ODISTAT_RESPONSE_DELAYED). If NULL, it is assumed that the caller does not need to be informed of the completion of the request; however, ODISTAT_RESPONSE_DELAYED will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

18-36 ODI Specification: Protocol Stacks and MLIDs (C Language)

Return Values

ODISTAT_SUCCESSFUL	The logical board was successfully reset.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will be completed asynchronously.
ODISTAT_FAIL	The MLID was unable to reset the logical board.
ODISTAT_BAD_COMMAND	The MLID does not support this command.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.

Remarks

This function resets the logical board associated with *boardNumber*. For most LAN drivers this is a NOP.

SetLookAheadSize

Index 9 (0x09)

Tells the MLID the amount of lookahead data that is needed by the caller to properly process received packets.

Syntax

```
#include <odi.h>

ODISTAT SetLookAheadSize (
    UINT32 BoardNumber,
    UINT32 RequestedSize,
    ECB *pAsyncECB);
```

Input Parameters

BoardNumber

The logical board number.

RequestedSize

The requested lookahead size.

pAsyncECB

Pointer to an ECB whose ESR (**(*ECB_ESR)(ECB*)**) is to be called if the MLID control service performs asynchronously (returns *ODISTAT_RESPONSE_DELAYED*). If NULL, it is assumed that the caller does not need to be informed of completion of the request; however, *ODISTAT_RESPONSE_DELAYED* will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

18-38 ODI Specification: Protocol Stacks and MLIDs (C Language)

Return Values

ODISTAT_SUCCESSFUL	The lookahead size was updated.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will complete asynchronously.
ODISTAT_BAD_PARAMETER	The requested lookahead size exceeded bounds or the specified board number is invalid.
ODISTAT_MLID_SHUTDOWN	The MLID is temporarily shutdown.

Remarks

As part of a protocol stack's initialization, this function should be invoked to properly configure the MLID specified in **MLIDCFG_BoardNumber** for the amount of lookahead data a protocol stack needs for packet reception. The lookahead data value must be between 0 and 128 bytes. If the requested size is less than the MLID's current lookahead size value, the MLID will use the larger value. In other words, it is impossible to adjust the size downward.

ShutdownNetworkInterface

Index 17 (0X11)

Allows an application to perform a partial shutdown of a logical board.

Syntax

```
#include <odi.h>

ODISTAT ShutdownNetworkInterface
    (UINT32 boardNumber,
     ECB *pAsyncECB );
```

Input Parameters

boardNumber

The logical board number.

pAsyncECB

Pointer to an ECB whose ESR ((***ECB_ESR**)(**ECB***)) is called if the MLID control service performs asynchronously (returns ODISTAT_RESPONSE_DELAYED). If NULL, it is assumed that the caller does not need to be informed of the completion of the request; however, ODISTAT_RESPONSE_DELAYED will still be returned if the MLID control service is performed asynchronously.

Output Parameters

None.

18-40 ODI Specification: Protocol Stacks and MLIDs (C Language)

Return Values

ODISTAT_SUCCESSFUL	The logical board was successfully shutdown.
ODISTAT_RESPONSE_DELAYED	The MLID control service could not complete in a reasonable amount of time and will be completed asynchronously.
ODISTAT_FAIL	The MLID was unable to shutdown the logical board.
ODISTAT_BAD_COMMAND	The MLID does not support this command.
ODISTAT_BAD_PARAMETER	The specified board number is invalid.

Remarks

This function performs a partial shutdown of the logical board associated with *boardNumber*.

18-42 ODI Specification: Protocol Stacks and MLIDs (C Language)

Appendix **A** *Event Control Blocks (ECBs)*

Appendix 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.

Event Control Blocks

The ODI system uses Event Control Blocks (ECBs) for two purposes:

- To describe the protocol data during packet transmission
- To describe the protocol buffers during packet reception

The format of the ECB is the same regardless of whether it is a send or a receive ECB.

This appendix includes the ECB structure in sample code and a table that describes the fields of the ECB.

Event Control Block Structure Sample Code

```
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
{
    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 A-1
Fragment Structure and ECB Field Descriptions

Name	Description
FragmentAddress	This field specifies a pointer to a data buffer of <i>FragmentLength</i> bytes.
FragmentLength	This field specifies the 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.

A-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description																		
ECB_NextLink	This field is typically used as a forward link to a list of ECBs. The current owner of the ECB (the protocol stack, in this case) uses this field.																		
ECB_PreviousLink	This field is typically used as a back link to manage a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from an MLID containing a received packet, this field contains the received packet error status defined as follows:																		
	<table> <tr> <th>Bit Value</th><th>Description</th></tr> <tr> <td>PAE_CRC_BIT</td><td>CRC error (for example, Frame Check Sequence (FCS) error).</td></tr> <tr> <td>PAE_CRC_ALIGN_BIT</td><td>CRC / Frame Alignment error.</td></tr> <tr> <td>PAE_RUNT_PACKET_BIT</td><td>Runt packet.</td></tr> <tr> <td>PAE_TOO_BIG_BIT</td><td>Packet larger than allowed by media.</td></tr> <tr> <td>PAE_NOT_ENABLED_BIT</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>PAE_MALFORMED_BIT</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>PAE_NO_COMPRES_BIT</td><td>Do not decompress the received packet.</td></tr> <tr> <td>PAE_NONCAN_ADDR_BIT</td><td>The Address present in <i>ECB_ImmediateAddress</i> is in noncanonical format.</td></tr> </table>	Bit Value	Description	PAE_CRC_BIT	CRC error (for example, Frame Check Sequence (FCS) error).	PAE_CRC_ALIGN_BIT	CRC / Frame Alignment error.	PAE_RUNT_PACKET_BIT	Runt packet.	PAE_TOO_BIG_BIT	Packet larger than allowed by media.	PAE_NOT_ENABLED_BIT	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.	PAE_MALFORMED_BIT	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.	PAE_NO_COMPRES_BIT	Do not decompress the received packet.	PAE_NONCAN_ADDR_BIT	The Address present in <i>ECB_ImmediateAddress</i> is in noncanonical format.
Bit Value	Description																		
PAE_CRC_BIT	CRC error (for example, Frame Check Sequence (FCS) error).																		
PAE_CRC_ALIGN_BIT	CRC / Frame Alignment error.																		
PAE_RUNT_PACKET_BIT	Runt packet.																		
PAE_TOO_BIG_BIT	Packet larger than allowed by media.																		
PAE_NOT_ENABLED_BIT	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.																		
PAE_MALFORMED_BIT	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.																		
PAE_NO_COMPRES_BIT	Do not decompress the received packet.																		
PAE_NONCAN_ADDR_BIT	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.																		

Event Control Blocks (ECBs) **A-3**

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description										
ECB_Status	<p>This field indicates the 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> <table><tr><td>ODISTAT_SUCCESSFUL</td><td>Packet was received successfully.</td></tr><tr><td>ODISTAT_RX_OVERFLOW</td><td>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.</td></tr><tr><td>ODISTAT_CANCELED</td><td>The ECB was not needed by the MLID. The MLID signals to the protocol stack that the ECB was not transmitted.</td></tr><tr><td>ODISTAT_MLID_SHUTDOWN</td><td>The LAN adapter specified in the <i>ECB_BoardNumber</i> field cannot be found. This usually means that the MLID has been shut down (temporarily or permanently).</td></tr><tr><td>ODISTAT_BAD_PARAMETER</td><td>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.</td></tr></table> <p>Note: The return values are ODISTAT cast as UINT16.</p>	ODISTAT_SUCCESSFUL	Packet was received successfully.	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.	ODISTAT_CANCELED	The ECB was not needed by the MLID. The MLID signals to the protocol stack that the ECB was not transmitted.	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 shut down (temporarily or permanently).	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.
ODISTAT_SUCCESSFUL	Packet was received successfully.										
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.										
ODISTAT_CANCELED	The ECB was not needed by the MLID. The MLID signals to the protocol stack that the ECB was not transmitted.										
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 shut down (temporarily or permanently).										
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.										
ECB_ESR	The protocol stack sets this field to point to an appropriate routine that is to be called when the send or receive event is complete (either successfully or with an error). This field must point to a valid handler ((* ECB_ESR)(ECB*)).										

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description		
ECB_StackID	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.		
	The following values are valid for the <i>ECB_StackID</i> field.		
RAW_SEND_PRIORITY_0	0xFFFF	0 = No Priority	
RAW_SEND_PRIORITY_1	0xFFFE	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	7 = Highest Priority	
SEND_PRIORITY_0	0xFFFF7	0 = No Priority	
SEND_PRIORITY_1	0xFFFF6	1 = Lowest 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	7 = Highest Priority	

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description
ECB_ProtocolID	<p>This field contains the Protocol ID (PID) value for sends and receives. If the ECB is a send ECB, the protocol stack sets this field before calling SendPacket. In a send ECB, 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 fills in this field with the protocol ID supplied in the LOOKAHEAD structure. The PID is stored in high-low order.</p>
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 that board number. On sends, a protocol stack fills in this field to indicate the target logical board.</p> <p>For receive ECBs, the protocol stack fills in this field with the board number supplied in the LOOKAHEAD structure.</p>
ECB_ImmediateAddress	<p>If the ECB is a send ECB, the protocol stack sets this field before calling SendPacket. 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>

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description												
ECB_DriverWorkspace	<p>This field is generally reserved for use by the MLID. Protocol stacks should not modify this field unless the protocol stack currently owns the ECB.</p> <p>The first byte, offset 0 of the <i>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>Value</th><th>Description</th></tr><tr><td>ECB_MULTICAST</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>ECB_BROADCAST</td><td>Broadcast: The packet was destined to all nodes on the physical network. Note: 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>ECB_UNICASTREMOTE</td><td>UnicastRemote: The packet was directly destined to another workstation on the physical network. Note, this bit set generally only occurs after the MLID has been entered into promiscuous mode or has received a packet due to source routing.</td></tr><tr><td>ECB_MULTICASTREMOTE</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. Generally, this bit is set only after the MLID has been entered into promiscuous mode.</td></tr><tr><td>ECB_SOURCE_ROUTE</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></table>	Value	Description	ECB_MULTICAST	Multicast: The packet was destined to a subset of group addresses on the physical network that the MLID has been programmed to support.	ECB_BROADCAST	Broadcast: The packet was destined to all nodes on the physical network. Note: on receiving a broadcast both b0 and b1 are set to 1, since a broadcast address is also a group address.	ECB_UNICASTREMOTE	UnicastRemote: The packet was directly destined to another workstation on the physical network. Note, this bit set generally only occurs after the MLID has been entered into promiscuous mode or has received a packet due to source routing.	ECB_MULTICASTREMOTE	MulticastRemote: The packet was destined to a subset of group addresses on the physical network that the MLID has not been programmed to support. Generally, this bit is set only after the MLID has been entered into promiscuous mode.	ECB_SOURCE_ROUTE	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.
Value	Description												
ECB_MULTICAST	Multicast: The packet was destined to a subset of group addresses on the physical network that the MLID has been programmed to support.												
ECB_BROADCAST	Broadcast: The packet was destined to all nodes on the physical network. Note: on receiving a broadcast both b0 and b1 are set to 1, since a broadcast address is also a group address.												
ECB_UNICASTREMOTE	UnicastRemote: The packet was directly destined to another workstation on the physical network. Note, this bit set generally only occurs after the MLID has been entered into promiscuous mode or has received a packet due to source routing.												
ECB_MULTICASTREMOTE	MulticastRemote: The packet was destined to a subset of group addresses on the physical network that the MLID has not been programmed to support. Generally, this bit is set only after the MLID has been entered into promiscuous mode.												
ECB_SOURCE_ROUTE	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.												

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description
ECB_GLOBALERROR	GlobalError: The packet contains errors. See the <i>ECB_PreviousLink</i> field as to specific error. This is an exclusive bit, if set all other bits should be 0. This value supersedes <i>SourceRoute</i> .
ECB_MACFRAME	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 0. Note: MAC frames by definition are not source routable.
ECB_UNICASTDIRECT	Direct: The packet was destined to this station only.
<hr/>	
The MLID supports 802.2 Type II. The number of control bytes present in the 802.2 header is presented in the second byte, for example, offset 1, of the <i>DriverWorkspace</i> field:	
Bit Value	Description
ECB_TYPE_I	1 control byte is present in the 802.2 header
ECB_TYPE_II	2 control bytes are present in the 802.2 header
ECB_RX_PRIORITY	RxPriorityFrame: The received packet is a priority packet. This is only valid for topologies that support a distinction in priority levels. When this bit is set the <i>LKAhd_PriorityLevel</i> field in the LOOKAHEAD structure contains the priority level. If the protocol stack needs the priority level when the ECB is returned it must save it at LOOKAHEAD time. This bit is not set if the received frame is at the normal priority level or lower.
If ECB_TYPE_I or ECB_TYPE_II is not set, no 802.2 header is in the frame.	
The second word, offsets 2 and 3 for example, of the <i>DriverWorkspace</i> field are filled with the size of the received frame minus the MAC header, for example, the total number of data bytes present in the frame.	
ECB_ProtocolWorkspace	This field is reserved for use by the originating protocol stack and must not be modified by the LSL or the MLIDs.

Table A-1
Fragment Structure and ECB Field Descriptions

Name	Description
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	This field contains the number of fragment buffer descriptors immediately following this field. This value cannot be larger than 16, for example, range $0 < ECB_FragmentCount \leq MAX_FRAG_COUNT$.
ECB_Fragment[1]	This field specifies a fragment structure.

A-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

Appendix **B** *Portability Issues*

Portability Issues Overview

For your code 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.

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 microprocessors store numbers in little endian order.

- Pay attention to alignment constraints when allocating memory and using pointers. The addresses that certain operands may be assigned to are restricted on some architectures.
- Be aware that pointers to objects may 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*, etc.). 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 (suggested).
- Use the *sizeof* operator to determine the size of an object, rather than making an assumption or hard-coding a value.

B-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

- 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.
- Watch out for signed characters. Code that assumes signed/unsigned is not portable.
- Avoid assuming ASCII. Characters may hold more than 8 bits.

Portability Issues **B-3**

- 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 may have holes. Simply computing the address of an unallocated element in an array may 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*'d files
- 32 nesting levels of parenthesized expressions within a full expression.
- 1024 macro identifiers simultaneously.
- 509 characters in a logical source line.
- Six significant initial characters in an external identifier.
- 127 members in a single structure or union.

B-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

- 31 parameters in one function call.

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 a programmer 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 that 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*.

B-6 ODI Specification: Protocol Stacks and MLIDs (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 an MLID.

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: Protocol Stacks and MLIDs (C Language)

```

CEtherTSMSEndComplete
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>Example of the description string: Novell Ethernet NE2000</p>
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.
MAP	Tells the linker to create a map file.

Platform Specific Information **C-3**

Table C-1

Linker Definition File Example Definitions *continued*

Name	Description
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 some debuggers.
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: Protocol Stacks and MLIDs (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

MLIDs 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 MLID 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 load parameters and examples of their use are described below.

Table C-2
Load Keywords and Parameters Descriptions

Name	Descriptions
PORT, PORT1	I/O mapped address base that the user wants the board to use. A port length can also be included as shown in the following examples. LOAD <driver> PORT=300 LOAD <driver> PORT=300:A LOAD <driver> PORT=300:A PORT1=700:8
MEM, MEM1	Beginning address of the shared RAM that the board can use. The size of the shared memory buffer can also be specified. LOAD <driver> MEM=C0000 LOAD <driver> MEM=C0000:1000 LOAD <driver> MEM=C0000:1000 MEM1=CC000
INT, INT1	Interrupt number that the board is expected to use to awaken the interrupt service routine. LOAD <driver> INT=3 LOAD <driver> INT=3 INT1=5
DMA, DMA1	If the board supports DMA, this is the direct memory address channel that the adapter should use for data transfer to memory. LOAD <driver> DMA=0 LOAD <driver> DMA=0 DMA1=3
SLOT	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. LOAD <driver> SLOT=4
RETRIES	Number of send retries that the MLID should use in its attempts to send packets. RETRIES = n

Table C-2
Load Keywords and Parameters Descriptions *continued*

Name	Descriptions
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>
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>
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>

C-8 ODI Specification: Protocol Stacks and MLIDs (C Language)

Appendix

D

ODI HEADER FILE

Appendix Overview

This appendix contains the contents of the ODI header file which contains structures and defines needed by the MLI or MPI interface.

ODI.H

```

/*-----*
*      Copyright Unpublished Work of Novell, Inc. All Rights Reserved
*
*      THIS WORK IS AN UNPUBLISHED WORK AND CONTAINS CONFIDENTIAL,
*      PROPRIETARY AND TRADE SECRET INFORMATION OF NOVELL, INC.
*      ACCESS TO THIS WORK IS RESTRICTED TO (I) NOVELL EMPLOYEES
*      WHO HAVE A NEED TO KNOW TO PERFORM TASKS WITHIN THE SCOPE
*      OF THEIR ASSIGNMENTS AND (ii) ENTITIES OTHER THAN NOVELL
*      WHO HAVE ENTERED INTO APPROPRIATE LICENSE AGREEMENTS.
*      NO PART OF THIS WORK MAY BE USED, PRACTICED, PERFORMED, COPIED,
*      DISTRIBUTED, REVISED, MODIFIED, TRANSLATED, ABRIDGED,
*      CONDENSED, EXPANDED, COLLECTED, COMPILED, LINKED, RECAST,
*      TRANSFORMED OR ADAPTED WITHOUT THE PRIOR WRITTEN CONSENT OF
*      NOVELL. ANY USE OR EXPLOITATION OF THIS WORK WITHOUT AUTHORIZATION
*      COULD SUBJECT THE PERPETRATOR TO CRIMINAL AND CIVIL LIABILITY.
*-----*/
/*****
*
* Program Name:C ODI Header File
*
* Filename:ODI.H
*
* ODI Spec Ver:1.11
*
* Description:This file is the main source for the C ODI SPECIFICATION.
* Any structures needed by the MLI or MPI interface are defined here.
*
*****/

```

```
#ifndef _ODI_Include_
#define _ODI_Include_

/* C ODI Specification Version Numbers */

#define ODI_SPEC_MAJOR_VER1
#define ODI_SPEC_MINOR_VER11

/*State Definitions TRUE / FALSE*/

#ifndef OS_NT
#ifndef OS_WIN95
#ifndef _cdecl
#define _cdecl
#endif
#endif
#endif
#ifdef OS_WIN95
#define CALLCNV _cdecl
#else
#define CALLCNV
#endif

#ifdef TRUE
#undef TRUE
#endif

#ifdef FALSE
#undef FALSE
#endif

#ifdef UNUSED
#undef UNUSED
#endif

#ifdef BOOLEAN
#undef BOOLEAN
#endif

#define FALSE 0x0
#define TRUE 0x1
#define UNUSED -1
#define BOOLEAN unsigned char
```

D-2 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

/*****
C ODI Type Definitions

The following are typedef definitions for parameters used by the
ANSI C ODI Interface. The conditional declarations define both a
preprocessor symbol as itself and a typedef of the same name.
The purpose behind this is to protect the typedef against multiple
declarations. The definition of a preprocessor definition as
itself is allowed by the ANSI language specification.
*****/

#ifndef MEON
#define MEON MEON
typedef unsigned char MEON;
#endif

/* Definition for MEON Strings, NB. MEON_STRING is really used as mnemonic.*/

#ifndef MEON_STRING
#define MEON_STRING MEON_STRING typedef unsigned char MEON_STRING;
/* by convention MEON_STRINGS
NULL terminated */
#endif

#ifndef UINT8
#define UINT8      UINT8
typedef unsigned char      UINT8;
#endif

#ifndef UINT16
#define UINT16     UINT16
typedef unsigned short      UINT16;
#endif

#ifndef UINT32
#define UINT32     UINT32
typedef unsigned int      UINT32;
#endif

#ifndef UINT64
#define UINT64     UINT64
typedef struct_      UINT64_
{
    UINT32      Low_UINT32;
    UINT32      High_UINT32;
} UINT64;

```

```
#endif

#ifndef VOID
#define VOID VOID
typedef void VOID;
#endif

/* declare the pointer for the ODI definitions */

#ifndef PMEON
#define PMEON PMEON
typedef MEON *PMEON;
#endif

#ifndef PUINT8
#define PUINT8 PUINT8
typedef UINT8 *PUINT8;
#endif

#ifndef PUINT16
#define PUINT16 PUINT16
typedef UINT16* PUINT16;
#endif

#ifndef PUINT32
#define PUINT32 PUINT32
typedef UINT32* PUINT32;
#endif

#ifndef PUINT64
#define PUINT64 PUINT64
typedef UINT64* PUINT64;
#endif

#ifndef PVOID
#define PVOID PVOID
typedef VOID* PVOID;
#endif
```

D-4 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

/* Well Defined Sizes */

#define PID_SIZE          0x06/* Number of Octets in Protocol Identifier */
#define ADDR_SIZE         0x06/* Number of Octets in Address */

/* define assumed Maximum Media Header Size that we'll encounter */
/**/
/* assume that it will be Token-Ring (with SRT)*/
/**/
/* AC, FC, Dest[6], Source[6], SRFields[30], 802.2UI[3], SNAP[5] = 52*/
/*      */
#define MAX_MEDIA_HEADER_SIZE52L

#define DefaultNumECBs     0x00
#define DefaultECBSize     1518L/* not including ECB Structure */
#define MinECBSize         (512+74+MAX_MEDIA_HEADER_SIZE)L
                                   /* Max. ECB size < 64K*/

#define MAXLOOKAHEADSIZE  128L/* Max. LookAhead Data Size */

#define MAXSTACKNAMELENGTH      65L
#define MAXNAMELENGTH          64L

#define MAXMULTICASTS          16L      /* MAXIMUM number of Multicast addresses*/
#define MAX_FRAG_COUNT          16L      /* MAXIMUM number of Fragment Count */

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

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

```

```

/* Chained Protocol Stack position values */

typedef enum_CHNPOS_
{
    CHNPOS_FIRST_MUST = 0,      /* Load at very first position*/
    CHNPOS_FIRST_NEXT = 1,     /* Load at next front position*/
    CHNPOS_LOAD_ORDER = 2,     /* Load dependent on load order */
    CHNPOS_LAST_NEXT = 3,      /* Load next end position*/
    CHNPOS_LAST_MUST = 4       /* Load at very end of chain*/
} CHNPOS;

#define CHNPOS_MAX_POSIT 0x0004 /* Maximum possible Chain position */
#define CHAINTYPE_TX 0x0000 /* Transmit type Chain Protocol Stack */
#define CHAINTYPE_RX 0x0001 /* Receive type Chain Protocol Stack */

/*SFT III Status values*/

typedef enum_SFTIII_STAT_{
    SFTIII_STAT_SUCCESSFUL = 0,
    SFTIII_STAT_MIRROR_NOT_ACTIVE = 1,
    SFTIII_STAT_NO_PARTNER = 2,
    SFTIII_STAT_OUT_OF_RESOURCES = 3,
    SFTIII_STAT_NOT_SUPPORTED = -1
}SFTIII_STAT;

/* Look Ahead Definitions */
/* MLIDCFG_LookAheadSize Values*/

#define DEFAULT_LOOK_AHEAD_SIZE 0x12/* Default of 18 look ahead bytes*/
#define MAX_LOOK_AHEAD_SIZE 0x80/* Maximum 128 look ahead bytes*/

/* Rx Packet Attributes(ie. LkAhd_PktAttr) */

#define PAE_CRC_BIT 0x00000001 /* CRC Error */
#define PAE_CRC_ALIGN_BIT 0x00000002 /* CRC/Frame Alignment Error */
#define PAE_RUNT_PACKET_BIT 0x00000004 /* Runt Packet */
#define PAE_TOO_BIG_BIT 0x00000010 /* Packet Too Large for Media */
#define PAE_NOT_ENABLED_BIT 0x00000020 /* Unsupported Frame */
#define PAE_MALFORMED_BIT 0x00000040 /* Malformed Packet */
#define PA_NO_COMPRESS_BIT 0x00004000 /* Do not compress received packet*/
#define PA_NONCAN_ADDR_BIT 0x00008000 /* Set if Addr.inImmediateAddress field is noncanonical*/

```

D-6 ODI Specification: Protocol Stacks and MLIDs (C Language)


```

#define PAE_ERROR_MASK(PAE_CRC_BIT | PAE_CRC_ALIGN_BIT |
PAE_RUNT_PACKET_BIT | \
PAE_TOO_BIG_BIT |
PAE_NOT_ENABLED_BIT | PAE_MALFORMED_BIT)

/* Rx Packet Destination Address Types(LkAhd_DestType, Modify Stack Filter) */
/* Low order 16 bits of the Rx Packet Destination Address Types are copied in
ECB_DriverWorkspace.DWs_il6val[0] */

#define DT_MULTICAST      0x00000001          /* Multicast Dest.
address (Group Address)
*/
#define DT_BROADCAST      0x00000002          /* Broadcast Dest.
address */
#define DT_REMOTE_UNICAST 0x00000004          /* Remote Unicast Dest.
address */
#define DT_REMOTE_MULTICAST 0x00000008        /* Unsupported
Multicast address */
#define DT_SOURCE_ROUTE   0x00000010          /*Source Routed
packet*/
#define DT_ERRORED        0x00000020          /* Global Error,
exculsive bit. */
#define DT_MAC_FRAME      0x00000040          /*MAC/SMT frames. (ie.
NON-DATA Frame)
*/
#define DT_DIRECT         0x00000080          /* Unicast for this
workstation */

#define DT_8022_TYPE_I     0x00000100          /*Set if packet is 802.2
Type I*/
#define DT_8022_TYPE_II    0x00000200          /*Set if
packet is 802.2 Type II
*/

#define DT_8022_BYTES_BITS 0x00000300
#define DT_RX_PRIORITY     0x00000400          /* Set if packet has priority
value other than base
values*/
#define DT_PROMISCUOUS (DT_ERRORED | DT_DIRECT | DT_MULTICAST |
DT_BROADCAST | \
DT_REMOTE_UNICAST |
DT_REMOTE_MULTICAST | DT_SOURCE_ROUTE | \
DT_MAC_FRAME |
DT_RX_PRIORITY)

/* mask off allowable Destination Address Types */

```

ODI Header File **D-7**

```

#define DT_MASK(DT_MULTICAST | DT_BROADCAST |
DT_REMOTE_UNICAST | \

DT_REMOTE_MULTICAST | DT_SOURCE_ROUTE | DT_ERRORED | \
DT_MAC_FRAME |
DT_DIRECT | DT_RX_PRIORITY)

/* ECB Definitions */

/* Stack ID Definitions */

#define ECB_RAWSEND 0xFFFF /* Raw Send, ie. ECB includes MAC Header */
/*implies MLID should not */
/* build MAC Header for this Tx.*/

/* Receive ECB_DriverWorkspace.DWs_il6val[0] */

#define ECB_MULTICAST      0x0001          /* Multicast Dest. address (Group
                                           Address) */
#define ECB_BROADCAST      0x0002          /* Broadcast Dest. address */
#define ECB_UNICASTREMOTE  0x0004          /* Remote Unicast Dest. address
                                           */
#define ECB_MULTICASTREMOTE 0x0008 /* Unsupported Multicast address */
#define ECB_SOURCE_ROUTE   0x0010          /* Source Routed packet*/
#define ECB_GLOBALERROR    0x0020          /* Set if packet contains errors
                                           (exculsive)*/

/* NB. If set all other bits should be reset.*/
#define ECB_MACFRAME 0x0040 /* Packet is not a data packet. */

/* NB. If set all other bits should be reset. */
#define ECB_UNICASTDIRECT 0x0080 /* Unicast for this workstation */

#define ECB_MASK 0xFFFF /* mask off allowable Destination Address Types */

#define ECB_TYPE_I      0x0100          /* Set if packet is 802.2 Type I
                                           */
#define ECB_TYPE_II     0x0200          /* Set if packet is 802.2 Type II
                                           */
#define ECB_RX_PRIORITY  0x0400          /* Set if packet has priority
                                           value other base
                                           values*/
#define ECB_PROMISCUOUS( ECB_ERRORED | ECB_MULTICAST | ECB_BROADCAST |
ECB_UNICASTREMOTE | \

```

D-8 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

ECB_MULTICASTREMOTE | ECB_SOURCE_ROUTE | ECB_MAC_FRAME | ECB_RX_PRIORITY)

/* PromiscuousChange state and mode values.*/
#define PROM_STATE_OFF 0x00 /* Disable Promiscuous Mode*/
#define PROM_STATE_ON 0x01 /* Enable Promiscuous Mode */
#define PROM_MODE_QUERY 0x00 /* Query as to promiscuous mode
                             */
#define PROM_MODE_MAC 0x01 /* MAC frames*/
#define PROM_MODE_NON_MAC 0x02 /* Non-MAC frames*/
#define PROM_MODE_MACANDNON 0x03 /* Both MAC and Non-MAC frames*/
#define PROM_MODE_SMT 0x04 /* FDDI SMT Type MAC frames. */
#define PROM_MODE_RMC 0x08 /* Remote Multicast frames */

/* Operation Scope Definitions */

typedef enum _OPERATION_SCOPE_
{
    OP_SCOPE_ADAPTER= 0,
    OP_SCOPE_LOGICAL_BOARD = 1
} OPERATION_SCOPE;

/* System Return Code Definitions */

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_IN_CRITICAL_SECTION= -117,
    ODISTAT_TRANSMIT_FAILED= -116,
    ODISTAT_PACKET_UNDELIVERABLE= -115,
    ODISTAT_CANCELED= -4
}ODISTAT;

#define ODISTAT_NO_SUCH_DRIVERODISTAT_MLID_SHUTDOWN

```

```

/* MLID Configuration Table Bit Defintions. */

/* MLID 'Flags' Bit Definitions. */
#define MF_HUB_MANAGEMENT_BIT      0x0100
#define MF_SOFT_FILT_GRP_BIT      0x0200
#define MF_GRP_ADDR_SUP_BIT       0x0400
#define MF_MULTICAST_TYPE_BITS    0x0600
#define MF_RECONFIG_BIT           0x0800
#define MF_PRIORITYSUP_BIT        0x1000

/* MLID 'ModeFlags' Bit Definitions. */
#define MM_REAL_DRV_BIT           0x0001
#define MM_USES_DMA_BIT           0x0002
#define MM_DEPENDABLE_BIT         0x0004
                                   /*Should only be set if MM_POINT_TO_POINT_BIT*/

/*set, for hardware that is normally*/
/*dependable but is not 100% guaranteed*/
#define MM_MULTICAST_BIT          0x0008

/*#redef'ed defineMM_POINT_TO_POINT_BIT0x0010*/
/* Set if point-to-point link,dynamic */

/* call setup and tear down, eg. X.25*/
#define MM_CSL_COMPLIANT_BIT       0x0010 /* Set if MLID is CSL compliant*/
#define MM_PREFILLED_ECB_BIT      0x0020 /* MLID supplies prefilled ECBs*/
#define MM_RAW_SENDS_BIT          0x0040
#define MM_DATA_SZ_UNKNOWN_BIT    0x0080
    #define MM_SMP_BIT             0x0100 /* Set if MLID is SMP enabled.*/
#define MM_FRAG_RECEIVES_BIT      0x0400 /* MLID can handle Fragmented
                                           Receive ECB. */
#define MM_C_HSM_BIT              0x0800 /* Set if HSM written in C. */
#define MM_FRAGS_PHYS_BIT         0x1000 /* Set if HSM wants Frags with
                                           Physical Addresses.*/
#define MM_PROMISCUOUS_BIT        0x2000 /* Set if supports Promiscuous
                                           Mode. */
#define MM_NONCANONICAL_BIT       0x4000 /* Set if Config Node Address
                                           Non-Canonical. */
#define MM_PHYS_NODE_ADDR_BIT     0x8000 /* Set if MLID utilizes Physical
                                           Node Address. */
#define MM_CANONICAL_BITS         0xC000

```

D-10 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

/* MLID 'SharingFlags' Bit Defintions */
#define MS_SHUTDOWN_BIT          0x0001
#define MS_SHARE_PORT0_BIT       0x0002
#define MS_SHARE_PORT1_BIT       0x0004
#define MS_SHARE_MEMORY0_BIT     0x0008
#define MS_SHARE_MEMORY1_BIT     0x0010
#define MS_SHARE_IRQ0_BIT        0x0020
#define MS_SHARE_IRQ1_BIT        0x0040
#define MS_SHARE_DMA0_BIT        0x0080
#define MS_SHARE_DMA1_BIT        0x0100
#define MS_HAS_CMD_INFO_BIT      0x0200
#define MS_NO_DEFAULT_INFO_BIT   0x0400
#define MS_MEM_PAGE_BIT          0x8000

/* MLID 'LineSpeed' Bit Definitions. */
#define MLS_MASK                  0x7FFF
#define MLS_KILO_IND_BIT         0x8000

/* MLID unused resource definitions. */
#define UNUSED_SLOT               0xFFFF
#define UNUSED_IO_PORT           0
#define UNUSED_IO_RANGE          0
#define UNUSED_MEMORY_ADDRESS    0
#define UNUSED_MEMORY_SIZE       0
#define UNUSED_INTERRUPT         0xFF
#define UNUSED_DMA_LINE          0xFF

#define UNASSIGNED_BOARD_NUMBER  0xFFFF

/* STAT_TABLE_ENTRY Definitons. */

#define ODI_STAT_UNUSED          0xFFFFFFFF/* Statistics Table Entry not in use.*/
#define ODI_STAT_UINT32          0x00000000/* Statistics Table Entry UINT32 Counter.
                                           */
#define ODI_STAT_UINT64          0x00000001/* Statistics Table Entry UINT64 Counter.
                                           */
#define ODI_STAT_MEON_STRING0x00000002/* Statistics Table Entry Counter is a
MEON_STRING. */
#define ODI_STAT_UNTYPED         0x00000003/* Statistics Table Entry Counter is a
                                           UINT32 */

/* length preceded array UINT8*/
#define ODI_STAT_RESETABLE0x80000000/* Statistics Table Entry Counter is
resetable */

/* by external entity.This define is only used */
/* by management agents and other modules that */
/* are using STAT_TABLE definintions from this */
/* header file.ODI_STAT_RESETABLE is not a */

```

ODI Header File **D-11**

```

/* part of the ODI Specification and is not allowed */
/* for MLID stats counters.ODI considers */
/* 0x80000000 a reserved bit to avoid conflicts */
/* with other modules in the system. */

#ifdef OS_NT
/* Set PRAGMA to pack these structures */

#pragma pack(1)

#endif

typedef struct _STAT_TABLE_ENTRY_
{
    UINT32 StatUseFlag;
    void* StatCounter;
    MEON_STRING* StatString;
}
StatTableEntry, *PStatTableEntry, STAT_TABLE_ENTRY;

/* StatUseFlagDetermines how StatCounter is defined. */
/* *StatCounterpointer as defined by the StatUseFlag.*/
/* *StatCounterpointer to a UINT32 or UINT64 counter.*/
/* *StatStringpointer to a MEON String, describing the statistics counter.*/

/* Definitions for Information Block for passing API's, eg. Function Lists */

typedef struct _INFO_BLOCK_
{
    UINT32 NumberOfAPIs;
    void(**SupportAPIArray)();
}
INFO_BLOCK, *PINFO_BLOCK;

/* Definitions for Link Support Layer (LSL) */

typedef struct _LOG_BRD_STAT_TABLE_ENTRY_
{
    UINT32 LogBrd_TransmittedPackets;
    UINT32 LogBrd_ReceivedPackets;
    UINT32 LogBrd_UnclaimedPackets;
    UINT32 LogBrd_TxOverloaded;
}
LogBrdStatTableEntry, *PLogBrdStatTableEntry, LOG_BRD_STAT_TABLE_ENTRY;

```

D-12 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

/*****
C LSL Configuration Table Definitions.
*****/

#define CLSL_CFG_TABLE_MAJOR_VER 2
#define CLSL_CFG_TABLE_MINOR_VER 1

/*****
C LSL Configuration Table System Flags definitions.
*****/

#define CLSL_CFG_SERVER_BIT0x40000000
#define CLSL_CFG_CLIENT_BIT0x80000000

/*****
C LSL Configuration Table Structure definition.
*****/

typedef struct _LSL_CONFIG_TABLE_
{
    UINT16          LConfigTableMajorVer;
    UINT16          LConfigTableMinorVer;
    MEON_STRING     *LSLLongName;
    MEON_STRING     *LSLShortName;
    UINT16          LSLMajorVer;
    UINT16          LSLMinorVer;
    UINT32          LMaxNumberOfBoards;
    UINT32          LMaxNumberOfStacks;
    UINT32          LConfigTableReserved0;
    UINT32          LConfigTableReserved1;
    UINT32          LConfigTableReserved2;
    UINT8           LSLCFG_ODISpecMajorVer;
    UINT8           LSLCFG_ODISpecMinorVer;
    UINT16          LConfigTableReserved3;
    UINT32          LSLCFG_SystemFlags;
    UINT32          LSLCFG_SmallECBCount;
    UINT32          LSLCFG_MediumECBCount;
    UINT32          LSLCFG_LargeECBCount;
    UINT32          LSLCFG_XLargeECBCount;
    UINT32          LSLCFG_HugeECBCount;
    UINT32          LSLCFG_SmallECBBelow16Count;
    UINT32          LSLCFG_MediumECBBelow16Count;
    UINT32          LSLCFG_LargeECBBelow16Count;
    UINT32          LSLCFG_XLargeECBBelow16Count;
    UINT32          LSLCFG_HugeECBBelow16Count;
    UINT32          LSLCFG_SmallECBMinCount;
    UINT32          LSLCFG_MediumECBMinCount;
    UINT32          LSLCFG_LargeECBMinCount;

```

```

        UINT32          LSLCFG_XLargeECBMinCount;
        UINT32          LSLCFG_HugeECBMinCount;
        UINT32          LSLCFG_SmallECBMaxCount;
        UINT32          LSLCFG_MediumECBMaxCount;
        UINT32          LSLCFG_LargeECBMaxCount;
        UINT32          LSLCFG_XLargeECBMaxCount;
        UINT32          LSLCFG_HugeECBMaxCount;
        UINT32          LSLCFG_SmallECBSize;
        UINT32          LSLCFG_MediumECBSize;
        UINT32          LSLCFG_LargeECBSize;
        UINT32          LSLCFG_XLargeECBSize;
        UINT32          LSLCFG_HugeECBSize;
    }
    LSL_ConfigTable, *PLSL_ConfigTable, LSL_CONFIG_TABLE;

    /*****
    C LSL Statistics Table Structure definition.
    *****/

    typedef struct _LSL_STATS_TABLE_
    {
        UINT16          LStatTableMajorVer;
        UINT16          LStatTableMinorVer;
        UINT32          LNumGenericCounters;
        STAT_TABLE_ENTRY (*LGenericCountersPtr)[];
        UINT32          LNumLogicalBoards;
        LOG_BRD_STAT_TABLE_ENTRY (*LogicalBoardStatTablePtr)[];
        UINT32          LNumCustomCounters;
        STAT_TABLE_ENTRY (*LCustomCountersPtr)[];
    }
    LSL_StatsTable, *PLSL_StatsTable, LSL_STATS_TABLE;

#define NUM_GENERIC_LSL_COUNTERS    10

#define LSL_TOTAL_TX_PACKET_COUNT    0
#define LSL_GET_ECB_REQUESTS        1
#define LSL_GET_ECB_FAILURES        2
#define LSL_AES_EVENTS_COUNT        3
#define LSL_POSTPONED_EVENTS        4
#define LSL_CANCEL_EVENT_FAILURES    5
#define LSL_VALID_BUFFERS_REUSED    6
#define LSL_RESERVED                7
#define LSL_TOTAL_RX_PACKETS        8
#define LSL_UNCLAIMED_PACKETS        9

    /* Definitions for LookAhead and Event Control Blocks (ECB). */

```

D-14 ODI Specification: Protocol Stacks and MLIDs (C Language)


```

typedef struct_FRAGMENT_STRUCT_
{
void*FragmentAddress;
UINT32FragmentLength;
}
FRAGMENTSTRUCT, FRAGMENT_STRUCT, *PFRAGMENTSTRUCT;

```

```

/*****

```

With compilers that implement strict alignment the location of
were padding is assumed in the structure has been indicated below.

```

*****/

```

```

typedef struct _ECB_
{
    struct _ECB_*ECB_NextLink;
    struct _ECB_*ECB_PreviousLink;
    UINT16 ECB_Status;
    /*UINT8 ECB_Pad1[2];*/ /* Compiler padding */
    void (_cdecl *ECB_ESR)(struct _ECB_*);
    UINT16 ECB_StackID;
    PROT_ID ECB_ProtocolID;
    UINT32 ECB_BoardNumber;
    NODE_ADDRECB_ImmediateAddress;
    /*UINT8 ECB_Pad2[2];*/ /* Compiler padding */

    union {
        UINT8Dws_i8val[4];
        UINT16Dws_i16val[2];
        UINT32Dws_i32val;
        void*Dws_pval;
    } ECB_DriverWorkspace;

    union {
        UINT8Pws_i8val[8];
        UINT16Pws_i16val[4];
        UINT32Pws_i32val[2];
        UINT64Pws_i64val;
        void*Pws_pval[2];
    } ECB_ProtocolWorkspace;

    UINT32 ECB_DataLength;
    UINT32E CB_FragmentCount;
    FRAGMENT_STRUCT ECB_Fragment[1];
}
ECB, *PECB;

```

```

#define MAX_ECB_FRAGS      16
#define RAW_SEND           0xFFFF /* If in ECB.ECB_StackID raw send (Old
                                   Def.) */
#define RAW_SEND_PRIORITY_0x0000 /* Same as old Def. No Priority */
#define RAW_SEND_PRIORITY_10x0001 /* Scale 1-7; 1 being lowest */
#define RAW_SEND_PRIORITY_20x0002
#define RAW_SEND_PRIORITY_30x0003
#define RAW_SEND_PRIORITY_40x0004
#define RAW_SEND_PRIORITY_50x0005
#define RAW_SEND_PRIORITY_60x0006
#define RAW_SEND_PRIORITY_70x0007 /* Scale 1-7 7 being Highest Priority */
#define SEND_PRIORITY_0     0x0007 /* Scale 1-7 0 No Priority */
#define SEND_PRIORITY_1     0x0006 /* Scale 1-7 1 being low priority */
#define SEND_PRIORITY_2     0x0005
#define SEND_PRIORITY_3     0x0004
#define SEND_PRIORITY_4     0x0003
#define SEND_PRIORITY_5     0x0002
#define SEND_PRIORITY_6     0x0001
#define SEND_PRIORITY_7     0x0000 /* Scale 1-7 7 being Highest Priority. */
#define NON_STACKED_BIT     0x8000 /* Used to filter non-stack IDs */
#define NON_STACKID_RAW_SEND_MASK 0x80F8 /* Used to filter raw send packets */

/*****
CTCB is used at the MLI interface
*****/
typedef struct _CTCB_FRAGMENT_BLOCK_STRUCT_
{
    UINT32CTCB_FragmentCount;
    FRAGMENT_STRUCT CTCB_Fragment[1];
} CTCB_FRAGMENT_BLOCK;

typedef struct _CTCB_
{
    void *CTCB_Reserved;
    UINT32 CTCB_BoardNumber;
    UINT32 CTCB_DriverWS[3];
    UINT32 CTCB_DataLen;
    CTCB_FRAGMENT_BLOCK*CTCB_FragBlockPtr;
    UINT32CTCB_MediaHeaderLen;
    UINT8
    CTCB_MediaHeader[MAX_MEDIA_HEADER_SIZE];
} CTCB;

```

D-16 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

/*****

```

With compilers that implement strict alignment the location of
were padding is assumed in the structure has been indicated below.

```

*****/

```

```

typedef struct_AES_ECB_
{
    struct_AES_ECB_*AES_Link;
    UINT32AES_MSecondValue;
    UINT16AES_Status;
    /*UINT8AES_Pad1[2];*/ /* Compiler padding */
    void (_cdecl *AES_ESR)(struct _AES_ECB_ *);
    UINT32AES_Reserved;
    void *AES_ResourceObj;
    void *AES_Context;
}
AESECB, *PAESECB, AES_ECB;

```

```

typedef struct_LOOKAHEAD_
{
    ECB*LkAhd_PreFilledECB;
    UINT8*LkAhd_MediaHeaderPtr;
    UINT32LkAhd_MediaHeaderLen;
    UINT8*LkAhd_DataLookAheadPtr;
    UINT32LkAhd_DataLookAheadLen;
    UINT32LkAhd_BoardNumber;
    UINT32LkAhd_PktAttr;
    UINT32LkAhd_DestType;
    UINT32LkAhd_FrameDataSize;
    UINT16LkAhd_PadAlignBytes1;
    PROT_IDLkAhd_ProtocolID;
    UINT16LkAhd_PadAlignBytes2;
    NODE_ADDR LkAhd_ImmediateAddress;
    UINT32LkAhd_FrameDataStartCopyOffset;
    UINT32LkAhd_FrameDataBytesWanted;
    ECB*LkAhd_ReturnedECB;
    UINT32LkAhd_PriorityLevel;
    void*LkAhd_Reserved;
}
LOOKAHEAD, *PLOOKAHEAD;

```

```

/* Definitions for Protocol Stack Configuration And Statistics Tables */

/*****
C Protocol Stack Configuration Table Definitions.
*****/

#define PSTK_CONFIG_TABLE_MAJOR_VER 2
#define PSTK_CONFIG_TABLE_MINOR_VER 1

/*****
C Protocol Stack Configuration Table System Flags definitions.
*****/

#define PSTK_CFG_AUTO_NETWORK_RESOLUTION_BIT 0x08000000
#define PSTK_CFG_AUTO_BIND_ACTIVE_BIT 0x10000000
#define PSTK_CFG_ROUTER_ACTIVE_BIT 0x20000000
#define PSTK_CFG_SERVER_BIT 0x40000000
#define PSTK_CFG_CLIENT_BIT 0x80000000

/*****
C Protocol Stack Configuration Table Structure definition.
*****/

typedef struct _PS_CONFIG_TABLE_
{
    UINT16PConfigTableMajorVer;
    UINT16PConfigTableMinorVer;
    MEON_STRING*PProtocolLongName;
    MEON_STRING*PProtocolShortName;
    UINT16PProtocolMajorVer;
    UINT16PProtocolMinorVer;
    UINT8 PConfigTable_ODISpecMajorVersion;
    UINT8 PConfigTable_ODISpecMinorVersion;
    UINT8 PConfigTable_ProtocolAPIMajorVersion;
    UINT8 PConfigTable_ProtocolAPIMinorVersion;
    UINT32 PConfigTable_SystemFlags;
    UINT32 PConfigTable_ProtocolFlags;
    UINT32 PConfigTable_ProtocolReserved;
}
PS_ConfigTable, *PPS_ConfigTable, PS_CONFIG_TABLE;

```

D-18 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

typedef struct _PS_STATS_TABLE_
{
    UINT16PStatTableMajorVer; /* Config Table Version 2.00 */
    UINT16PStatTableMinorVer; /* For the CODI 1.10 Spec */
    UINT32PNumGenericCounters;
    STAT_TABLE_ENTRY(*PGenericCountersPtr)[];
    UINT32PNumCustomCounters;
    STAT_TABLE_ENTRY(*PCustomCountersPtr)[];
}
PS_StatsTable, *PPS_StatsTable, PS_STATS_TABLE;

#define NUM_GENERIC_PS_COUNTERS    3

#define PS_TOTAL_TX_PACKETS        0
#define PS_TOTAL_RX_PACKETS        1
#define PS_IGNORED_RX_PACKETS      2

/*****
Network Address Information structure used by protocol stack IOCTL 9.
*****/

typedef struct_NETWORK_ADDRESS_INFO_
{
    UINT32    addressType;
    UINT32    size;
    UINT8     address[32];
} NETWORK_ADDRESS_INFO;

/*****

With compilers that implement strict alignment the location of
were padding is assumed in the structure has been indicated below.

*****/

typedef struct _Lan_Memory_Configuration_
{
    void*      MemoryAddress;
    UINT16     MemorySize;
    /*UINT8     Lan_Mem_Pad1[2];*/ /* Compiler padding */
} Lan_Memory_Configuration;

```

```
/* ***** */
/* Definitions for MLID Configuration, Statistics Tables and Misc. structures
   */
/* ***** */
```

/******

With compilers that implement strict alignment the location of
were padding is assumed in the structure has been indicated below.

*****/

```
typedef struct _MLID_CONFIG_TABLE_
{
    MEON            MLIDCFG_Signature[26];
    UINT8           MLIDCFG_MajorVersion; /* Config Table Version
                                           1.21 */
    UINT8           MLIDCFG_MinorVersion; /* for the CODI 1.11 Spec
                                           */

    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;
    /*UINT8         MLIDCFG_Pad1[2];*/ /* Compiler padding */
    UINT32          (_cdecl *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;
    /*UINT8         MLIDCFG_Pad2[2];*/ /* Compiler padding */
    void            *MLIDCFG_DriverLink;
    UINT16          MLIDCFG_SharingFlags;
    UINT16          MLIDCFG_Slot;
    UINT16          MLIDCFG_IOPort0;
    UINT16          MLIDCFG_IORange0;
    UINT16          MLIDCFG_IOPort1;
    UINT16          MLIDCFG_IORange1;
}
```

ODI Header File **D-21**

```

Lan_Memory_Configuration LAN_MEMORY_CONFIGURATION[2];

#define MLIDCFG_MemoryAddress0 LAN_MEMORY_CONFIGURATION[0].MemoryAddress
#define MLIDCFG_MemorySize0 LAN_MEMORY_CONFIGURATION[0].MemorySize
#define MLIDCFG_MemoryAddress1 LAN_MEMORY_CONFIGURATION[1].MemoryAddress
#define MLIDCFG_MemorySize1 LAN_MEMORY_CONFIGURATION[1].MemorySize

    UINT8    MLIDCFG_Interrupt0;
    UINT8    MLIDCFG_Interrupt1;
    UINT8    MLIDCFG_DMALine0;
    UINT8    MLIDCFG_DMALine1;
    void     *MLIDCFG_ResourceTag;
    void     *MLIDCFG_Config;
    void     *MLIDCFG_CommandString;
    MEON_STRING MLID
        CFG_LogicalName[18];
    /*UINT8 MLIDCFG_Pad3[2];**/* Compiler padding */
    void     *MLIDCFG_LinearMemory0;
    void     *MLIDCFG_LinearMemory1;
    UINT16   MLIDCFG_ChannelNumber;
    /*UINT8 MLIDCFG_Pad4[2];**/* Compiler padding */
    void     *MLIDCFG_DBusTag;
    UINT8    MLIDCFG_DIOConfigMajorVer;
    UINT8    MLIDCFG_DIOConfigMinorVer;
    /*UINT8 MLIDCFG_Pad5[2];**/* Compiler padding */
}
MLID_ConfigTable, *PMLID_ConfigTable, MLID_CONFIG_TABLE;

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;
    Lan_Memory_Configuration
    LAN_MEMORY_CONFIGURATION[2];

#define IO_MemoryAddress 0 LAN_MEMORY_CONFIGURATION[0].MemoryAddress
#define IO_MemorySize 0 LAN_MEMORY_CONFIGURATION[0].MemorySize
#define IO_MemoryAddress 1 LAN_MEMORY_CONFIGURATION[1].MemoryAddress
#define IO_MemorySize 1 LAN_MEMORY_CONFIGURATION[1].MemorySize

    UINT8    IO_Interrupt0;
    UINT8    IO_Interrupt1;
    UINT8    IO_DMALine0;

```

D-22 ODI Specification: Protocol Stacks and MLIDs (C Language)


```

    UINT8IO_DMALine1;
    struct ResourceTagStructure*IO_ResourceTag;
    void      *IO_Config;
    void      *IO_CommandString;
    MEON_STRINGIO_LogicalName[18];
    /*UINT8IO_Pad1[2];**/* Compiler padding */
    void      *IO_LinearMemory0;
    void      *IO_LinearMemory1;
    UINT16    IO_ChannelNumber;
    /*UINT8IO_Pad2[2];**/* Compiler padding */
    void      *IO_DBusTag;
    UINT8     IO_DIOConfigMajorVer;
    UINT8     IO_DIOConfigMinorVer;
    /*UINT8IO_Pad3[2];**/* Compiler padding */
}
IO_CONFIG;

typedef struct _MLID_STATS_TABLE_
{
    UINT16          MStatTableMajorVer;
    UINT16          MStatTableMinorVer;
    UINT32          MNumGenericCounters;
    STAT_TABLE_ENTRY (*MGenericCountsPtr)[];
    UINT32          MNumMediaCounters;
    STAT_TABLE_ENTRY (*MMediaCountsPtr)[];
    UINT32          MNumCustomCounters;
    STAT_TABLE_ENTRY (*MCustomCountersPtr)[];
}
MLID_StatsTable, *PMLID_StatsTable, MLID_STATS_TABLE;

#define NUM_GENERIC_MLID_COUNTERS          20
#define MLID_TOTAL_TX_PACKET_COUNT        0
#define MLID_TOTAL_RX_PACKET_COUNT        1
#define MLID_NO_ECB_AVAILABLE_COUNT        2
#define MLID_PACKET_TX_TOO_BIG_COUNT      3
#define MLID_PACKET_TX_TOO_SMALL_COUNT    4
#define MLID_PACKET_RX_OVERFLOW_COUNT      5
#define MLID_PACKET_RX_TOO_BIG_COUNT      6
#define MLID_PACKET_RX_TOO_SMALL_COUNT    7
#define MLID_PACKET_TX_MISC_ERROR_COUNT    8
#define MLID_PACKET_RX_MISC_ERROR_COUNT    9
#define MLID_RETRY_TX_COUNT                10
#define MLID_CHECKSUM_ERROR_COUNT          11
#define MLID_HARDWARE_RX_MISMATCH_COUNT    12
#define MLID_TOTAL_TX_OK_BYTE_COUNT        13
#define MLID_TOTAL_RX_OK_BYTE_COUNT        14

```

```

#define MLID_TOTAL_GROUP_ADDR_TX_COUNT 15
#define MLID_TOTAL_GROUP_ADDR_RX_COUNT 16
#define MLID_ADAPTER_RESET_COUNT 17
#define MLID_ADAPTER_OPR_TIME_STAMP 18
#define MLID_Q_DEPTH 19
#define NUM_TOKEN_SPECIFIC_COUNTERS 13
#define TRN_AC_ERROR_COUNT 0
#define TRN_ABORT_DELIMITER_COUNTER 1
#define TRN_BURST_ERROR_COUNTER 2
#define TRN_FRAME_COPIED_ERROR_COUNTER 3
#define TRN_FREQUENCY_ERROR_COUNTER 4
#define TRN_INTERNAL_ERROR_COUNTER 5
#define TRN_LAST_RING_STATUS 6
#define TRN_LINE_ERROR_COUNTER 7
#define TRN_LOST_FRAME_COUNTER 8
#define TRN_TOKEN_ERROR_COUNTER 9
#define TRN_UPSTREAM_NODE_ADDRESS 10
#define TRN_LAST_RING_ID 11
#define TRN_LAST_BEACON_TYPE 12

#define NUM_ETHERNET_SPECIFIC_COUNTERS 8

#define ETH_TX_OK_SINGLE_COLLISIONS_COUNT 0
#define ETH_TX_OK_MULTIPLE_COLLISIONS_COUNT 1
#define ETH_TX_OK_BUT_DEFERRED 2
#define ETH_TX_ABORT_LATE_COLLISION 3
#define ETH_TX_ABORT_EXCESS_COLLISION 4
#define ETH_TX_ABORT_CARRIER_SENSE 5
#define ETH_TX_ABORT_EXCESSIVE_DEFERRAL 6
#define ETH_RX_ABORT_FRAME_ALIGNMENT 7

#define NUM_FDDI_SPECIFIC_COUNTERS 10

#define FDDI_CONFIGURATION_STATE 0
#define FDDI_UPSTREAM_NODE 1
#define FDDI_DOWNSTREAM_NODE 2
#define FDDI_FRAME_ERROR_COUNT 3
#define FDDI_FRAMES_LOST_COUNT 4
#define FDDI_RING_MANAGEMENT_STATE 5
#define FDDI_LCT_FAILURE_COUNT 6
#define FDDI_LEM_REJECT_COUNT 7
#define FDDI_LEM_COUNT 8
#define FDDI_L_CONNECTION_STATE 9

```

D-24 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

typedef struct _MLID_REG_
{
    void (*MLIDSendHandler)(ECB*, void*);
    INFO_BLOCK*MLID ControlHandler;
    void *MLID SendContext;
    void *MLID ModuleHandle;
}
MLID_Reg, *PMLID_Reg, MLID_REG;

/* Definitions for Bound Protocol Stacks */

typedef struct _PS_BOUND_NODE_
{
    MEON_STRING *ProtocolName;
    ODISTAT(CALLCNV *ProtocolReceiveHandler)(LOOKAHEAD*);
    INFO_BLOCK *ProtocolControlHandler;
    void *ProtocolResourceObj;
}
PS_BoundNode, *PPS_BoundNode, PS_BOUND_NODE;

/* Definitions for PreScan Rx and Default Chained Protocol Stacks */

typedef struct _PS_CHAINED_RX_NODE_
{
    struct _PS_CHAINED_RX_NODE_*StackChainLink;
    UINT32 StackChainBoardNumber;
    CHNPOS StackChainPositionRequested;
    ODISTAT (CALLCNV *StackRxChainHandler)(LOOKAHEAD*, struct
        _PS_CHAINED_RX_NODE_ *);
    INFO_BLOCK *StackChainControl;
    UINT32 StackChainFilter;
    void *StackChainContext;
    void *StackChainResourceObj;
}
PS_ChainedRxNode, *PPS_ChainedRxNode, PS_CHAINED_RX_NODE;

/* Definitions for PreScan Tx Chained Protocol Stacks */

typedef struct _PS_CHAINED_TX_NODE_
{
    struct _PS_CHAINED_TX_NODE_*StackChainLink;
    UINT32 StackChainBoardNumber;
    CHNPOS StackChainPositionRequested;
    ODISTAT (CALLCNV *StackTxChainHandler)(ECB*, struct
        _PS_CHAINED_TX_NODE_ *);
    INFO_BLOCK*StackChainControl;
    UINT32 StackChainFilter;
}

```

ODI Header File **D-25**

```

        void      *StackChainContext;
        void      *StackChainResourceObj;
    }
    PS_ChainedTxNode, *PPS_ChainedTxNode, PS_CHAINED_TX_NODE;

/*Definitions for SFT III Exchange Protocol Control Service */

typedef struct_SFTIII_EXCHANGE_NODE_
{
    UINT32      SubFunction;
    void*       Parameter1;
    void*       Parameter2;
} SFTIIIXchangeNode, SFTIII_EXCHANGE_NODE;

#ifdef   OS_NT

/* Reset PRAGMA to normal after packing above structures */

#pragmapack( )

#endif

/*===== [ Function Prototypes ] =====*/

ODISTAT  CLSL_AddProtocolID(PROT_ID*ProtocolID,
                           MEON_STRING      *ProtocolName,
                           MEON_STRING      *FrameName);

ODISTAT  CLSL_BindProtocolToBoard(UINT32ProtocolNumber,
                                  UINT32      BoardNumber,
                                  MEON_STRING      *UserParmString);

ODISTAT  CLSL_BindStack(UINT32ProtocolNumber,
                        UINT32BoardNumber);

ODISTAT  CLSL_CancelAESEvent(AES_ECB*TimerAESECB);

ODISTAT  CLSL_CancelEvent(ECB*ECBBuffer);

ODISTAT  CLSL_ControlStackFilter(UINT32 BoardNumber,
                                  UINT32  Function,
                                  UINT32  Mask,
                                  void     *Parameter1,
                                  void     *Parameter2);

ODISTAT  CLSL_Dummy(void);

```

D-26 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

ODISTAT  CLSL_DeRegisterDefaultChain(PS_CHAINED_RX_NODE *StackChainNode);

ODISTAT  CLSL_DeRegisterMLID(UINT32BoardNumber);

ODISTAT  CLSL_DeRegisterPreScanChain(PS_CHAINED_RX_NODE *PStkChainRxNode,
PS_CHAINED_TX_NODE *PStkChainTxNode);

ODISTAT  CLSL_DeRegisterStack(UINT32ProtocolNumber);

voidCLSL_FastHoldEvent(ECB*ECBBuffer);

void      CLSL_FastSendComplete(ECB*SendECB);
ODISTAT  CLSL_GetBoundBoardInfo (UINT32BoardNumber,
UINT32    *StackBuffer);

UINT32CLSL_GetIntervalMarker(void);

LSL_CONFIG_TABLE*CLSL_GetLSLConfiguration(void);

LSL_STATS_TABLE*CLSL_GetLSLStatistics(void);

UINT32CLSL_GetMaxECBBufferSize(void);

INFO_BLOCK*CLSL_GetMLIDControlEntry(UINT32 BoardNumber,
ODISTAT *ErrorStatus);

PROT_ID* CLSL_GetPIDFromStackIDBoard(UINT32ProtocolNumber,
UINT32 BoardNumber,
ODISTAT *ErrorStatus);

INFO_BLOCK      *CLSL_GetProtocolControlEntry(UINT32ProtocolNumber,
ODISTAT *ErrorStatus);

ECB      *CLSL_GetSizedECB(UINT32ECBDataSize,
void      *pResourceObj,
BOOLEAN Below16Meg);

ECB      *      CLSL_GetMultipleECBs(UINT32ECBDataSize,
void      *pResourceObj,
UINT32     *nECBs);

ODISTAT  CLSL_GetStackECB(LOOKAHEAD*LookAheadBuf);

ODISTAT  CLSL_GetStackIDFromName(MEON_STRING*Name,
UINT32      *ProtocolNumber);

```

ODI Header File **D-27**

```

ODISTAT  CLSL_GetStartofChain(UINT32 BoardNumber,
                               PS_CHAINED_RX_NODE    **DefaultChainStartNode,
                               PS_CHAINED_RX_NODE    **PreScanRxChainStartNode,
                               PS_CHAINED_TX_NODE    **PreScanTxChainStartNode);

void CLSL_HoldEvent( ECB*HoldECB );

ODISTAT  CLSL_ModifyStackFilter(void*StackIdentifier,
                               UINT32   BoardNumber,
                               UINT32   NewMask,
                               UINT32   *pCurrentMask);

ODISTAT  CLSL_RegisterDefaultChain(PS_CHAINED_RX_NODE*StackChainNode);

ODISTAT  CLSL_RegisterMLID (MLID_REG*MLIDHandlers,
                            MLID_CONFIG_TABLE *MLIDConfigTable,
                            UINT32   *BoardNumber);

ODISTAT  CLSL_RegisterPreScanChain( PS_CHAINED_RX_NODE*PStkChnPreRxNode,
                                    PS_CHAINED_TX_NODE*PStkChnPreTxNode);

ODISTAT  CLSL_RegisterStack(PS_BOUND_NODE*ProtocolNode,
                            UINT32   *ProtocolNumber);

ODISTAT  CLSL_ReSubmitDefault(PS_CHAINED_RX_NODE*StackChainnode,
                              LOOKAHEAD*LookAheadBuf);

ODISTAT  CLSL_ReSubmitPreScanRx(PS_CHAINED_RX_NODE*StackChainnode,
                              LOOKAHEAD*LookAheadBuf);

ODISTAT  CLSL_ReSubmitPreScanTx(PS_CHAINED_TX_NODE*StackChainnode,
                              ECB          *TransmitECB);

ODISTAT_cdecl
        CLSL_ReturnECB( ECB*ReturnedECB );

ODISTAT  CLSL_ScheduleAESEvent( AES_ECB*TimerAESECB );

void      CLSL_SendComplete( ECB*SendECB );

ODISTAT  CLSL_SendPacket( ECB*SendECB );

SFTIII_STAT CLSL_SendProtocolInfoToPartner( UINT32 ProtocolNumber,
                                             UINT8   *ProtocolInfo,
                                             UINT32   Length,
                                             void      (*InfoSendCallBack)
(
    UINT32   Reserved,

```

D-28 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

        UINT8 *ProtocolInfo));

SFTIIII_STATCLSL_SendProtocolInfoToOtherEngine(UINT32ProtocolNumber,
        UINT8      *ProtocolInfo,
        UINT32     Length,
        void      (*InfoSendCallBack)
        void      CLSL_ServiceEvents(void);

        ODISTAT CLSL_UnbindStack(UINT32ProtocolNumber,
        UINT32     BoardNumber);

void CLSL_Reserved( void );

/*===== [ Macro Defintions ] =====*/

/*Macro Definitions to ease access to Control Procedures for
Protocol Stacks and MLIDs
*/

/*Protocol Stack Control Functions*/

#define PSTK_NUM_API 10L

#define PSTK_GET_CONFIGURATION      0x0000
#define PSTK_GET_STATISTICS        0x0001
#define PSTK_BIND                  0x0002
#define PSTK_UNBIND                0x0003
#define PSTK_MLID_DEREGISTER       0x0004
#define PSTK_PROMISCUOUS_STATE     0x0005
#define PSTK_RESERVED              0x0006
#define PSTK_GET_PROTOCOL_STRG     0x0007
#define PSTK_PROT_MANAGE            0x0008
#define PSTK_GET_BOUND_NETWORK_INFO 0x0009

/*w is ptr. to INFO_BLOCK defining API Array for Protocol Stack Control*/

#define PStkCntl_GetConfig(w, x)\
((PS_CONFIG_TABLE* (CALLCNV *)(void*))\
w->SupportAPIArray[PSTK_GET_CONFIGURATION])(x)

#define PStkCntl_GetStats(w, x)\
((PS_STATS_TABLE* (CALLCNV *)(void*)) \
w->SupportAPIArray[PSTK_GET_STATISTICS])(x)

#define PStkCntl_Bind(w, x, y, z)\
((ODISTAT (CALLCNV *)(UINT32, MEON_STRING*, void*))\
w->SupportAPIArray[PSTK_BIND])(x, y, z)

```

ODI Header File **D-29**

```

#define PStkCntl_MLIDDeReg(w, x, y)\
((void (CALLCNV *) (UINT32, void*))\
w->SupportAPIArray[PSTK_MLID_DEREGISTER])(x, y)

#define PStkCntl_Unbind(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, MEON_STRING*, void*))\
w->SupportAPIArray[PSTK_UNBIND])(x, y, z)

#define PStkCntl_PromiscState(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, UINT32, void*))\
w->SupportAPIArray[PSTK_PROMISCUOUS_STATE])(x, y, z)

#define PStkCntl_GetProtocolString(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, MEON_STRING*, void*))\
w->SupportAPIArray[PSTK_GET_PROTOCOL_STRG])(x, y, z)

#define PStkCntl_ProtManage(w, x, y)\
((ODISTAT (CALLCNV *) (ECB *, void*))\
w->SupportAPIArray[PSTK_PROT_MANAGE])(x, y)

#define PSTKCntl_GetBoundNetInfo (w, x, y, z) \
((ODISTAT (CALLCNV*) (UINT32, NETWORK_ADDRESS_INFO*, void* )) \
w->SupportAPIArray[PSTK_GET_GOUND_NET_INFO])(x, y, z)

/*MLID Control Functions*/

#define MLID_NUM_API 19L

#define MLID_GET_CONFIGURATION 0x0000 /* 0 - GetMLIDConfiguration */
#define MLID_GET_STATISTICS 0x0001 /* 1 - GetMLIDStatistics */
#define MLID_ADD_MULTICAST 0x0002 /* 2 - AddMulticastAddress */
#define MLID_DELETE_MULTICAST 0x0003 /* 3 - DeleteMulticastAddress */
#define MLID_RESERVED 0x0004 /* 4 - Reserved */
#define MLID_SHUTDOWN 0x0005 /* 5 - MLIDShutdown */
#define MLID_RESET 0x0006 /* 6 - MLIDReset */
#define MLID_RESERVED1 0x0007 /* 7 - Reserved1 */
#define MLID_RESERVED2 0x0008 /* 8 - Reserved2 */
#define MLID_SET_LOOK_AHEAD 0x0009 /* 9 - SetLookAheadSize */
#define MLID_PROMISCUOUS_CHANGE 0x000A /* 10 - PromiscuousChange */
#define MLID_REGISTER_TX_MONITOR 0x000B /* 11 - RegisterMonitor */
#define MLID_RESERVED3 0x000C /* 12 - Reserved3 */
#define MLID_RESERVED4 0x000D /* 13 - Reserved4 */
#define MLID_MANAGEMENT 0x000E /* 14 - MLIDManagement */
#define MLID_GET_MULTICAST_INFO 0x000F /* 15 - GetMulticastInfo */
#define MLID_REMOVE_NETWORK_INTERFACE 0x0010 /* 16 - RemoveNetworkInterface

```

D-30 ODI Specification: Protocol Stacks and MLIDs (C Language)


```

        */
#define MLID_SHUTDOWN_NETWORK_INTERFACE0x0011/* 17 - ShutdownNetworkInterface */
#define MLID_RESET_NETWORK_INTERFACE          0x0012 /* 18 - ResetNetworkInterface */

/* Shutdown type defines */

#define SHUTDOWN_PERMANENT          0x0000
#define SHUTDOWN_PARTIAL           0x0001

/*w is ptr. to INFO_BLOCK defining API Array for MLID Control*/

#define MLIDCntl_GetConfig(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, MLID_CONFIG_TABLE**, ECB*))\
w->SupportAPIArray[MLID_GET_CONFIGURATION])(x, y, z)

#define MLIDCntl_GetStats(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, MLID_STATS_TABLE**, ECB*))\
w->SupportAPIArray[MLID_GET_STATISTICS])(x, y, z)

#define MLIDCntl_AddMulti(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, NODE_ADDR*, ECB*))\
w->SupportAPIArray[MLID_ADD_MULTICAST])(x, y, z)

#define MLIDCntl_DelMulti(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, NODE_ADDR*, ECB*))\
w->SupportAPIArray[MLID_DELETE_MULTICAST])(x, y, z)

#define MLIDCntl_Shutdown(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, UINT32, ECB*))\
w->SupportAPIArray[MLID_SHUTDOWN])(x, y, z)

#define MLIDCntl_Reset(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, ECB*))\
w->SupportAPIArray[MLID_RESET])(x, y)

#define MLIDCntl_SetLookAhead(w, x, y, z)\
((ODISTAT (CALLCNV *) (UINT32, UINT32, ECB*))\
w->SupportAPIArray[MLID_SET_LOOK_AHEAD])(x, y, z)

#define MLIDCntl_PromisChange(w, x, y, z, aa)\
((ODISTAT (CALLCNV *) (UINT32, UINT32, UINT32*, ECB*))\
w->SupportAPIArray[MLID_PROMISCUOUS_CHANGE])(x, y, z, aa)

```

```

#define MLIDCntl_RegMon(w, x, y, z, aa)\
((ODISTAT (CALLCNV *) (UINT32, void*, ECB*, BOOLEAN))\
w->SupportAPIArray[MLID_REGISTER_TX_MONITOR])(x, y, z, aa)

#define MLIDCntl_Management(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, ECB*))\
w->SupportAPIArray[MLID_MANAGEMENT])(x, y)

#define MLIDCntl_GetMulticastInfo(w, x, y) \
((ODISTAT (CALLCNV *) (UINT32, ECB*))\
w->SupportAPIArray[MLID_GET_MULTICAST_INFO])(x, y)

#define MLIDCntl_RemoveNetworkInterface(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, ECB*))\
w->SupportAPIArray[MLID_REMOVE_NETWORK_INTERFACE])(x, y)

#define MLIDCntl_ShutdownNetworkInterface(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, ECB*))\
w->SupportAPIArray[MLID_SHUTDOWN_NETWORK_INTERFACE])(x, y)

#define MLIDCntl_ResetNetworkInterface(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, ECB*))\
w->SupportAPIArray[MLID_RESET_NETWORK_INTERFACE])(x, y)

typedef struct _GROUP_ADDR_LIST_NODE_ {
    NODE_ADDRGRP_ADDR;
    UINT16GRP_ADDR_COUNT;
} GROUP_ADDR_LIST_NODE;

/*LSL Function Indexes */

#define LSL_NUM_API 49L

#define CLSL_GET_SIZED_ECB 0x0000 /* 0 - CLSL_GetSizedECB */
#define CLSL_RETURN_ECB 0x0001 /* 1 - CLSL_ReturnECB */
#define CLSL_CANCEL_EVENT 0x0002 /* 2 - CLSL_CancelEvent */
#define CLSL_SCHEDULE_AES_EVENT 0x0003 /* 3 - CLSL_ScheduleAESEvent */
#define CLSL_CANCEL_AES_EVENT 0x0004 /* 4 - CLSL_CancelAESEvent */
#define CLSL_GET_INTERVAL_MARKER 0x0005 /* 5 - CLSL_GetIntervalMarker */
#define CLSL_REGISTER_STACK 0x0006 /* 6 - CLSL_RegisterStack */
#define CLSL_DEREGISTER_STACK 0x0007 /* 7 - CLSL_DeRegisterStack */
#define CLSL_RESERVED 0x0008 /* 8 - Reserved */
#define CLSL_RESERVED1 0x0009 /* 9 - Reserved1 */
#define CLSL_RESERVED2 0x000A /* 10 - Reserved2 */
#define CLSL_GET_STACK_ECB 0x000B /* 11 - CLSL_GetStackECB */

```

D-32 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

#define CLSL_SEND_PACKET          0x000C    /* 12 - CLSL_SendPacket */
#define CLSL_FAST_SEND_COMPLETE  0x000D    /* 13 - CLSL_FastSendComplete */
#define CLSL_SEND_COMPLETE       0x000E    /* 14 - CLSL_SendComplete */
#define CLSL_REGISTER_MLID       0x000F    /* 15 - CLSL_RegisterMLID */

#define CLSL_GET_STACK_ID_FROM_NAME
        0x0010    /* 16 - CLSL_GetStackIDFromName */

#define CLSL_GET_PID_FROM_STACK_ID_BOARD
        0x0011    /* 17 - CLSL_GetPIDFromStackIDBoard */

#define CLSL_GET_MLID_CONTROL_ENTRY
        0x0012    /* 18 - CLSL_GetMLIDControlEntry */

#define CLSL_GET_PROTOCOL_CONTROL_ENTRY
        0x0013    /* 19 - CLSL_GetProtocolControlEntry */

#define CLSL_GET_LSL_STATISTICS  0x0014    /* 20 - CLSL_GetLSLStatistics */
#define CLSL_BIND_STACK          0x0015    /* 21 - CLSL_BindStack */
#define CLSL_UNBIND_STACK        0x0016    /* 22 - CLSL_UnbindStack */
#define CLSL_ADD_PROTOCOL_ID     0x0017    /* 23 - CLSL_AddProtocolID */
#define CLSL_GET_BOUND_BOARD_INFO
        0x0018    /* 24 - CLSL_GetBoundBoardInfo */
#define CLSL_GET_LSL_CONFIGURATION 0x0019    /* 25 - CLSL_GetLSLConfiguration
        */
#define CLSL_DEREGISTER_MLID     0x001A    /* 26 - CLSL_DeRegisterMLID */

#define CLSL_REGISTER_DEFAULT_CHAIN
        0x001B    /* 27 - CLSL_RegisterDefaultChain */

#define CLSL_REGISTER_PRESCAN_CHAIN
        0x001C    /* 28 - CLSL_RegisterPreScanChain */

#define CLSL_RESERVED3          0x001D    /* 29 - Reserved3 */

#define CLSL_DEREGISTER_DEFAULT_CHAIN
        0x001E    /* 30 - DeRegisterDefaultChain */

#define CLSL_DEREGISTER_PRESCAN_CHAIN
        0x001F    /* 31 - DeRegisterPreScanChain */

#define CLSL_RESERVED4          0x0020    /* 32 - Reserved4 */
#define CLSL_GET_START_OF_CHAIN  0x0021    /* 33 - CLSL_GetStartofChain */
#define CLSL_RESUBMIT_DEFAULT    0x0022    /* 34 - CLSL_ReSubmitDefault */

#define CLSL_RESUBMIT_PRESCAN_RX
        0x0023    /* 35 - CLSL_ReSubmitPreScanRx */

```

ODI Header File **D-33**

```

#define CLSL_RESUBMIT_PRESCAN_TX
        0x0024 /* 36 - CLSL_ResubmitPreScanTx */
#define CLSL_HOLD_EVENT 0x0025 /* 37 - CLSL_HoldEvent */
#define CLSL_FAST_HOLD_EVENT 0x0026 /* 38 - CLSL_FastHoldEvent */

#define CLSL_GET_MAX_ECB_BUFFER_SIZE
        0x0027 /* 39 - CLSL_GetMaxECBBufferSize */

#define CLSL_RESERVED5 0x0028 /* 40 - Reserved5 */
#define CLSL_SERVICE_EVENTS 0x0029 /* 41 - CLSL_ServiceEvents */

#define CLSL_MODIFY_STACK_FILTER
        0x002A /* 42 - CLSL_ModifyStackFilter */

#define CLSL_CONTROL_STACK_FILTER
        0x002B /* 43 - CLSL_ControlStackFilter */

#define CLSL_SEND_PROTOCOL_INFO_TO_OTHER_ENGINE
        0x002C /* 44 - CLSL_SendProtocolInfoToOtherEngine (Server
ONLY) */

#define CLSL_SEND_PROTOCOL_INFO_TO_PARTNER
        0x002D /* 45 - CLSL_SendProtocolInfoToPartner
(Server ONLY) */

#define CLSL_BIND_PROTOCOL_TO_BOARD
        0x002E /* 46 - CLSL_BindProtocolToBoard */

#define CLSL_GET_MULTIPLE_ECBS 0x002F /* 47 - CLSL_GetMultipleECBs */

#define CLSL_GET_PHYSICAL_ADDRESS_OF_ECB 0x0030
/*48 CLSL_GetPhysicalAddressOfECB */

/* w is ptr. to INFO_BLOCK defining API Array for C LSL APIs */

#define CLSLEntry_GetSizedECB(w, x, y, z)\
((ECB* (CALLCNV *) (UINT32, void*, BOOLEAN))\
w->SupportAPIArray[CLSL_GET_SIZED_ECB])(x, y, z)

#define CLSLEntry_ReturnECB(w, x)\
((ODISTAT (_cdecl *) (ECB*))\
w->SupportAPIArray[CLSL_RETURN_ECB])(x)

#define CLSLEntry_CancelEvent(w, x)\
((ODISTAT (CALLCNV *) (ECB*))\
w->SupportAPIArray[CLSL_CANCEL_EVENT])(x)

```

D-34 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

#define CLSLEntry_ScheduleAESEvent(w, x)\
((void (CALLCNV *) (AES_ECB*))\
w->SupportAPIArray[CLSL_SCHEDULE_AES_EVENT])(x)

#define CLSLEntry_CancelAESEvent(w, x)\
((ODISTAT (CALLCNV *) (AES_ECB*))\
w->SupportAPIArray[CLSL_CANCEL_AES_EVENT])(x)

#define CLSLEntry_GetIntervalMarker(w)\
((UINT32 (CALLCNV *) (void))\
w->SupportAPIArray[CLSL_GET_INTERVAL_MARKER])()

#define CLSLEntry_RegisterStack(w, x, y)\
((ODISTAT (CALLCNV *) (PS_BOUND_NODE*, UINT32*))\
w->SupportAPIArray[CLSL_REGISTER_STACK])(x, y)

#define CLSLEntry_DeRegisterStack(w, x)\
((ODISTAT (CALLCNV *) (UINT32))\
w->SupportAPIArray[CLSL_DEREGISTER_STACK])(x)

#define CLSLEntry_GetStackECB(w, x)\
((ODISTAT (CALLCNV *) (LOOKAHEAD*))\
w->SupportAPIArray[CLSL_GET_STACK_ECB])(x)

#define CLSLEntry_SendPacket(w, x)\
((ODISTAT (CALLCNV *) (ECB*))\
w->SupportAPIArray[CLSL_SEND_PACKET])(x)

#define CLSLEntry_FastSendComplete(w, x)\
((void (CALLCNV *) (ECB*))\
w->SupportAPIArray[CLSL_FAST_SEND_COMPLETE])(x)

#define CLSLEntry_SendComplete(w, x)\
((void (CALLCNV *) (ECB*))\
w->SupportAPIArray[CLSL_SEND_COMPLETE])(x)

#define CLSLEntry_RegisterMLID(w, x, y, z)\
((ODISTAT (CALLCNV *) (MLID_REG*, MLID_CONFIG_TABLE*, UINT32*))\
w->SupportAPIArray[CLSL_REGISTER_MLID])(x, y, z)

#define CLSLEntry_GetStackIDFromName (w, x, y)\
((ODISTAT (CALLCNV *) (MEON_STRING*, UINT32*))\
w->SupportAPIArray[CLSL_GET_STACK_ID_FROM_NAME])(x, y)

#define CLSLEntry_GetPIDFromStackIDBoard(w, x, y, z)\
((PROT_ID* (CALLCNV *) (UINT32, UINT32, ODISTAT*))\
w->SupportAPIArray[CLSL_GET_PID_FROM_STACK_ID_BOARD])(x, y, z)

```

```

#define CLSLEntry_GetMLIDControlEntry(w, x, y)\
((INFO_BLOCK* (CALLCNV *) (UINT32, ODISTAT*))\
w->SupportAPIArray[CLSL_GET_MLID_CONTROL_ENTRY])(x, y)

#define CLSLEntry_GetProtocolControlEntry(w, x, y)\
((INFO_BLOCK* (CALLCNV *) (UINT32, ODISTAT*))\
w->SupportAPIArray[CLSL_GET_PROTOCOL_CONTROL_ENTRY])(x, y)

#define CLSLEntry_GetLSLStatistics(w)\
((LSL_STATS_TABLE* (CALLCNV *) (void))\
w->SupportAPIArray[CLSL_GET_LSL_STATISTICS])()

#define CLSLEntry_BindStack(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, UINT32))\
w->SupportAPIArray[CLSL_BIND_STACK])(x, y)

#define CLSLEntry_UnbindStack(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, UINT32))\
w->SupportAPIArray[CLSL_GET_UNBIND_STACK])(x, y)

#define CLSLEntry_AddProtocolID(w, x, y, z)\
((ODISTAT (CALLCNV *) (PROT_ID*, MEON_STRING*, MEON_STRING*))\
w->SupportAPIArray[CLSL_ADD_PROTOCOL_ID])(x, y, z)

#define CLSLEntry_GetBoundBoardInfo(w, x, y)\
((ODISTAT (CALLCNV *) (UINT32, UINT32*))\
w->SupportAPIArray[CLSL_GET_BOUND_BOARD_INFO])(x, y)

#define CLSLEntry_GetLSLConfiguration(w) \
((LSL_CONFIG_TABLE* (CALLCNV *) (void))\
w->SupportAPIArray[CLSL_GET_LSL_CONFIGURATION])()

#define CLSLEntry_DeRegisterMLID(w, x)\
((ODISTAT (CALLCNV *) (UINT32))\
w->SupportAPIArray[CLSL_DEREGISTER_MLID])(x)

#define CLSLEntry_RegisterDefaultChain(w, x)\
((ODISTAT (CALLCNV *) (PS_CHAINED_RX_NODE*))\
w->SupportAPIArray[CLSL_REGISTER_DEFAULT_CHAIN])(x)

#define CLSLEntry_RegisterPreScanChain(w, x, y)\
((ODISTAT (CALLCNV *) (PS_CHAINED_RX_NODE*, PS_CHAINED_TX_NODE*))\
w->SupportAPIArray[CLSL_REGISTER_PRE_SCAN_CHAIN])(x, y)

#define CLSLEntry_DeRegisterDefaultChain(w, x)\

```

D-36 ODI Specification: Protocol Stacks and MLIDs (C Language)

```

        ((ODISTAT (CALLCNV *) (PS_CHAINED_RX_NODE*)) \
w->SupportAPIArray[CLSL_DEREGISTER_DEFAULT_CHAIN])(x)

#define CLSLEntry_DeRegisterPreScanChain(w, x, y) \
((ODISTAT (CALLCNV *) (PS_CHAINED_RX_NODE*, PS_CHAINED_TX_NODE*)) \
w->SupportAPIArray[CLSL_DEREGISTER_PRE_SCAN_CHAIN])(x, y)

#define CLSLEntry_GetStartofChain(w, x, y, z, aa) \
((ODISTAT (CALLCNV *) (UINT32, PS_CHAINED_RX_NODE*, \
PS_CHAINED_RX_NODE*, PS_CHAINED_TX_NODE*)) \
w->SupportAPIArray[CLSL_GET_START_OF_CHAIN])(x, y, z, aa)

#define CLSLEntry_ReSubmitDefault(w, x, y) \
((ODISTAT (CALLCNV *) (PS_CHAINED_RX_NODE*, LOOKAHEAD*)) \
w->SupportAPIArray[CLSL_RESUBMIT_DEFAULT])(x, y)

#define CLSLEntry_ReSubmitPreScanRx(w, x, y) \
((ODISTAT (CALLCNV *) (PS_CHAINED_RX_NODE*, LOOKAHEAD*)) \
w->SupportAPIArray[CLSL_RESUBMIT_PRESCAN_RX])(x, y)

#define CLSLEntry_ResubmitPreScanTx(w, x, y) \
((ODISTAT (CALLCNV *) (PS_CHAINED_TX_NODE*, ECB*)) \
w->SupportAPIArray[CLSL_RESUBMIT_PRESCAN_TX])(x, y)

#define CLSLEntry_HoldEvent(w, x) \
((void (CALLCNV *) (ECB*)) \
w->SupportAPIArray[CLSL_HOLD_EVENT])(x)

#define CLSLEntry_FastHoldEvent(w, x) \
((void (CALLCNV *) (ECB*)) \
w->SupportAPIArray[CLSL_FAST_HOLD_EVENT])(x)

#define CLSLEntry_GetMaxECBBufferSize(w) \
((UINT32 (CALLCNV *) (void)) \
w->SupportAPIArray[CLSL_GET_MAX_ECB_BUFFER_SIZE])()

#define CLSLEntry_ServiceEvents(w) \
((void (CALLCNV *) (void)) \
w->SupportAPIArray[CLSL_SERVICE_EVENTS])()

#define CLSLEntry_ModifyStackFilter(w, x, y, z, aa) \
((ODISTAT (CALLCNV *) (void*, UINT32, UINT32, UINT32*)) \
w->SupportAPIArray[CLSL_MODIFY_STACK_FILTER])(x, y, z, aa)

#define CLSLEntry_ControlStackFilter(w, x, y, z, aa, bb) \
((ODISTAT (CALLCNV *) (UINT32, UINT32, UINT32, void*, void*)) \
w->SupportAPIArray[CLSL_CONTROL_STACK_FILTER])(x, y, z, aa, bb)

```

ODI Header File **D-37**

```

#define CLSLEntry_SendProtocolInfoToOtherEngine(w, x, y, z, aa)\
    ((SFTIII_STAT (CALLCNV *) (UINT32, UINT8*, UINT32, \
    void(*InfoSendCallBack)(UINT8*)))\
    w->SupportAPIArray[CLSL_SEND_PROTOCOL_INFO_TO_OTHER_ENGINE])(x, y, z,
    aa)

#define CLSLEntry_SendProtocolInfoToPartner(w, x, y, z, aa)\
    ((SFTIII_STAT (CALLCNV *) (UINT32, UINT8*, UINT32, \
    void(*InfoSendCallBack)(UINT32, UINT8*)))\
    w->SupportAPIArray[CLSL_SEND_PROTOCOL_INFO_TO_PARTNER])(x, y, z, aa)

#define CLSLEntry_BindProtocolToBoard(w, x, y, z)\
    ((ODI_STAT (CALLCNV *) (UINT32, UINT32, MEON_STRING*))\
    w->SupportAPIArray[CLSL_BIND_PROTOCOL_TO_BOARD])(x, y, z)

#define CLSLEntry_GetMultipleECBs(w, x, y, z)\
    ((ECB* (CALLCNV *) (UINT32, void*, UINT32*))\
    w->SupportAPIArray[CLSL_GET_MULTIPLE_ECBS])(x, y, z)

#define CLSLEntry_GetPhysicalAddressOfECB (w, x)/
    ((void (CALLCNV *) (ECB*))\
    w->SupportAPIArray[CLSL_GET_PHYSICAL_ADDRESS_OF_ECB])(x)

#endif /* _ODI_Include_ */

```

D-38 ODI Specification: Protocol Stacks and MLIDs (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*.

2**ODI Specification: Protocol Stacks and MLIDs (C Language)**

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.

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.

Packet

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

Peripheral Component Interconnect—PCI

A 32-bit or 64-bit bus standard with multiplexed address and data lines.

Personal Computer Memory Card International Association—PCMCIA

A 16-bit bus standard.

PID--Protocol Identification

A 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 1.20 to Doc Version 1.21 and from Doc Version 1.21 to Doc Version 1.22.

Items 1 through 27 are Doc Version 1.21 changes.

Items 28 through 32 are Doc Version 1.22 changes.

Note



All page numbers refer to the current Doc Version: Doc Version **1.22**.

1. The following new function was added to Chapter 7:

GetBoundNetworkInfo

Index 9

Gets the bound network address for a board / protocol stack combination.

Syntax

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

```
ODISTAT GetBoundNetworkInfo (
    UINT32 BoardNumber,
    NETWORK_ADDRESS_INFO *networkAddress
    void *StackIdentifier );
```

Input Parameters

boardNumber

The board number the protocol stack is to return the network address for.

networkAddress

Pointer to a buffer where the bound network address for the protocol is returned.

StackIdentifier

Pointer is either a Stack ID (SID) identifying a bound protocol stack (in other words, the content of the StackIdentifier parameter is less than the maximum number of bound protocol stacks supported LSL_ConfigTable.LMaxNumberOfStacks), or the pointer is a pointer to a stack chain node.

Output Parameters

networkAddress

NULL is placed at the start of the buffer if no address is returned.

Return Values

ODISTAT_SUCCESSFUL	The network address was successfully returned. Note: ODISTAT_SUCCESSFUL is returned even if the addressType and size fields are zero, and the address field is NULL; this implies that there is no network address for the board and protocol combination.
ODISTAT_BAD_PARAMETER	The MLID corresponding to the requested board number or the protocol stack corresponding to the specified StackIdentifier.

Remarks

The protocol stack will fill in the NETOWRK_ADDRESS_INFO structure addressType field with it's assigned transport address type, the size field with the length of the address, and the address field with the bound network address. IPX returns all 12 bytes, network:node:socket. IP returns 4 bytes, network address only (no socket).

The following Transport Address types have been assigned:

IPX	1
IP	2
DDP	3
NETBEUI	4

The `networkAddress` structure is defined in ODI.H as follows:

```
typedef struct_NETWORK_ADDRESS_INFO_
{
    UINT32addressType;
    UINT32size;
    UINT8address[32];
}NETWORK_ADDRESS_INFO;
```

2. On page 7-1, **GetBoundNetworkInfo** was added to the list of currently defined functions.
3. On page 7-1, **SFTIIExchange** and **ProtocolManagement** were deleted from the list of currently defined functions.
4. On page 7-2, the code sample was replaced with the following code sample:

```
PStkCntl_GetConfig(infoBlock, stackIdentifier)
PStkCntl_GetStats(infoBlock, stackIdentifier)
PStkCntl_Bind(infoBlock, boardNumber, userParmString, stackIdentifier)
PStkCntl_MLIDDeReg(infoBlock, boardNumber, stackIdentifier)
PStkCntl_Unbind(infoBlock, boardNumber, userParmString, stackIdentifier )
PStkCntl_PromiscState(infoBlock, boardNumber, promiscuousMask,
stackIdentifier)
PStkCntl_GetProtocolString(infoBlock, boardNumber, printString,
stackIdentifier)
PStkCntl_ProtManage(infoBlock, ManagementECB, stackIdentifier)
PStkCntl_GetBoundNetInfo(infoBlock, boardNumber, networkAddress,
stackIdentifier)
```

5. The function, **SFTIIExchange**, was deleted from Chapter 7.
6. On page 12-20, in **Table 12-4, "MLIDCFG_SharingFlags Bits Descriptions"**, in the description for `MS_SHUTDOWN_BIT`, the word "adapter" was changed to "logical board".
7. On page 18-4, the following **Note** was added:

If an ECB has been provided, the ESR is only called if
ODISTAT_RESPONSE_DELAYED is returned.
8. On page A-8, in **Table A-1**, under Bit Value, the second occurrence of "ECB_TYPE_I" was changed to "ECB_TYPE_II".

9. On page D-11, the #define value for MS_MEM_PAGE_BIT was changed from 0x0800 to 0x8000.

10. On page D-16, the following two #defines were added:

```
#define NON_STACKED_BIT 0x8000 /* Used to filter non-stack IDs */
#define NON_STACKID_RAW_SEND_MASK 0x80F8 /* Used to filter raw send packets
*/
```

11. On page D-19, the following structure was added:

```

/*****
Network Address Information structure used by protocol stack IOCTL 9.
*****/
typedef struct_NETWORK_ADDRESS_INFO_
{
    UINT32 addressType;
    UINT32 size;
    UINT8 address[32];
} NETWORK_ADDRESS_INFO;
```

12. On page D-29, under "/*Protocol Stack Control Functions*/", the following #define was deleted:

```
#define PSTK_SFTIIExchange 0x0006
```

13. On page D-29, under "/*Protocol Stack Control Functions*/", the following two #defines were added:

```
#define PSTK_RESERVED 0x0006
#define PSTK_GET_BOUND_NETWORK_INFO 0x0009
```

14. On page D-30, the following #define was deleted:

```
#define PStkCntl_SFTIIExchange
```

15. On page D-30, the following #define was added:

```
#define PSTKCntl_GetBoundNetInfo (w, x, y, z) \
((ODISTAT (CALLCNV*)(UINT32, NETWORK_ADDRESS_INFO*, void*)) \
w->SupportAPIArray[PSTK_GET_GOUND_NET_INFO])(x, y, z)
```

16. On page 6-7, under the *ECB_DataLength* Field, the word "date" was changed to "data".

17. On page 9-11, the value of the #define for LSL_NUM_API was changed from 48L to 49L.

18. On page 9-12, the following entry was added as the last item to the LSLAPI_Array:

```
(void (*) )CLSL_GetPhysicalAddressOfECB
```

19. On page 10-2, the following entry was added to the list of functions available from the LSL:

```
CLSL_GetPhysicalAddressOfECB
```

20. On page 10-4, the following entry was added to the list of functions indexed in the information block:

```
48 CLSL_GetPhysicalAddressOfECB
```

21. On page 10-5, the following entry was added to the list of macros:

```
CLSLEntry_GetPhysicalAddressOfECB (info block, ecb)
```

22. The following new API was added to Chapter 10:

CLSL_GetPhysicalAddressOfECB

Index 48 (0x030)

Gets the physical address of an LSL ECB.

Syntax

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

```
ECB *CLSL_GetPhysicalAddressOfECB  
    (ECB *ecb);
```

Input Parameters

ecb

Pointer (logical address) to an LSL ECB.

Output Parameters

None.

Return Values

Pointer (physical address) of the ECB structure.

Remarks

This function can only be used for ECBs obtained via **CLSL_GetSizedECB** or **CLSL_GetMultipleECBs**.

23. On page D-28, the entry for

```
ODISTAT   CLSL_ReturnECB(ECB*ReturnedECB);
```

was changed to:

```
ODISTAT_cdecl   CLSL_ReturnECB(ECB*ReturnedECB);
```

24. On page D-32, under "/* LSL Function Indexes */", the value of the #define for LSL_NUM_API was changed from 48L to 49L.

25. On page D-34, right after "#define CLSL_GET_MULTIPLE_ECBS" the following entry was added:

```
#define CLSL_GET_PHYSICAL_ADDRESS_OF_ECB 0x0030 /*
48 CLSL_GetPhysicalAddressOfECB */
```

26. On page D-34, under "#define CLSLEntry_ReturnECB (w, x)",

```
CALLCNV *
```

was changed to:

```
_cdecl *
```

27. On page D-38, the following entry was added as the last #define:

```
#define CLSLEntry_GetPhysicalAddressOfECB (w, x)/
((void (CALLCNV *) (ECB*)) /
w->SupportAPIArray[CLSL_GET_PHYSICAL_ADDRESS_OF_ECB]) (x)
```

28. On page 5-5, right after the **Multiple Chained Protocol Stacks** section, the following section was added:

MAC Packet Reception

To receive MAC frames, bound protocol stacks must register using either the MACTOK or MACFDI protocol ID. All three reception methods (bound, prescan, and default) must set their filter mask to include DT_MAC_FRAME.

29. On page 9-9, under STAT_TABLE_ENTRY, the following entries:

```
{ ODI_STAT_UINT32, &LPostponedEvents, NULL },
{ ODI_STAT_UINT32, &LValidBufferReused, NULL },
{ ODI_STAT_UINT32, &LReserved, NULL },
```

were changed to:

```
{ ODI_STAT_UINT32, NULL, NULL },
{ ODI_STAT_UINT32, NULL, NULL },
{ ODI_STAT_UINT32, NULL, NULL },
```

30. On page 9-10, in **Table 9-4, "Generic STAT_TABLE_ENTRY Counters Array Fields**, the following entries:

```
LPostponedEvents
LValidBuffersReused
```

were changed to:

```
LReserved1
LReserved2
```

31. On page 10-36:

See Chapter 15, "MLID Control Routines"

was corrected to read:

See Chapter 18, "MLID Control Routines"

32. On page 12-25, under **Field Descriptions:** *StatUseFlag*, the following sentence was added to the definition for ODI_STAT_MEON_STRING:

The maximum string length is 256, including the NULL termination.

and the following sentence was added to the definition for ODI_STAT_UNTYPED:

This value is generally used for debugging and is displayed in hexadecimal bytes.

33. On page 5-12, in the first paragraph, the following text:

There is no PID associated with MAC layer frames in this specification. In previous ODI assembly language specifications, the values **MACTOK** (for Token-Ring management frames) and **MACFDI** (for FDDI management frames) were used. These values are no longer valid.

has been changed to read as follows:

The PID associated with MAC layer frames is **MACTOK** for Token-Ring management frames and **MACFDI** for FDDI management frames. Refer to the "Protocol Stack Packet Reception Methods" section of this chapter for more information about MAC packet reception.

Index

A

- adapter
 - base memory address 12-11
 - reinitializing 18-22
 - shutting down 18-24
- adapter data space 12-24
 - allocating 11-8
 - defined 11-8
- adding
 - Protocol ID (PID) 4-4
- AddMulticastAddress function 18-7
- ADDR_SIZE parameter xxiv
- AES event
 - canceling 10-14, 10-39
 - scheduling 10-80
- AES_ECB structure xxxii, 10-81
 - field descriptions 10-81
- alignment issues B-5
- ANSI C xxi, B-1
- ASCII B-3
- assumptions
 - coding B-5

B

- base memory address
 - adapter 12-11
- big endian B-1
- Bind function 4-9, 7-3
- binding
 - NET.CFG file entry 4-3
 - protocol stack

- logical board 4-8
 - protocol stack to adapter 10-10, 10-12
 - protocol stack to frame type 10-10, 10-12
 - protocol stack to MLID 7-3
- board
 - service routine
 - overview 11-4
- BoardNumber
 - filling in 2-7
- BOOLEAN enumeration xxii
- bound protocol stack 5-4
 - defined 2-9
 - receive handler 5-16
 - registering 10-69
- building CHSM
 - NetWare/Intel C-1
- bus type
 - Extended Industry Standard Architecture 11-13
 - Industry Standard Architecture (ISA) 11-13
 - listed 11-13
 - Micro Channel Architecture 11-13
 - Peripheral Component Interconnect (PCI) 11-13
 - Personal Computer Memory Card Internati 11-13

C

- canceling
 - AES events 10-14, 10-39
 - events 10-15
- canonical and noncanonical addressing
 - document xix, 12-4
- chain
 - protocol stack
 - default 2-10

- defined 4-9
 - prescan 2-10
 - CHNPOS enumeration xxvi
 - CHSM
 - building
 - NetWare/Intel C-1
 - revision level 12-9
 - CLSL_AddProtocolID function 10-8
 - CLSL_BindProtocolToBoard function 10-10
 - CLSL_BindStack function 10-12
 - CLSL_CancelAESEvent function 10-14, 10-39
 - CLSL_CancelEvent function 10-15
 - CLSL_ControlStackFilter function 10-16
 - CLSL_DeRegisterDefaultChain function 10-18
 - CLSL_DeRegisterMLID function 10-20
 - CLSL_DeRegisterPreScanChain function 10-21
 - CLSL_DeRegisterStack function 10-23
 - CLSL_FastHoldEvent function 10-25
 - CLSL_FastSendComplete function 10-27
 - CLSL_GetBoundBoardInfo function 10-29
 - CLSL_GetIntervalMarker function 10-31
 - CLSL_GetLSLConfiguration function 10-32
 - CLSL_GetLSLStatistics function 10-33
 - CLSL_GetMaxECBBufferSize function 10-34
 - CLSL_GetMLIDControlEntry function 10-35
 - CLSL_GetPIDFromStackIDBoard function 10-40
 - CLSL_GetProtocolControlEntry function 10-42
 - CLSL_GetSizedECB function 10-44
 - CLSL_GetStackECB function 10-46
 - CLSL_GetStackIDFromName function 10-49
 - CLSL_GetStartOfChain function 10-51
 - CLSL_HoldEvent function 10-53
 - CLSL_ModifyStackFilter function 10-55
 - CLSL_RegisterDefaultChain function 10-58
 - CLSL_RegisterMLID function 10-61
 - CLSL_RegisterPreScanChain function 10-64
 - CLSL_RegisterStack function 10-69
 - CLSL_ReSubmitDefault function 10-72
 - CLSL_ReSubmitPreScanRx function 10-75
 - CLSL_ReSubmitPreScanTx function 10-77
 - CLSL_ReturnECB function 10-79
 - CLSL_ScheduleAESEvent function 10-80
 - CLSL_SendComplete function 10-83
 - CLSL_SendPacket function 10-85
 - CLSL_SendProtocolInfoToOtherEngine func 10-89
 - CLSL_SendProtocolInfoToPartner function 10-87
 - CLSL_ServiceEvents function 10-91
 - CLSL_UnbindStack function 10-92
 - code
 - portability xxi, B-1
 - completion codes
 - listed 8-2
 - configuration table
 - LSL xxviii, 9-1
 - major version number 9-3
 - minor version number 9-3
 - pointer to 10-32
 - MLID xxxiv, 11-5, 12-2
 - major version number 12-4
 - minor version number 12-4
 - protocol stack xxxiii, 3-1
 - major version number 3-2
 - minor version number 3-2
 - control
 - routines
 - MLID
 - overview 11-4
 - control block
 - event xxxi, 6-4, A-1
 - control procedure
 - required 11-2
 - supported 11-2
 - control routines
 - MLID 18-1
- ## D
- data
 - transfer mode
 - methods 11-13
 - data flow 2-5
 - receive 1-8
 - send 1-6
 - data packing B-5
 - data space
 - adapter 12-24
 - frame 12-1

- data structures
 - MLID 11-5
- default
 - protocol stack 5-4
 - chaining 2-10, 4-9
 - defined 2-10
 - deregistering 10-18
 - receive handler 5-22
 - registering 10-58
- DeleteMulticastAddress function 18-11
- deregistering
 - default protocol stacks 10-18
 - prescan protocol stacks 10-21
- destination
 - determining packet 8-1
- Destination SAP (DSAP) 6-6
- Destination Service Access Point (DSAP) 6-6
- determining
 - packet destination 1-4, 8-1
- DMA channel
 - default 12-13
- document
 - canonical and noncanonical addressing s xix, 12-4
 - frame types xix, 2-9
 - hub management interface xix
 - installation information file xix
 - MLID message definition xix
 - other xix
 - protocol IDs (PIDs) xix, 2-9
 - source routing xix
 - supplement xix
- dynamic method, logical board service 4-3

E

- ECB (Event Control Block) 6-4
- ECB_BoardNumber field 6-6, A-6
- ECB_DataLength field 6-7, A-9
- ECB_DriverWorkspace field A-7
- ECB_ESR field 6-5, A-4
- ECB_Fragment field A-9
- ECB_FragmentCount field 6-7, A-9
- ECB_ImmediateAddress field 6-7, A-6

- ECB_NextLink field A-3
- ECB_PreviousLink field A-3
- ECB_ProtocolID field 6-6, A-6
- ECB_ProtocolWorkspace field A-8
- ECB_StackID field 6-5, A-5
- ECB_Status field A-4
- enumeration
 - BOOLEAN xxii
 - CHNPOS xxvi
 - ODISTAT xxiv
 - SFTIII_STAT xxv
- ETH_RxAbortFrameAlignment field 12-35
- ETH_TxAbortCarrierSense field 12-35
- ETH_TxAbortExcessiveDeferral field 12-35
- ETH_TxAbortExcessCollision field 12-34
- ETH_TxAbortLastCollision field 12-34
- ETH_TxOKButDeferred field 12-34
- ETH_TxOKMultipleCollisionCount field 12-34
- ETH_TxOKSingleCollisionCount field 12-34
- Ethernet
 - group addresses 18-10
 - multicast addresses 18-10
- event
 - canceled 10-15
- Event Control Block (ECB) A-1
 - description A-1
 - ECB_BoardNumber field 6-6
 - ECB_DataLength field 6-7
 - ECB_ESR field 6-5
 - ECB_FragmentCount field 6-7
 - ECB_ImmediateAddress field 6-7
 - ECB_ProtocolID field 6-6
 - ECB_StackID field 6-5
 - maximum buffer size 10-34
 - structure xxxi, A-1
- execution time xx
 - privileged time xx
 - process time xx
- explicit method, logical board service 4-3
- Extended Industry Standard Architecture 11-13

F

FDI_ConfigurationStats field 12-36
 FDI_DownstreamNode field 12-36
 FDI_FrameErrorCount field 12-36
 FDI_FramesLostCount field 12-36
 FDI_LConnectionState field 12-37
 FDI_LCTFailureCount field 12-37
 FDI_LemCount field 12-37
 FDI_LemRejectCount field 12-37
 FDI_RingManagementCount field 12-37
 FDI_UpstreamNode field 12-36
 filter mask
 modifying 10-55
 flags field 12-20
 flow of data
 receive 1-8
 send 1-6
 fragment descriptors 6-8
 fragment structure xxxi, A-1
 FRAGMENT_STRUCT structure xxxi, A-1
 FragmentAddress field A-2
 frame
 data space 12-1
 allocating 11-8
 defined 11-8
 supporting multiple types 2-1, 11-7, 11-8
 type
 relation of to logical board 11-7
 frame types
 document xix, 2-9

G

generic statistics counter
 media specific 12-24
 standard 12-24
 GetMLIDConfiguration function 7-16, 18-13, 18-17,
 18-34, 18-36, 18-40
 GetMLIDStatistics function 18-15
 GetProtocolStackConfiguration function 7-6, 7-8
 GetProtocolStackStatistics function 7-9

GetProtocolStringForBoard function 7-10

group addresses
 adding 18-7
 disabling 18-11
 Ethernet 18-10
 maximum supported 18-9
 group addressing
 hardware 18-9

H

handles
 system
 used by protocol stack in packet receipt 2-9
 hardware
 bus type
 listed 11-13
 data transfer 11-13
 independence 2-1
 HardwareDriverMLID string 12-4
 header file B-2
 hub management 12-18
 bit 12-18
 hub management interface
 document xix

I

Industry Standard Architecture (ISA) bus 11-13
 INFO_BLOCK structure xxviii, 10-7
 field descriptions 10-7
 information block xxviii, 10-7
 initializing
 MLID 11-3
 protocol stack 2-7, 4-1
 installation information file
 document xix
 interface
 Multiple Link Interface (MLI) 10-1
 Multiple Protocol Interface (MPI) 10-1
 interrupt
 vector number 12-12

L

- LAESEventCount field 9-10
- LCancelEventFailures field 9-10
- LConfigTableMajorVer field 9-3
- LConfigTableMinorVer field 9-3
- LConfigTableReserved0 field 9-3
- LConfigTableReserved1 field 9-3
- LConfigTableReserved2 field 9-3
- LCustomCountersPtr field 9-8
- LGenericCountersPtr field 9-8
- LGetECBFailures field 9-10
- LGetECBRequests field 9-10
- Link Support Layer (LSL)
 - completion codes
 - listed 8-2
 - configuration table xxviii, 9-1
 - pointer to 10-32
 - defined 1-4, 8-1
 - interfaces 10-1
 - locating 4-2, 10-6
 - statistics table xxx, 9-7
 - pointer to 10-33
- linker definition file
 - NetWare/Intel C-2
- lins speed
 - protocol stack 4-6
- little endian B-1
- LkAhd_BoardNumber field 5-8
- LkAhd_DataLookAheadLen field 5-8
- LkAhd_DataLookAheadPtr field 5-8
- LkAhd_DestType field 5-10
- LkAhd_FrameDataBytesWanted field 5-13
- LkAhd_FrameDataSize field 5-12
- LkAhd_FrameDataStartCopyOffset field 5-12
- LkAhd_ImmediateAddress field 5-12
- LkAhd_MediaHeaderLen field 5-7
- LkAhd_MediaHeaderPtr field 5-7
- LkAhd_PktAttr field 5-9
- LkAhd_PreFilledECB field 5-7
- LkAhd_PriorityLevel field 5-13
- LkAhd_ProtocolID field 5-12
- LkAhd_Reserved field 5-13
- LkAhd_ReturnedECB field 5-13
- LMaxNumberOfBoards field 9-3
- LMaxNumberOfStacks field 9-3
- LNumCustomCounters field 9-8
- LNumGenericCounters field 9-8
- LNumLogicalBoards field 9-8
- LOG_BRD_STAT_TABLE_ENTRY structure xxviii, 9-7
- LogBrd_ReceivedPackets field 9-8
- LogBrd_TransmittedPackets field 9-8
- LogBrd_UnclaimedPackets field 9-8
- logical
 - board
 - number
 - defined 2-7
 - registering with LSL 10-61
 - relation of to frame type 11-7
 - routing packet to 2-8
 - servicing 4-3
 - supporting 2-1
 - logical board service
 - dynamic method 4-3
 - explicit method 4-3
- LogicalBoardStatTablePtr field 9-8
- lookahead xxxii, 5-6
 - method 4-8, 5-5
 - receive handler 5-6
 - size
 - setting 18-38
- LOOKAHEAD structure xxxii, 5-6
 - field descriptions 5-7
- LReserved field 9-10
- LSL_CONFIG_TABLE structure xxviii, 9-1
- LSL_STATS_TABLE structure xxx, 9-7
- LSLLongName field 9-3
- LSLMajorVer field 9-3
- LSLMinorVer field 9-3
- LSLSHORTNAME field 9-3
- LStatTableMajorVer field 9-8
- LStatTableMinorVer field 9-8
- LTOTALRXPACKETS field 9-10
- LTOTALTXPACKETS field 9-10
- LUnclaimedPackets field 9-10

M

MACFDI 5-12, 16

MACTOK 5-12, 16

MAdapterOprTimeStamp field 12-31

MAdapterResetCount field 12-31

major version number

LSL

configuration table 9-3

statistics table 9-8

MLID

configuration table 12-4

statistics table 12-27

protocol stack 3-2

configuration table 3-2

statistics table 3-5

MChecksumErrorCount field 12-30

MCustomCounterPtr field 12-27

media

independence 2-1

media specific counter 12-32

Ethernet 12-34

FDDI 12-36

MEON xxii

MEON_STRING xxii

MF_GRP_ADDR_SUP_BIT bit 12-18

MF_HUB_MANAGEMENT_BIT bit 12-18

MF_SOFT_FILT_GRP_BIT bit 12-18

MGenericCountersPtr field 12-27

MHardwareRxMismatchCount field 12-31

Micro Channel Architecture bus 11-13

minor version number

LSL

configuration table 9-3

statistics table 9-8

MLID

configuration table 12-4

statistics table 12-27

protocol stack 3-2

configuration table 3-2

statistics table 3-5

MLI (Multiple Link Interface)

defined 1-6

MLID (Multiple Link Interface Driver)

configuration table xxxiv, 12-2

control routines

overview 11-4

defined 1-5

message definition

document xix

multiple frame support,, see also frame, supporting

multiple types

removing

overview 11-5

statistics table xxxv, 12-24, 12-26

timeout detection 11-5

MLID_CONFIG_TABLE structure xxxiv, 12-2

MLID_REG structure xxxvi, 10-62

field descriptions 10-62

MLID_STATS_TABLE structure xxxv, 12-26

MLIDCFG_BestDataSize field 4-7, 12-5

MLIDCFG_BoardInstance field 12-4

MLIDCFG_BoardNumber field 12-4

MLIDCFG_CardName field 12-6

MLIDCFG_ChannelNumber field 12-14

MLIDCFG_CommandString field 12-13

MLIDCFG_Config field 12-13

MLIDCFG_DBusTag field 12-14

MLIDCFG_DIOConfigMajorVer field 12-14

MLIDCFG_DIOConfigMinorVer field 12-14

MLIDCFG_DMALine0 field 12-13

MLIDCFG_DMALine1 field 12-13

MLIDCFG_DriveMajorVer field 12-9

MLIDCFG_DriverLink field 12-9

MLIDCFG_DriverMinorVer field 12-9

MLIDCFG_Flags field 12-9, 12-20

MLIDCFG_FrameID field 12-6

MLIDCFG_FrameTypeString field 12-6

MLIDCFG_Interrupt0 field 12-12

MLIDCFG_Interrupt1 field 12-12

MLIDCFG_IOPort0 field 12-10

MLIDCFG_IOPort1 field 12-11

MLIDCFG_IORange0 field 12-11

MLIDCFG_IORange1 field 12-11

MLIDCFG_LinearMemory0 field 12-13

MLIDCFG_LinearMemory1 field 12-14

MLIDCFG_LineSpeed field 4-6, 12-7

MLIDCFG_LogicalName field 12-13
 MLIDCFG_LookAheadSize field 12-8
 MLIDCFG_MajorVersion field 12-4
 MLIDCFG_MaxFrameSize field 4-7, 12-5
 MLIDCFG_MemoryAddress0 field 12-11
 MLIDCFG_MemoryAddress1 field 12-12
 MLIDCFG_MemorySize0 field 12-11
 MLIDCFG_MemorySize1 field 12-12
 MLIDCFG_MinorVersion field 12-4
 MLIDCFG_ModeFlags field 6-5, 12-4, 12-15
 MLIDCFG_NodeAddress field 12-4
 MLIDCFG_PrioritySup field 12-8
 MLIDCFG_Reserved0 field 12-6
 MLIDCFG_Reserved1 field 12-8
 MLIDCFG_Reserved2 field 12-8
 MLIDCFG_ResourceTag field 12-13
 MLIDCFG_SendRetries field 12-9
 MLIDCFG_SharingFlags field 12-10
 MLIDCFG_ShortName field 12-6
 MLIDCFG_Signature field 12-4
 MLIDCFG_Slot field 12-10
 MLIDCFG_SourceRouting field 12-7
 MLIDCFG_TransportTime field 4-6, 12-6
 MLIDCFG_WorstDataSize field 4-7, 12-5
 MLIDDeRegistered function 7-12
 MLIDManagement function 18-20
 MLIDReset function 18-22
 MLIDShutdown function 18-24
 MM_C_HSM_BIT bit 12-16
 MM_CSL_COMPLIANT_BIT bit 12-16
 MM_DATA_SZ_UNKNOWN_BIT bit 12-16
 MM_DEPENDABLE_BIT bit 12-15
 MM_FRAG_RECEIVES_BIT bit 12-16
 MM_FRAGS_PHYS_BIT bit 12-16
 MM_MULTICAST_BIT bit 12-15
 MM_PREFILLED_ECB_BIT bit 12-16
 MM_RAW_SENDS_BIT bit 6-5, 12-16
 MM_SMP_BIT bit 12-16
 MMediaCountersPtr field 12-27
 MNoECBAvailableCount field 12-30
 MNumCustomCounters field 12-27
 MNumGenericCounters field 12-27
 MNumMediaCounters field 12-27
 mode flags field 12-15
 modifying
 filter mask 10-55
 monitoring
 packet transmission 18-30
 MPacketRxOverflowCount field 12-30
 MPacketRxTooBigCount field 12-30
 MPacketRxTooSmallCount field 12-30
 MPacketTxTooBigCount field 12-30
 MPacketTxTooSmallCount field 12-30
 MQDepth field 12-31
 MRetryTxCount field 12-30
 MS_HAS_CMD_INFO_BIT bit 12-21
 MS_NO_DEFAULT_INFO_BIT bit 12-20
 MS_SHARE_DMA0_BIT bit 12-20
 MS_SHARE_DMA1_BIT bit 12-20
 MS_SHARE_IRQ0_BIT bit 12-20
 MS_SHARE_IRQ1_BIT bit 12-20
 MS_SHARE_MEMORY0_BIT bit 12-20
 MS_SHARE_MEMORY1_BIT bit 12-20
 MS_SHARE_PORT0_BIT bit 12-20
 MS_SHARE_PORT1_BIT bit 12-20
 MS_SHUTDOWN_BIT bit 12-20
 MStatTableMajorVer field 12-27
 MStatTableMinorVer field 12-27
 MTotalGroupAddrRxCount field 12-31
 MTotalGroupAddrTxCount field 12-31
 MTotalRxMiscCount field 12-30
 MTotalRxOKByteCount field 12-31
 MTotalRxPacketCount field 12-30
 MtotalTxMiscCount field 12-30
 MTotalTxOKByteCount field 12-31
 MTotalTxPacketCount field 12-30
 multicast addresses
 adding 18-7
 disabling 18-11
 Ethernet 18-10
 maximum supported 18-9
 multicast addressing
 hardware 18-9
 support
 MLID 11-12
 multicast support
 protocol stack 4-7
 multicast transmission

- NET.CFG file 4-8
- multiple
 - operating systems
 - supporting 11-1
- multiple board support
 - protocol stacks 4-5
- multiple chained protocol stack 5-5
- multiple frame support
 - MLID 11-7
- Multiple Link Interface (MLI) 10-1
- Multiple Link Interface Driver (MLID)
 - configuration table 11-5
 - control routines 18-1
 - data structures 11-5
 - definition 11-2
 - design considerations 11-12
 - initializing 11-3
 - multicast addressing support 11-12
 - multiple frame support 11-7
 - optional functionality 11-12
 - portability 11-1
 - promiscuous mode support 11-12
 - recommended functionality 11-6
 - reenetrancy 11-6
 - source routing support 11-12
 - statistics table 11-5
 - transmit monitor 18-32
- Multiple Protocol Interface (MPI) 10-1
 - defined 1-3
- multiplexing
 - protocol stacks 2-1
- multiprocessing bit 12-16
- multiprocessor platform xvii

N

- nesting level B-4
- NET.CFG file
 - ``bind" entry 4-3
 - multicast addresses 4-8
- NetWare/Intel
 - building CHSM C-1
 - creating source file C-1

- linker definition file C-2
- NODE_ADDR structure xxiv
- NULL B-2, B-3

O

- ODI (Open DataLink Interface) sp 1-1, 1-2
- ODI_STAT_UINT32 status xxvii, 12-25
- ODI_STAT_UINT64 status xxvii, 12-25
- ODI_STAT_UNUSED status xxvii, 12-25
- ODISTAT enumeration xxiv
- offsetof macro B-3
- operating system
 - supporting multiple 11-1
- outstanding transmit requests
 - number of 6-3

P

- packet
 - destination
 - determining 1-4, 8-1
 - flow 1-6, 2-5
 - reception
 - priority level 5-11, 5-13
 - protocol stack
 - choosing a method 5-4
 - events 5-2
 - methods 5-4
 - methods of 2-9
 - overview 2-9
 - process 2-10
 - system handles 2-9
 - routing
 - to logical board 2-8
 - to protocol stack 2-5
 - transmission 11-4
 - monitoring 18-30
 - protocol stack 6-1
- packet size
 - maximum 4-6
- packets

- protocol stack
 - steps in accepting 5-26
- PConfigTableMajorVer field 3-2
- PConfigTableMinorVer field 3-2
- PCustomCountersPtr field 3-5
- performance
 - measuring
 - protocol stack 4-6
- Peripheral Component Interconnect (PCI) 11-13
- Personal Computer Memory Card Internati 11-13
- PGenericCountersPtr field 3-5
- PID_SIZE parameter xxiv
- PIgnoredRxPackets field 3-7
- platform
 - multiprocessor xvii
- PNumCustomCounters field 3-5
- PNumGenericCounters field 3-5
- portability
 - alignment B-5
 - assumptions B-5
 - data packing B-5
 - issues B-1
 - MLID 11-1
 - requirements xxi
 - rules B-1
- porting code B-1
- PProtocolLongName field 3-2
- PProtocolMajorVer field 3-2
- PProtocolMinorVer field 3-2
- PProtocolShortName field 3-2
- pragma B-2
- prescan protocol stack 5-4
 - chaining 2-10, 4-9
 - defined 2-10
 - deregistering 10-21
 - receive handler 5-22
 - registering 10-64
- prescan transmit method
 - protocol stack 6-2
- priority level
 - packet reception 5-11, 5-13
- priority sends 6-3
- priority transmits 6-3
- privileged time xx
- process time xx
- promiscuous mode
 - defined 11-12
 - diabling 18-26
 - enabling 18-26
 - support
 - MLID 11-12
- PromiscuousChange function 18-26
- PromiscuousStatus function 7-14
- PROT_ID structure xxiv
- Protocol ID (PID) 2-7
 - adding 4-4
 - conditions for 4-5
 - defined 2-7
 - registering 10-8
 - value
 - obtaining 4-5
- protocol receive complete handler
 - default stacks 5-29
 - prescan stacks 5-29
- protocol receive handler
 - bound stack 5-16
 - default stack 5-22
 - prescan stack 5-22
- protocol stack
 - accepting packet
 - general steps 5-26
 - binding to adapter 10-10, 10-12
 - binding to frame type 10-10, 10-12
 - binding to logical board 4-8
 - bound 5-4
 - defined 2-9
 - registering 10-69
 - chaining 2-10, 4-9
 - configuration table xxxiii, 3-1
 - customizing 4-5
 - default 5-4
 - defined 2-10
 - deregistering 10-18
 - registering 10-58
 - defined 1-2
 - independence 2-1
 - initialization 4-1
 - initializing 2-7

- line speed 4-6
- major version number 3-2
- maximum packet size 4-6
- measuring performance 4-6
- minor version number 3-2
- multicast support 4-7
- multiple board support 4-5
- multiple chained 5-5
- multiple frame support,, see also frame, supporting
 - multiple types
- multiplexing 2-1
- overview 2-1
- packet
 - reception
 - methods of 2-9
 - process 2-10
- packet receive events 5-2
- packet reception methods 5-4
- packet transmission 6-1
- prescan 5-4
 - defined 2-10
 - deregistering 10-21
 - registering 10-64
- prescan transmit method 6-2
- receive lookahead 4-8
- registering with LSL 4-2
- routing packet to 2-5
- send routine event 6-1
- statistics table xxxiii, 3-4
- transmit routine event 6-1
- unbinding from adapter 10-92
- unbinding from frame type 10-92
- protocol transmit complete handler 6-13
- protocol transmit handler
 - prescan stacks 6-10
- PS_BOUND_NODE structure xxxvi, 10-70
 - field descriptions 10-71
- PS_CHAINED_RX_NODE structure xxxvi, 10-59, 10-65
 - field descriptions 10-59, 10-66
- PS_CHAINED_TX_NODE structure xxxvii, 10-67
 - field descriptions 10-67
- PS_CONFIG_TABLE structure xxxiii, 3-1
- PS_STATS_TABLE structure xxxiii, 3-4

- PStatTableMajorVer field 3-5
- PStatTableMinorVer field 3-5
- PTotalRxPackets field 3-7
- PTotalTxPackets field 3-7

R

- raw send
 - ECB 6-5
- receive complete handler
 - default stacks 5-29
 - prescan stacks 5-29
- receive handler
 - receive lookahead 5-6
- receive lookahead xxxii, 4-8, 5-5, 5-6
 - receive handler 5-6
- reception
 - packet
 - priority level 5-11, 5-13
- reentrancy
 - MLID 11-6
- reentrant code
 - implementing multiple frame support in 11-7
- registering
 - bound protocol stacks 10-69
 - default protocol stacks 10-58
 - logical board with LSL 10-61
 - prescan protocol stacks 10-64
 - Protocol ID (PID) 10-8
 - protocol stacks with LSL 4-2
- RegisterMonitor function 18-30
- removing
 - MLID
 - overview 11-5
- required control procedures 11-2
- resetting adapters 18-22
- retries at sending packet 12-9
- revision level
 - CHSM 12-9
- routing
 - packet
 - to logical board 2-8
 - to protocol stack 2-5

S

- scheduling
 - AES events 10-80
- sending packets
 - from LSL to MLID 10-85
 - number of retries 12-9
- sending protocol information to IOEngin 10-87
- sending protocol information to other e 10-89
- service
 - dynamic method 4-3
 - explicit method 4-3
- servicing
 - events in the LSL hold queue 10-91
 - logical boards 4-3
- SetLookAheadSize function 18-38
- SFTIII
 - sending protocol information 10-87, 10-89
- SFTIII_EXCHANGE_NODE structure xxxvii
- SFTIII_STAT enumeration xxv
- shutting down adapters 18-24
- sizeof operator B-2
- source file
 - creating
 - NetWare/Intel C-1
- source routing
 - document xix
- source routing support
 - MLID 11-12
- Source SAP (SSAP) 6-6
- specification version number 8-2
- specification version string 8-2
- speed
 - topology 12-7
- Stack ID (SID) 2-7
 - defined 2-7
- STAT_TABLE_ENTRY structure xxvii, 12-25
- statistics counter
 - custom 12-24
 - generic 12-24
 - media specific 12-24
 - standard 12-24
- statistics table
 - LSL xxx, 9-7
 - major version number 9-8
 - minor version number 9-8
 - pointer to 10-33
 - media specific counter
 - description 12-32
 - MLID xxxv, 11-5, 12-24, 12-26
 - major version number 12-27
 - minor version number 12-27
 - protocol stack xxxiii, 3-4
 - major version number 3-5
 - minor version number 3-5
- structure
 - AES_ECB xxxii, 10-81
 - ECB xxxi, A-1
 - fragment xxxi, A-1
 - FRAGMENT_STRUCT xxxi, A-1
 - INFO_BLOCK xxviii, 10-7
 - LOG_BRD_STAT_TABLE_ENTRY xxviii, 9-7
 - LOOKAHEAD xxxii, 5-6
 - LSL_CONFIG_TABLE xxviii, 9-1
 - LSL_STATS_TABLE xxx, 9-7
 - MLID_CONFIG_TABLE xxxiv, 12-2
 - MLID_REG xxxvi, 10-62
 - MLID_STATS_TABLE xxxv, 12-26
 - NODE_ADDR xxiv
 - PROT_ID xxiv
 - PS_BOUND_NODE xxxvi, 10-70
 - PS_CHAINED_RX_NODE xxxvi, 10-59, 10-65
 - PS_CHAINED_TX_NODE xxxvii, 10-67
 - PS_CONFIG_TABLE xxxiii, 3-1
 - PS_STATS_TABLE xxxiii, 3-4
 - SFTIII_EXCHANGE_NODE xxxvii
 - STAT_TABLE_ENTRY xxvii, 12-25
- supporting
 - logical boards 2-1
 - multiple
 - operating systems 11-1
- system
 - handles
 - used by protocol stack in packet receipt 2-9

T

- timeout
 - MLID
 - overview 11-5
- timeout detection
 - MLID 11-5
- timing marker 10-31
- group addresses
 - TokenRing 18-10
- media specific counter
 - TokenRing 12-32
- multicast addresses
 - TokenRing 18-10
- TokenRing 18-10
- topology
 - speed 12-7
- translation limit B-4
- transmit monitor
 - MLID 18-32
- transmit routine event
 - protocol stack 6-1
- transmits
 - priority 6-3
- transmitting
 - packets 11-4
- TRN_AbortDelimiterCounter field 12-32
- TRN_ACErrCounter field 12-32
- TRN_BurstErrCounter field 12-32
- TRN_FrameCopiedErrCounter field 12-32
- TRN_FrequencyErrCounter field 12-32
- TRN_InternalErrCounter field 12-32
- TRN_LastBeaconType field 12-34
- TRN_LastRingID field 12-33
- TRN_LastRingStatus field 12-33
- TRN_LineErrCounter field 12-33
- TRN_LostFrameCounter field 12-33
- TRN_TokenErrCounter field 12-33
- TRN_UpstreamNodeAddress field 12-33

U

- UINT16 xxii
- UINT32 xxii
- UINT64 xxii
- UINT8 xxii
- Unbind function 7-19
- unbinding
 - protocol stack from adapter 7-19, 10-92
 - protocol stack from frame type 10-92

V

- vector number
 - interrupt 12-12
- version number
 - LSL
 - configuration table 9-3
 - statistics table 9-8
 - MLID
 - configuration table 12-4
 - statistics table 12-27
 - protocol stack
 - configuration table 3-2
 - statistics table 3-5
- version string
 - specification 8-2
- void* B-3
- volatile variable xvii

Trademarks

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.