ODI™ Developer's Guide

 APPLE COMPUTER, INC.

This manual is copyrighted by Apple or by Apple's suppliers, with all rights reserved. Under the copyright laws, this manual may not be copied, in whole or in part, without the written consent of Apple Computer, Inc. except in the normal use of software or to make a backup copy of the software. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased may be sold, given, or lent to another person. Under the law, copying includes translating into another language.

© Apple Computer, Inc., 1988

20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Apple, the Apple logo, LaserWriter, and Macintosh are registered trademarks of Apple Computer, Inc.

Ethernet is a registered trademark of Xerox Corporation.

ITC Avant Garde Gothic, ITC Garamond, and ITC Zapf Dingbats are registered trade- marks of International Typeface Corporation.

MicroChannel is a trademark of Internation Business Corporation.

MLI/MPI, MLI, MPI, and Link Support Layer are jointly owned trademarks of Apple Computer and Novell, Inc.

POSTSCRIPT is a registered trademark of Adobe Systems Incorporated.

Varityper is a registered trade-mark, and VT600 is a trademark, of AM International, Inc.

Simultaneously published in the United States and Canada.

2/21/88

Contents

PART II Writing Protocol Stacks for the MPI

Chapter 1 Introducing the Open Data-Link Interface

The Open Data-Link Interface is a new system jointly developed by Apple Computer, Inc. and Novell that provides unmatched flexibility for both network developers and end users. The Open Data-Link Interface includes the Multiple Link Interface (MLI™) and the Multiple Protocol Interface (MPI™). The MLI and MPI are the interfaces for network card drivers and protocol stacks to the Link Support Layer (LSL™). The LSL provides packet transfer between these interfaces in a way that allows different protocol stacks to use link-level drivers interchangeably and simultaneously. The Open Data-Link Interface puts an end to the need for one-driver to one-stack communication.

Benefits for the user

Imagine the inconvenience that could result if messaging had to be done without the assistance of a postal service. It's the same inconvenience that takes place when you're forced to buy a separate driver or interface card to support each stack on your network. The Open Data-Link Interface acts like a postal service and more, allowing a single driver to support any number of stacks—just like a single postal service supports many kinds messengers. Any driver written to the MLI/MPI specification can receive packets from any stack written to the specification. That means you don't have to buy, install, or maintain separate drivers or interface cards for each protocol in your network. One driver will handle all the protocols.

Under the Open Data-Link Interface data packets need only be directed to a special module, the Link Support Layer, instead of a specified driver or stack. The Link Support Layer is like the postal service of this specification because it correctly steers inbound and outbound packets to the specified stacks and drivers. That means your system is responsible only for directing data packets to the Link Support Layer instead of reaching the full distance to the protocol stack or some specified driver. Just as the postal service knows how to deliver messages directly to you, the Link Support Layer knows how to deliver packets directly to the driver specified by the protocol stack. Similarly, the Link Support Layer knows how to deliver packets to the protocol stack specified by the driver.

You can benefit directly from using the Open Data-Link Interface in the following ways:

- You can expand your networking system by adding networking protocols without having to add more interface cards.
  You don't have to buy different cards for different systems. If you need to switch back and forth between environments such as TCP/IP and Netware®, you need only one interface card.
- You protect your investment.
  With the Open Data-Link Interface, the one essential driver can be used in any workstation accessing any environment. Once you invest in an Open Data-Link Interface driver, you're protected because no matter how your network changes, the OLI driver will always communicate with any stack written to the OLI.
- You spend less time and money on support.
  With just one driver supporting many protocol stacks, you have fewer components to support. When you take away all the additional drivers or interface cards except the one Multiple Link Interface Driver (MLID) you would need to support the variety of protocol stacks on your system. All the hardware you removed represents how much less hardware you have to support.

Benefits for the developer

By writing drivers and stacks that follow the MLI/MPI specification, you seize a range of benefits for yourself that will ultimately profit the user community. The MLI/MPI specification gives you a standard by which to design network card drivers and protocol stacks that use the MLI/MPI interface. Writing to the specification guarantees that the drivers and stacks work with each other. You only need to develop once, develop correctly, and you will ultimately save development resources and time to market.

Hardware developers who write drivers to this specification can have their drivers transparently communicate with any protocol stack written to this specification. Similarly, protocol stack developers who modify their stacks to meet this specification can have their stacks communicate with any driver written to this specification.

As a developer, you can benefit directly from writing to the Open Data-Link Interface in the following ways:

- You reduce the labor in your development process.
  When you write your drivers or protocol stacks according to the MLI/MPI specification, you labor only once because all other MLI/MPI compliant systems will work with yours. You don't have to develop with "one-driver one-stack" communication in mind. You only have to develop your drivers and stacks for communication with the Link Support Layer.
- You write a driver that has a full-feature set.
  The driver you write to the MLI/MPI specification contains a full feature set that goes beyond transporting packets to the Link Support Layer. Your driver can also call on the Link Support Layer for any number of support commands including name lookup, registration, and statistics on any other drivers in the system. Under the Open Data-Link Interface, your driver is capable of supporting multiple protocol stacks instead of just one.
- Your protocol stacks have access to all network interface cards.
  The stack you write to the MLI/MPI specification automatically gives you compatibility to any card written to the same specification. This means there are more cards that can use your stack.
- Your protocol stacks can co-exist transparently with other stacks written to the MLI/MPI specification.
  Because all the stacks in an Open Data-Link Interface System are written according to the same specification, they can co-exist without conflict. ■

MLI/MPI module specifics

In order to implement the multiplexing environment of the Open Data-Link Interface, the MLI/MPI design uses the following three modules:

1. The Multiple Link Interface Driver (MLID), supplied by the interface card manufacturer.

   This module implements the actual interface to the card. The MLID must define the following entry points:

   - □ a Send Entry Point
     This allows protocol stacks to transmit through the MLID's interface card.
   - □ a Driver (MLID) Control Entry Point
     This manages all the miscellaneous informational and control requests made of the MLID by stacks.

2. The Link Support Layer (LSL), supplied by Apple Computer and Novell.

   The LSL is responsible for coordinating communication between the MLIDs and the stacks. In addition, it provides many common support routines needed by MLIDs and stacks. It is also the central point where MLIDs and stacks conduct registration to identify each other.

   The level of communication provided by the Link Support Layer between stacks and drivers allows a fully multiplexed environment.

   The LSL has the following four entry points:

   - □ Initialization Entry Point
     This entry point is where MLIDs and stacks register themselves with the LSL and exchange entry point and configuration information.

- □ MLID Support Entry Point

  This entry point allows the MLID to use the event-handling, timer, and queueing facilities in the Link Support Layer.

- □ Protocol Stack Support Entry Point

  This entry point allows stacks (and applications, under MS-DOS) to gain access to services from the LSL. Additionally, it allows the stacks to queue packets, to schedule timer events, to get Receive buffers, to perform stack ID-to-physical and physical-to-stack ID mappings, to obtain error information, and to communicate directly with other stacks and MLIDs.

- □ General Services Entry Point

  This entry point allows stacks (and applications, under DOS) to access some general services from the LSL, such as memory management functions. This entry point additionally provides a generic communications medium so modules can add new general services to the entry point.

3. The protocol stack.

   This module is the implementation of a protocol. The MLI/MPI specification does not define how an application communicates with a protocol stack because higher-level services vary from stack to stack. The MLI/MPI specification details how a stack will communicate with the LSL and, ultimately, the MLIDs. (The protocol stack receives packets from the Link Support Layer and then processes these received packets. The protocol stack also creates outgoing packets and transmits them through the Link Support Layer. From the Link Support Layer, the packets are delivered to the MLID that was requested by the protocol stack.)

   The processing of these packets allows higher-level services (such as registration and lookup of entity names, and transaction processing) to exist. Because each stack maintains its own set of higher-level services, the availability of a particular service will vary from stack to stack.

   The stack contains the following five entry points:

   - □ Protocol Stack Control Entry Point

     This is the entry point where stacks can call each other and exchange configuration information and statistics about their operation.

   - □ Receive Entry Point (Optional)

     This is the entry point where stacks normally receive incoming packets from MLIDs.

   - □ Default Receiver Entry Point (Optional)

     This is an alternate entry point for receiving incoming packets from MLIDs.

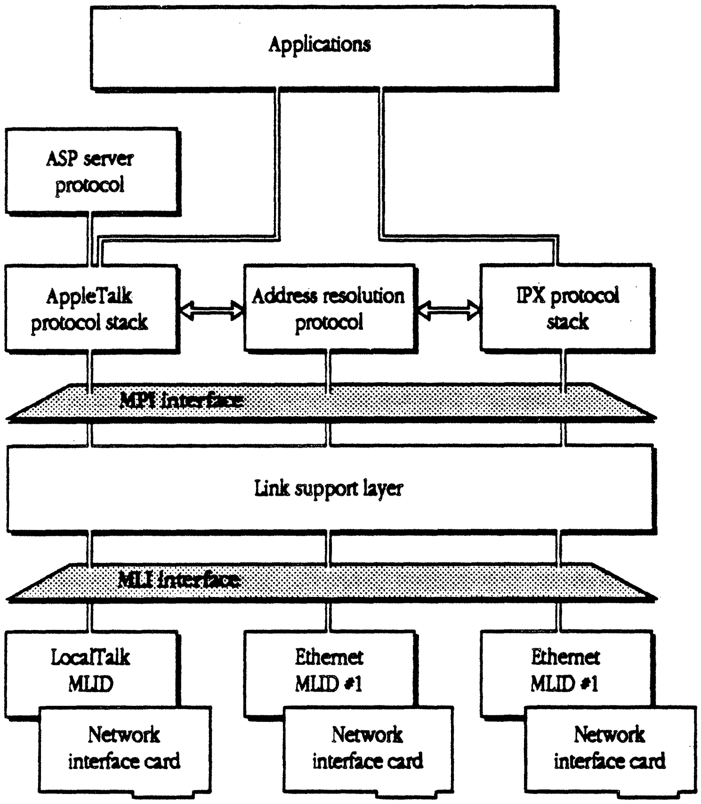
   - □ PreScan Entry Point (Optional)

     This is a special entry point for stacks that need to filter or preview incoming packets before they are routed by the Link Support Layer.

   - □ Application Entry Point

This is the entry point where applications call the stack. The definition of this entry point is dependent on the specific protocol stack. As a result, the Application Entry Point is undefined in the MLI/MPI specification.

The following diagram provides a visual overview of a sample network system. Arrows indicate optional communication paths.



Protocol stack independence from MLIDs

In order for protocol stacks to achieve independence from the link-level envelope of the underlying media, the following assumptions are made in the current implementations:

1. Link-level physical addresses can be uniquely expressed in 48 bits or less.
2. Link-level envelopes, which have a field for demultiplexing of incoming packets (called the protocol ID in this specification), can uniquely express this field in 48 bits or less.
3. A 48 bit physical address corresponding to 0FFFFFFFFFFFFH is considered to be a broadcast request by all MLIDs.

The user can configure the system to recognize a particular protocol stack by specifying a number (up to 48 bits) that describes the protocol ID for the stack. The user (or configuration program) would enter this 48-bit number and the name of its corresponding stack into the NET.CFG file. The LSL would then be able to route incoming packets from an MLID to the specified protocol stack.

Every MLID registers with the Link Support Layer. As part of the registration process, the MLID tells the LSL the name and protocol ID for each of the stacks that the MLID recognizes. The stack name and protocol ID are usually obtained from the NET.CFG file. The LSL then assigns a stack ID to each known protocol name. The LSL also provides each MLID with a "board number" (or multiple board numbers if the MLID is written to handle multiple link interface cards).

Stacks identify themselves by making the RegisterStack call to the Link Support Layer. The protocol stack can then find out the Protocol ID for a given board number by making the GetPIDfromStackIDBoard call to the Link Support Layer. The protocol ID is used in a Send Packet command to tell the MLID which protocol ID to put in the link-level envelope.

By using the board number, a stack specifies which MLID will transmit a packet. The board number also allows a stack to identify the MLID from which an incoming packet originated.

Developing MLIDs

Writing to the MLI/MPI specification gives customers a flexible solution by allowing them to mix and match network cards and services. But you need to have your MLIDs certified before they can be marketed. If you are interested in developing to this specification, contact your authorized Novell representative to get a standard developer's kit.

Chapter 2 **The Link Support Layer**

This chapter describes the services provided by the Link Support Layer to MLIDs and protocol stacks.

The Link Support Layer contains special services called support routines to help the function of both MLIDs and protocol stacks. By using the calls specified in the chapters on support and controi procedures, MLIDs and protocol stacks can access these services. These routines are designed to be very efficient and to use a minimum amount of program stack space. Calls are made to the Link Support Layer at the following four entry points:

- Protocol Stack Support Entry Point
- MLID Support Entry Point
- General Services Entry Point
- Initialization Entry Point ■

Protocol Stack Support Entry Point

This entry point is a far-call address that can dispatch all Link Support Layer commands available to a stack. Many of these commands are internally equivalent to those provided by the MLID Support Entry Point. However, these LSL commands are dispatched through the Protocol Stack Support Entry Point so that all commands available to a protocol stack are available through this entry point.

The Protocol Stack Support Entry Point provides procedures for the following:

- to allow a stack to obtain and return Event Control Blocks (ECBs are Buffers used to send or receive packets, or to schedule timers.)
- to enqueue and recover ECBs for later use
- to register and deregister the stack
- to provide timing services
- to determine stack and protocol IDs
- to get statistics
- to bind with MLIDs
- to transmit packets through an MLID
- to provide other services that allow stacks to obtain information about MLIDs and other protocol stacks. The Link Support Layer maintains a list of all active stacks and MLIDs

The Link Support Layer uses the caller's program stack space to accomplish its work. Packet reception and Event Service Routines (ESRs), which are called when an ECB event completes for Asynchronous Event Services (AES), which are timing routines, are dispatched on interrupt-time program stacks. As a result, program stack swapping under MS-DOS must be used for any routines that use more than about 32 bytes of program stack space.

MLID Support Entry Point

This entry point is a far-call address that can dispatch all Link Support Layer commands available to an MLID. Many of these commands are internally equivalent to those provided by the Protocol Stack Support Entry Point but are dispatched through the MLID Support Entry Point. As a result, all commands accessible to an MLID are available through this entry point.

This entry point provides procedures for the following events:

- to allow an MLID to obtain and return ECBs for packet reception

- to enqueue and recover Transmit ECBs for later use
- to hold Receive ECBs for processing by the Link Support Layer
- to register and deregister the MLID
- to provide timing services
- to add Protocol IDs
- to start and end critical sections

General Services Entry Point

This entry point is available to all protocol stacks. Under MS-DOS, it is available to applications. (The General Services Entry Point is not available to MLIDs.) It contains a small memory manager and some hooks to allow other stacks (and applications, under MS-DOS) to add new commands that can be accessed through the General Services Entry Point. This ability to add new commands is intended to allow stacks/applications to find each other easily and exchange entry points.

What You Need to Know

Before you get started writing for the MLI/MPI, you should understand the following characteristics:

- A limited number of MLIDs are supported by the Link Support Layer. This number can be found by making the GetMLIDControlEntry call. Make this call incrementing the board number parameter from 0 until the NO_MORE_ITEMS error code is returned.
- A limited number of stacks are supported by the Link Support Layer. This number can be found by making the GetProtocolControlEntry call. Make this call incrementing the Stack ID parameter from 0 until the NO_MORE_ITEMS error code is returned.
- All MLIDs must be able to transmit and receive packets of at least 586 bytes, not counting the media header envelope. If the media does not support this requirement, the MLID *must* implement a strategy to join packets to give the Link Support Layer and protocol stacks the impression that the MLID can transmit and receive at least this packet size. (Refer to the description of the ReturnECB call in Chapter 7 for some hints on implementing this strategy.)
- All version numbers in the specification are decimal. A major version number of 1 and a minor version number of decimal 31 is intended to imply version 1.31.
- On 808X implementations, all calls to the Link Support Layer will preserve the Direction Flag.

Chapter 3 **Link Support Layer General Services**

This chapter details the general services that the Link Support Layer provides through its General Services Entry Point. The general services of the Link Support Layer are available to protocol stacks (and other Ring 0 processes under OS/2), and to applications under MS-DOS. Register BX is used to specify the desired general service command.

The values of BX are allocated in the following way:

0000H to 1FFFH	General services provided by Apple and Novell
2000H to 3FFFH	General services administered by Apple
4000H to 5FFFH	General services administered by Novell
6000H to 7FFFH	General services administered by Apple and Novell
8000H to FFFFH	Available for general use

All general services added to the Link Support Layer in the range 8000H to FFFFH must support AX = 0 as an incoming parameter, and must return AX = 0. In addition these general services must return the address of a description record (described next) in ES:SI. As an option, DX:BX can return an entry point for the general service.

The following structure represents a Description record:

Offset	Length	Description
0	11	0-terminated General
12	1	name (no leading length byte)
13	1	month
14	1	day
15	1	year
15	1	major version
16	1	minor version

If the general service is implemented as a TSR program under DOS, the PSP of the TSR should be stored in the word immediately preceding this structure.

In addition, the function AX = 1 is reserved for general service removal. When this call is made, the service should determine if it can be removed. If it can, the general service should restore and clean up areas such as memory allocation, interrupt vectors, and making the Remove General Service call; the general service should also return AX = 0. It should return AX = BAD_COMMAND if removal is not supported. FAIL is returned in AX if removal is supported but the service cannot be removed at this time. In this case, ES:SI should point to a 0-terminated string describing why the general service cannot be removed. A 0-terminated string has no leading lengh byte.

■

AllocMemory

This command allocates memory to the protocol stack. The memory can be freed when it is no longer needed by using the FreeMemory command.

Assumes:

- BX = 0
- Registers Preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- CX contains the number of bytes required

Returns:

- Interrupts: Enabled on exit
- AX = 0; memory was available and ES:SI will point to the allocated memory (in OS/2, the allocated memory is located in the GDT)
- AX < 0; an error occurred:

AX = OUT_OF_RESOURCES	if the memory pool does not have enough memory to satisfy a request
AX = BAD_PARAMETER	if a request needs more memory than allowed. The maximum number is implementation dependent but will always be greater than 32K and less than 64K. For MS-DOS, this number is 65516 bytes

◆ *Note*. This command is not avialable under OS/2. Use the memory support procedures defined by OS/2.

FreeMemory

This command returns memory that was allocated by the AllocMemory command to the memory pool .

Assumes:

- BX = 1
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI contains a pointer to the allocated memory.

Returns:

- Interrupts: Enabled on exit

- AX = 0; the memory was returned to the pool
- AX < 0; an error occurred:

AX = BAD_PARAMETER	if the pointer returned did not come from the memory pool

◆ *Note*: This command is not available under OS/2. Use the memory support procedures defined by OS/2.

ReAllocMemory

This command allows reduction of the size of an allocated memory block, returning some of the memory to the pool. If CX is passed in 0FFFFH, the size of a block of memory can be discovered. In addition, CX always returns the actual size of the block. The size may be more than requested as a result of quantization in the memory manager.

Assumes:

- BX = 2
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- CX contains the number of bytes to which the memory block is to be resized
- ES:SI contains the pointer to the block of memory to be resized

Returns:

- Interrupts: Enabled on exit
- CX = size of memory block
- AX = 0; the resizing was done
- AX < 0; an error occurred:

AX = BAD_PARAMETER	if the pointer returned did not come from the memory pool
AX = OUT_OF_RESOURCES	if more memory than was in the original block of memory is requested

◆ *Note*: This command is not available under OS/2. Use the memory support procedures defined by OS/2.

MemoryStatistics

This command returns the current status of the memory pool.

Assumes:

- BX = 3
- Registers preserved: All except AX, BX
- Interrupts: Unspecified
- ES:SI contains a pointer to six words

Returns:

- Interrupts: never changed from the way they entered
- AX = 0

ES:SI points to six words as follows:

word 0:	number of paragraphs of memory available
word 1:	number of paragraphs of memory in use
word 2:	number of paragraphs in the largest block of memory
word 3:	number of available blocks of memory
word 4:	number of bytes overhead per allocation
word 5:	number of bytes minimum allocation

◆ *Note*: This command is not available under OS/2. Use the memory support procedures defined by OS/2.

AddMemoryToPool

This command allows a protocol stack or a terminate-and-stay-resident (TSR) application to give more memory to the buffer pool.

Assumes:

- BX = 4
- Registers preserved: DS, SS, SP, and BP
- Interrupts: enabled on entry
- CX contains the number of paragraphs to add to the pool
- ES contains the segment address to add to the pool
- once memory is given to the pool, it can never be removed

Returns:

- Interrupts: enabled on exit
- AX = 0; no errors are possible

◆ *Note* This command is not available under OS/.2. Use the memory support procedures defined by OS/2.

AddGeneralService

This command allows protocol stacks, other Ring 0 processes under OS/2, and TSR applications under DOS to add new commands to the General Services Entry Point. The entry point (*entry*) of the new command will be called whenever the General Services Entry Point is entered with a command code matching *command* in the passed structure. This command is especially useful for enabling a process to locate other pieces of itself. For example, a stack could register itself to allow another piece of the protocol stack, which is not always loaded, to find and communicate with the master stack.

Before a new general service can be added to the General Services Entry Point, an available command code in the range of 8000H—FFFFH needs to be located. This is done by making a General Service Entry Point call with the desired command code in BX and with AX set to 0. If the command code is already in use, AX is returned still set to 0, and ES:SI will contain a description record address. The description record may be examined to determine what general service is installed for this command code. If the command code is not in use, AX is returned containing the BAD_COMMAND (8008H) error code. The program that is installing the new General Service can then execute an AddGeneralService command to actually add the new service.

Assumes

- BX = 5
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to one of the following structures (which must be in the GDT under OS/2):

Offset	Length	Description
0	4	reserved
4	4	service entry point
8	2	command number

Returns:

- Interrupts: Return enabled .
- AX = 0; the command was added to the general services supported

- AX < 0; an error occurred:

AX = DUPLICATE_ENTRY	if there is already a general service with the requested command code

◆ *Note:* The memory that ES:SI points to is being used by the Link Support Layer until the general service is removed with the RemoveGeneralService call.

RemoveGeneralService

This command allows the removal of a general service that was added with the AddGeneralService command call.

Assumes:

- BX = 6
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to one of the following structures, which must be in the GDT under OS/2:

Offset	Length	Description
0	4	reserved
4	4	service entry point
8	2	command number

This structure must be the one passed to the AddGeneralService call and not a copy of it.

Returns:

- Interrupts: Return enabled
- AX = 0; the command was removed from the general services supported
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	if there is not a general service matching the passed structure

Part I Writing Drivers for the MLI

Chapter 4 **MLID Operations**

This chapter briefly describes the operation of an MLID. The MLID is responsible for receiving packets at the link layer and routing them to the Link Support Layer as well as transmitting packets through the interface adapter. The MLID normally consists of the following three parts:

- an MLID Control Entry Point handler
- a Send Entry Point handler
- an Interrupt Service Routine (ISR)

This chapter provides a description of the Send Entry Point handler and the ISR The operation of the MLID Control Entry Point handler is discussed in Chapter 6, "MLID Control Procedures." ■

The Send Entry Point Handler

The Send Entry Point handler gets the Event Control Block (ECB) in ES:SI and may modify all registers except SS, SP, DS, and BP. The following example, written in pseudocode, shows one way to implement the handler:

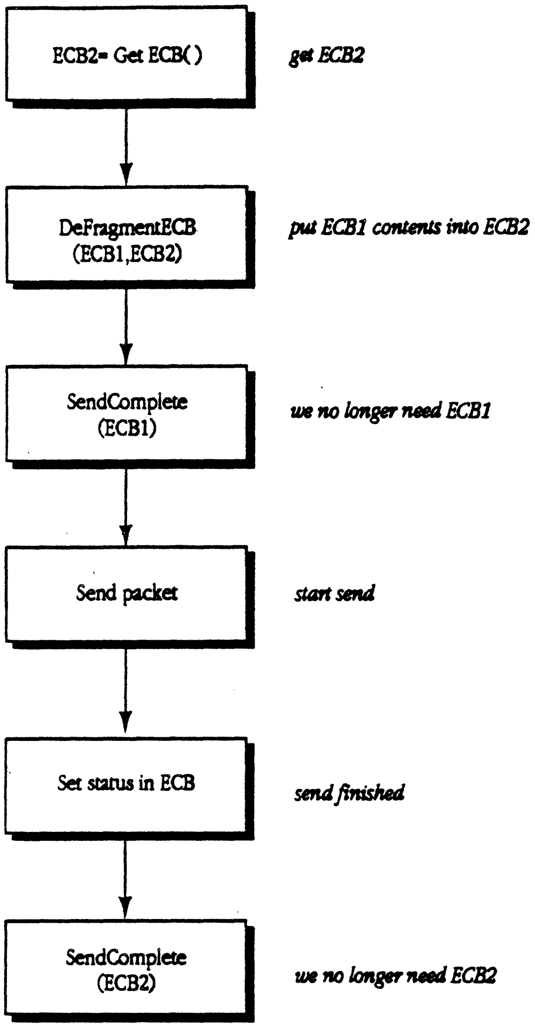
```
If (Shut down)

{
    Set AX = NO_SUCH_DRIVER
    Set Status in ECB = NO_SUCH_DRIVER
    Return
}

if (send_busy_flag)

{

        EnqueueSend

}

else

{

        set send_busy_flag

        if (ECB.StackID == OffffH)

        {

                start raw mode send

        }

        else

        {

                create Link-Layer envelope

                start send

        }

                    :
```

```
        (transmit  completes)

                        :

        increment  TotalTXPackets  statistic

        set  Status  in  ECB  to  0  or  appropriate  error  code  (see  note
        following  the  example)

        call  SendComplete  with  the  ECB;  (see  note  following  the  example)

        clear  send_busy_flag;

}
```

◆ *Note:* If the transmitter is asynchronous (that is, if it does not wait for completion), the *Status* is set and *SendComplete* is called from the ISR that services the "transmit complete" interrupt. The ECB address should be saved before leaving the transmit routine so that the correct ECB can be returned by *SendComplete* when the transmit is complete.

For MLIDs that must use DMA, *GetECB* followed by *DeFragmentECB* can be used to obtain an ECB that is fully defragmented and does not cross a 64 Kilobyte (KB) DMA boundary. In this case, *SendComplete* should be called immediately after the *DeFragmentECB* for the original ECB. In addition, *SendComplete* should be called after the send completes for the defragmented ECB. The following programming example shows how *GetECB* can be used for MLIDs that use DMA:

```
        call  GetECB  to  obtain  a  DMA-capable  ECB

        call  DeFragmentECB  to  copy  and  defragment  the  original  ECB  to
the  new  one

        call  SendComplete  on  original  ECB  to  return  it  to  the  buffer  pool

        start  send  or  start  raw  mode  send

                        :

        (transmit  completes)

                        :

        set  Status  in  ECB  to  0  or  appropriate  error  code

        call  SendComplete  on  defragmented  ECB

                        :
```



The Interrupt Service Routine

The following example shows one way the ISR might be implemented:

```
push all registers

call StartCriticalSection if needed

shut off board source of interrupts

re-arm the interrupt system
```

```
/* First check for and process incoming packets. */

if (packet_received)
{
    increment TotalRXPackets in statistics
    if (fatal errors occurred in the packet)
    {
        increment appropriate statistic
        throw away packet
    }
    else
    if (packet came from myself)      /* NOTE: ignore packets from self */
    {
        throw away packet
    }
    else
    if (packet too large or too small)
    {
        increment appropriate statistic
        throw away packet
    }
    else
    {
        call GetECB
        if (ECB available)
        {
```

```
                        read packet into ECB

                        set ImmAddr in ECB to the source address from the packet

                        set ProtoID in ECB to the protocol ID in the packet, or 0 if
                        the packet does not have a protocol ID

                        set BoardNo in ECB to the board number of the board which
                        received the packet

                        set FragCnt in ECB to 1

                        set SendLen to length of the data in the packet  (not counting
                        envelope)

                        set FragLen1 to length of the data in the packet (not counting
                        envelope)

                        set FragPtr1 to start of the data in the packet (immediately
                        following envelope)

                        call HoldRcvEvent with the ECB

                }

                else

                {

                        throw away packet

                        increment NoECBsAvail in statistics

                }

        }

}


/*

        If transmit_complete generates an interrupt, check here for
        transmit completion.      (See later note.)

*/




/*
```

```
        Here, we can optionally loop back to the top and check for more
        received packets or transmit completions.

*/




/* Now check for queued transmits. Send a queued packet if the
transmitter is available. */



if (not send_busy_flag)

{

        call GetNextSend

        if (a send event exists)

        {

                set send_busy_flag

                start send

                set Status in ECB to 0 or error code (see note under the
                Send Entry Point Handler)

                call SendComplete with the ECB

        }

}

cli

turn board interrupts back on

call EndCriticalSection or call ServiceEvents

pop all registers

iret
```

- ◆ *Note:* If the transmitter is asynchronous, the ISR may also have to handle *transmit complete* interrupts. If so, the following processing should be done in the ISR. (The previous example shows the internal structure of the ISR.)

```
                    :

if (transmit_complete)

{

       increment TotalTxPackets statistic (if not already incremented
       at transmit start time)

       set Status in ECB to 0 or appropriate error code

       call SendComplete on transmitted ECB

       clear send_busy_flag

}

                    :
```

Chapter 5 **MLID Initialization**

This chapter describes the initialization process for an MLID. MLIDs initialize themselves when the MLID loads itself in the computer's system. The MLID must be initialized before it can send and receive packets on the network. ■

The installation process of an MLID occurs in the following stages:

1. The MLID registers with the Link Support Layer. With OS/2, registration occurs when the MLID sends an IOCTL command to the LINKSUP$ device using the general IOCTL command (DosDevIOCtl) with a function category of 0A1H and a function code of 1. However, with DOS, the Link Support Layer is a TSR program. As a result, the Link Support Layer's Initialization Entry Point on a DOS-based network is found using the INT 2FH multiplexing address. The exact procedure for doing this is described in Appendix J.
2. The MLID reads the NET.CFG file and fills in the MLID Configuration Table with the necessary information. (See Appendix A for the format of the table and Appendix H for the format of the NET.CFG file.)
3. The MLID calls the Link Support Layer Initialization Entry Point with the following information:

BX = 1 MLI initialization function code

ES:SI Points to a table with the following information:

Offset	Bytes	Description
0	4	Ring 0 address of the MLID Send Entry Point. All packets to be sent on the network will be sent through this address
4	4	Ring 0 address of the MLID Control Entry Point
8	4	Address of the MLID Configuration Table valid at the time this call is made

DS:DI Address of four words in memory for the Link Support Layer to return configuration information into, in the following format:

Offset	Bytes	Description
0	4	Ring 0 Address of the MLID Support Entry Point of the Link Support Layer
4	2	Board number assigned to the MLID
6	2	Maximum buffer size of receive ECBs in the system

◆ *Note:* With OS/2, the parameters are sent in the IOCTL parameter buffer and returned in the IOCTL data buffer.

4. At this point, the developer should initialize the hardware. If the hardware fails, make the DeRegisterMLID call to the MLID Support Entry Point to remove the MLID from the Link

Support Layer's list of MLIDs. The process should then be terminated and an error message sent to the user.

5. The MLID informs the Link Support Layer about the Protocols the MLID can process. The protocols are processed using the AddProtocolID call defined in Chapter 7. Only protocol IDs mentioned in NET.CFG should be added, since there are a limited number of protocol stacks supported by the Link Support Layer.
6. The MLID terminates to the operating system and remains resident. At this point, the driver is installed in the computer's system and is able to begin sending and receiving packets.

Chapter 6 MLID Control Procedures

This chapter describes procedures that must be written for the MLID so that it can support protocol stacks.

To call the MLID control procedures, place a function code into BX and call the MLID Control Entry Point. The MLID Control Entry Point becomes available to the Link Support Layer at initialization time. The return value in AX will always be generated so that the Z and S flags are set correctly. AX will be 0 (and the Z flag set, and S flag clear) if the call was completed with no error. AX will be less than 0 (and the Z flag clear and S flag set) if the call was completed with an error. The value of AX will indicate the error. If an MLID does not support one of the following calls, it must return BAD_COMMAND in AX.

The following commands are provided by MLIDs:

■ GetMLIDConfiguration	this call must be supported
■ GetMLIDStatistics	this call must be supported
■ AddMulticastAddress	support for this call is optional
■ DeleteMulticastAddress	support for this call is optional
■ ReceptionControl	this call must be supported
■ MLIDShutdown	this call must be supported
■ MLIDReset	this call must be supported
■ CreateConnection	support for this call is optional. The correct error code must be returned

■ RemoveConnection	support for this call is optional. The correct error code must be returned
■ AddPromiscuousSourceFilter	support for this call is optional
■ AddPromiscuousDestinationFilter	support for this call is optional
■ ClearPromiscuousFilters	support for this call is optional
■ DriverPoll	support for this call is optional (OS dependent) ■

GetMLIDConfiguration

This command allows a protocol stack to determine the configuration of an MLID.

Assumes

- BX = 0
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- ES:SI returns a pointer to the *MLID Configuration Table* (see appendix A for a description of this table)
- AX=0; no errors are possible

GetMLIDStatistics

This command returns a pointer to the MLID Statistics Table describing statistics of the MLID, such as the number of transmitted and received packets.

Assumes

- BX = 1
- Registers preserved: DS, SS, SP, and BP

- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- ES:SI points to the MLID Statistics Table, the format of which is described in Appendix D
- AX = 0; no errors are possible

AddMulticastAddress

This command adds a multicast address to the MLID address list. Once in this list, packets with this address can be accepted as valid.

The MLID must maintain a count of the number of times that an address is added. This allows multiple protocol stacks to add the same multicast address. (See DeleteMulticastAddress later in this section.)

Assumes

- BX = 2
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry

  ES:SI points to a 6-byte multicast address to add to the MLID multicast list

Returns

- Interrupts: Enabled on exit
- AX = 0; the MLID added the multicast addresses; once in this list, packets with this address will be accepted as valid. Use of the AddMulticastAddress call *does not* automatically enable multicast reception. To enable multicast reception, use the Reception Control command described later in this chapter
- AX < 0 if an error occurred:

AX = BAD_COMMAND	the MLID does not support multicast addressing
AX = OUT_OF_RESOURCES	the MLID is out of room to add another multicast address
AX = BAD_PARAMETER	the address pointed to by ES:SI is not a valid multicast address

◆ *Note*: Use of the AddMulticastAddress call does not automatically enable multicast reception. The Reception Control command (described later in this chapter) must be used to enable multicast reception.

DeleteMulticastAddress

This command removes an instance of a multicast address from the MLID's list of addresses.

When an MLID receives the DeleteMulticastAddress command, the MLID must decrement its count of the number of times an address was added and only remove the address from its internal tables when the counter decrements to 0. If a multicast addresss is removed, a packet with this address will no longer be accepted as valid.

Assumes

- BX = 3
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to a 6-byte address to remove from the multicast address list

Returns

- Interrupts: enabled on exit
- AX = 0; the MLID removed the multicast address from its list
- AX < 0; an error occurred:

  AX = BAD_COMMAND the MLID does not support multicast addressing

  AX = ITEM_NOT_FOUND the multicast address was not found in the MLID's valid addresses list

ReceptionControl

This command allows a protocol stack to determine or set which packet types an MLID will receive by means of the bit settings passed in AX.

At a minimum, MLIDs must support Bit 0 being set. In order to be useful with most protocol stacks, the MLID should also be able to support broadcast packets (Bit 2 being set).

Assumes

- BX = 4
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry
- CX = 0. This setting indicates a read function
- CX is greater or less than 0. This setting indicates a write function

AX contains a bitmap indicating the types of incoming packets the MLID should accept:

Bit 0 = accept packets to the MLID's address

Bit 1 = accept packets to the MLID's multicast list

Bit 2 = accept broadcast packets

Bit 3 = accept all packets (promiscuous)

Returns

- Interrupts: Enabled on exit
- AX = 0; no error occurred
- AX < 0; an error occurred:

AX = BAD_PARAMETER	the MLID does not support one or more of the settings you requested in AX

- BX is equal to the reception control setting after the function was executed even if an error occurred.

◆ *Note*: Promiscuous address filters (see AddPromiscuousSource, AddPromiscuousDestination, and ClearPromiscuousFilters later in this chapter) are not affected by changing reception control in and out of the promiscuous mode.

MLIDShutdown

This command allows a protocol stack to shut down an MLID. If this call is supported by the MLID and invoked, the MLID should unhook all interrupts that it has intercepted. In this way, if the MLID is removed from memory, there will be no adverse effects. The MLID must fail all incoming transmit requests with the NO_SUCH_DRIVER error code. In addition, the driver should flush its send queue by repeatedly calling GetNextSend and Send Complete (with a CANCELLED error code in the ECB Status Field) until the queue is empty. The driver should also set Bit 0 in the share flag of the MLID Configuration Table.

Assumes

- BX = 5
- Registers preserved: DS, SS, SP, and BP
- AX = 0 if the caller wants the MLID both to shutdown its hardware and de-register itself.
- AX ≠ 0 if the MLID should only shut down its hardware and unhook its interrupts.
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- AX = 0. The MLID successfully shut down its hardware. Set Bit 0 in the the share flag
- AX < 0 if an error occurred:

AX = FAIL	the MLID cannot shut down its hardware
AX = BAD_COMMAND	the MLID does not support this command

▲ **Caution:** In order to remove the MLID safely from memory, the DeRegisterMLID call must be made to the Link Support Layer. ▲

MLIDReset

This command instructs the MLID to reinitialize its hardware and prepare to become operational. This command should also install its interrupt vectors needed for MLID operation if they are not already installed.

Assumes

- BX = 6
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- AX = 0; the MLID successfully reinitialized the interface card
- AX < 0; an error occurred:

AX = FAIL	the MLID cannot restart because of a hardware or software failure

◆ *Note:* If it already has been initialized, the MLID should reset the hardware.

CreateConnection

This command tells an MLID that the protocol stack will establish a lengthy connection with the address to which ES:SI is pointing. CreateConnection allows the MLID to more efficiently handle operations such as caching source routes. However, the MLID should still function properly even if this call is never made.

Assumes

- BX = 7
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to a 6-byte address to create a connection with it

Returns

- Interrupts: Enabled on exit
- AX = 0; the MLID successfully connected to this address
- AX < 0; an error occurred:

AX = FAIL	the connection was not established (the meaning of this error code is undefined for this version of the specification)

RemoveConnection

This command tells an MLID that the protocol stack will no longer maintain a lengthy connection with the address to which ES:SI is pointing. RemoveConnection allows the MLID to handle commands such as caching source routes more efficiently. However, the MLID should still function properly even if this call is never made.

Assumes

- BX = 8
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to a 6-byte address to break a connection with the address

Returns

- Interrupts: Enabled on exit
- AX = 0

AddPromiscuousSourceFilter

This command allows a protocol stack to request an MLID to filter source addresses when the MLID is in promiscuous mode. If the MLID has no source addresses in its filter table, packets from any source address will be accepted. Otherwise, only those packets from addresses in the source address filter list will be accepted.

The number of slots which the MLID keeps to support promiscuous filtering depends on the implementation of the MLID.

Assumes

- BX = 9
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to a 6-byte address; the MLID should use this as a source address filter in promiscuous mode

Returns

- Interrupts: Enabled on exit
- AX = 0; the source address was added to the MLID's list of source address filters
- AX < 0; an error occurred:

AX = BAD_COMMAND	the MLID does not support filtering promiscuous source addresses
AX = OUT_OF_RESOURCES	the MLID has no more slots to store filter information

◆ *Note*: Promiscuous address filters (see, AddPromiscuousDestination, and ClearPromiscuousFilters later in this chapter) are not affected by changing reception control in and out of promiscuous mode (see ReceptionControl earlier in this chapter).

AddPromiscuousDestinationFilter

This command allows a protocol stack to request an MLID to filter destination addresses when the MLID is in promiscuous mode. If the MLID has no destination addresses in its filter table, packets to any destination address will be accepted. Otherwise, only those packets sent to addresses in the destination address filter list will be accepted.

The number of slots the MLID keeps to support promiscuous filtering is dependent on the implementation of the MLID.

Assumes

- BX = 10
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- ES:SI points to a 6-byte address; the MLID should use this as a destination address filter in promiscuous mode

Returns

- Interrupts: Enabled on exit
- AX = 0; the destination address was added to the MLID's list of destination address filters
- AX < 0; an error occurred:

AX = BAD_COMMAND	the MLID does not support filtering promiscuous destination addresses
AX = OUT_OF_RESOURCES	the MLID has no more slots to store filter information

◆ *Note:* Promiscuous address filters (see AddPromiscuousSource, AddPromiscuousDestination, and ClearPromiscuousFilters later in this chapter) are not affected by changing reception control in and out of promiscuous mode (see ReceptionControl earlier in this chapter).

ClearPromiscuousFilters

This command removes all promiscuous address filters from the tables maintained by the MLID.

Assumes

- BX = 11
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- AX = 0; the node supports promiscuous address filtering
- AX < 0; an error occurred:

AX = BAD_COMMAND the MLID does not support filtering promiscuous destination addresses

This is the only way to remove promiscuous address filters.

◆ *Note*: Promiscuous address filters (see AddPromiscuousSource, AddPromiscuousDestination, and ClearPromiscuousFilters later in this chapter) are not affected by changing reception control in and out of promiscuous mode (see ReceptionControl earlier in this chapter).

DriverPoll

This command is called by the Link Support Layer periodically if Bit 5 is set in the ModeFlags field of the MLID Configuration Table. (This bit indicates that the driver is a polled driver.)

Assumes

- BX = 12
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit

◆ *Note*: This call returns nothing to the caller. Its use is implementation-dependent but is commonly used as an ISR routine for interface cards with no interrupt capability.

Chapter 7 Link Support Commands for MLIDs

This chapter describes support commands in the Link Support Layer that can be issued by the MLID. To call the support commands, place a function code in BX and call the MLID Support Entry Point. (The MLID Support Entry Point is obtained at MLID initialization from the Link Support Layer.) The value of AX will indicate the error. AX will be 0 (and the Z flag set and S flag clear) if the call completed with no error. AX will be less than 0 (and the Z flag clear and S flag set) if the call completed with an error.

The following list presents the calls an MLID can make to the Link Support Layer for support commands. The remainder of this chapter details separately the function of each call.

- GetECB
- ReturnECB
- DeFragmentECB
- ScheduleAESEvent
- CancelAESEvent
- GetIntervalMarker
- DeRegisterMLID
- HoldRcvEvent
- StartCriticalSection
- EndCriticalSection
- GetCriticalSectionStatus
- ServiceEvents
- EnqueueSend
- GetNextSend

- SendComplete
- AddProtocolID ■

GetECB

An MLID calls GetECB whenever the MLID receives a valid packet. The MLID reads the packet into the ECB and sets the ProtoID, BoardNo, and ImmAddr fields to correspond with the incoming packet. The FragCnt field must be set to 1, while the FragPtr1 field must point to the position in the packet data immediately following the link-level envelope. FragLen1 and SendLen will indicate the length of the packet data, not including the size of the envelope. The envelope of the packet will always immediately follow the FragLen1 field of the ECB.

If an ECB is not available, the MLID should discard the incoming packet.

An ECB is always DMA-ready (in other words, on PC-compatibles the ECB will not cross a 64K boundary).

In OS/2, these ECBs are in the Global Descriptor Table (GDT) and addressable by all Ring 0 processes.

Assumes

- BX = 0
- Register preserved: DS, SS, SP, and BP
- Interrupts: Disabled on entry

Returns

- Interrupts: Remain disabled
- AX = 0; an ECB was available:

  AX < 0 ; an error occurred:
  AX = OUT_OF_RESOURCES   no ECBs available
- ES:SI points to the returned ECB, if one was available

ReturnECB

ReturnECB returns an ECB that was allocated through the GetECB command. The ECB is returned to the Link Support Layer's ECB pool for use when needed at a later time.

Most MLIDs do not need to use this command. It is only required if the MLID could not properly receive the incoming packet (after having called GetECB) or if the MLID needs to fragment packets to overcome media protocol constraints (such as limited packet size). Such MLIDs must piece together multiple packets before reporting packet reception. However, the MLID must obtain an ECB (using GetECB) and then hold the ECB when the first fragment arrives. If all of the fragments do not arrive within an amount of time determined by the MLID, the MLID ignores the fragments in its possession and returns the ECB.

Assumes

- BX = 1
- Registers preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry
- ES:SI points to an ECB to be returned to the Link Support Layer's ECB Pool.

Returns

- Interrupts: Remain disabled
- AX = 0
- AX < 0; an error occurred:

  AX = BAD_PARAMETER   the ECB does not belong to the Link Support Layer

DeFragmentECB

This command allows an MLID to defragment an ECB quickly . The resulting ECB will be a copy of the original, except it will have only one fragment. The data from the original fragments will be placed at the specified offset from the FragLen1 field in the destination ECB.

If AX contains 0FFFFH on entry to this command, then ES:SI points to a data buffer only and not an ECB. In this case, the header information from the source ECB will not be copied, but defragmented data will be moved to the data buffer at ES:SI. Otherwise, AX contains an offset past the FragLen1 field to begin storing the defragmented data.

Assumes

- BX = 2
- Register preserved: DS, SS, SP and BP, ES and SI
- Interrupts: Unspecified
- CX:DI points to an ECB to defragment
- ES:SI points to an ECB to hold the defragmented copy
- AX contains the number of bytes past the FragLen1 field in the destination ECB to begin storing the defragmented data. (For example, use 0 if the data is to begin immediately after the FragLen1 field at the start of the envelope field.) If this offset is 0FFFFH, then ES:SI holds only a data buffer, and ECB header information will not be copied

Returns

- Interrupts: Returned the same way they were entered and are not enabled inside the routine
- AX = 0
- AX < 0; an error occurred:

AX = BAD_PARAMETER	the ECB to be defragmented contains invalid fields describing the data contents

◆ *Note*: Because this routine takes a long time to complete, interrupts should be left on if possible.

ScheduleAESEvent

This command schedules a driver-defined event to occur at the end of a specified time interval. An ECB must be supplied for the AES timing system. When the timeout expires, the Status field is set to 0 and the ESR is called. ES:SI will point to the ECB. An ECB which is already in use may not be passed again to this command. To reset the timer of an AES Event, use CancelAESEvent, then issue a new ScheduleAESEvent call.

When using OS/2, the ECB used for this call must be in the GDT . This call requires only the first four fields of an ECB (FLink, BLink, Status, and ESR).

Assumes

- BX = 3
- Registers preserved: DS, SS, SP, BP, ES, and SI
- Interrupts: Unspecified
- ES:SI points to an ECB filled in with the BLink field containing an unsigned 32-bit number for the number of milliseconds that can elapse before the Status field is set to 0 and the ESR Routine is called. The ESR field of the ECB *must* be valid

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- AX = 0

CancelAESEvent

This command cancels the AES event to which ES:SI is pointing. The Status field of the ECB will be set to the CANCELLED error code. The ESR Routine will *not* be called.

Assumes

- BX = 4
- Registers preserved: DS, SS, SP, BP, ES, and SI

- Interrupts: Unspecified
- ES:SI points to the AES ECB to cancel

Returns

- Interrupts: Are returned the same way as they were entered and are not enabled inside the routine
- AX = 0; the cancel command was completed.
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	the ECB was not found in the AES Event queue

GetIntervalMarker

This command returns a timing marker in milliseconds; the marker can be used, for example, for timing retry events. The value of this marker has no relation to any real-world, absolute time. When time marker values are compared with each other, the difference is elapsed time in milliseconds.

Assumes

- BX = 5
- Registers preserved: all except DX:AX
- Interrupts: unspecified

Returns

- Interrupts: are returned the same way they were entered and are not enabled inside the routine
- DX:AX returns the current interval time in milliseconds

DeRegisterMLID

This command allows the driver to tell the Link Support Layer that the board whose number is in AX will no longer be available to the system.

The Link Support Layer will call all protocol stacks that are bound to this driver using the MLIDDeRegistered Protocol Control command to notify them of the deregistration.

Assumes

- BX = 6
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry

- AX contains the board number being deregistered

Returns

- Interrupts: Remain enabled
- AX = 0; the Link Support Layer successfully deregistered the MLID
- AX < 0 ; an error occurred:

AX = BAD_PARAMETER	the Link Support Layer does not have an MLID registered as the board number passed in AX

◆ *Note* The driver should flush its send queue before making this call as described in the section on the MLIDShutdown command in Chapter 6.

HoldRcvEvent

HoldRcvEvent must be called every time a valid packet is received into an ECB. A pointer to this ECB is passed in registers ES:SI. All required fields in the ECB must have been set as indicated in the Assumes section.

The ECB is placed in a temporary holding queue. The ServiceEvents routine (described later) calls the appropriate protocol stack with the incoming packet. The stack must then return the ECB to the Link Support Layer's ECB pool. ServiceEvents is called near the end of the driver's ISR. (Refer to Chapter 4, MLID Operations, for more information.)

Assumes

- BX = 7
- Registers preserved: DS, SS, SP, BP, ES, and SI
- Interrupts: Disabled on entry
- ES:SI contains a pointer to an ECB associated with a completed receive event
- The Status, ImmAddr, ProtoID, and BoardNo fields are set appropriately for the completed packet reception. In addition, the FragCnt field is set to 1, the FragPtr1 field points to the start of packet data immediately following the link-level envelope, and the FragLen1 and SendLen fields contain the size of the packet, not including the size of the envelope. The envelope of the packet will be located immediately following the FragLen1 field, and the packet data will immediately follow the envelope

Returns

- Interrupts: Remain disabled
- AX = 0

StartCriticalSection

StartCriticalSection and EndCriticalSection are two support commands that prevent the Link Support Layer from processing events while the driver is executing critical sections of code. Event processing is delayed until EndCriticalSection is called.

These two support commands bracket any areas of the MLID code that execute with interrupts enabled and that must run to completion for the MLID to continue running smoothly. Also, by definition, the EndCriticalSection command must not be called until the driver is in a state in which it would not be affected by having either AES or ServiceEvents routines executed. Control might not be returned to the driver for some time.

The StartCriticalSection and EndCriticalSection calls should be made so that in the course of execution they are properly balanced. Any imbalances may result in the workstation or server locking up. A Start/End pair of calls may be nested within the domain of another Start/End pair. In some cases, this will result when a critical code section is called from the higher levels of software or called from another critical code section.

In MS-DOS, standard pop-up applications will be disabled while inside the critical section.

Assumes

- BX = 8
- Registers preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry
- AX is the board number of the MLID making the call

Returns

- Interrupts: Remain disabled
- AX = 0
- BX = total number of outstanding calls to StartCriticalSection
- CX = total number of outstanding calls to StartCriticalSection for the requested board number

EndCriticalSection

EndCriticalSection marks the point at which Link Support Layer events and MS-DOS pop-ups can resume. For more information on EndCriticalSection, refer to the description of the previous call, StartCriticalSection.

Assumes

- BX = 9
- Registers preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry

- AX is the board number of the MLID making the call

Returns

- Interrupts: Remain disabled, but they may have been enabled while the EndCriticalSection function executed.
- AX = 0
- BX = total number of outstanding calls to StartCriticalSection
- CX = total number of outstanding calls to StartCriticalSection for the requested board number

GetCriticalSectionStatus

This command reports the current critical section status. For more information, refer to the description of the StartCriticalSection command found earlier in this chapter.

Assumes

- BX = 10
- Registers preserved: DS, SS, SP and BP
- Interrupts: Unspecified
- AX = board number

Returns

- Interrupts: Return unchanged
- AX = 0
- BX = total number of outstanding calls to StartCriticalSection
- CX = total number of outstanding calls to StartCriticalSection for the requested board number

ServiceEvents

ServiceEvents is invoked to complete the processing of network events that have been queued by the HoldRcvEvent command and to process pending AES timer events. ServiceEvents processes each ECB in the holding queue. The ECBs in the queue are processed in the same order that they were added to the list, that is according to the "first in, first out" (FIFO) scheme. ServiceEvents also processes AES-timer ECBs.

The driver ISR should call ServiceEvents immediately before the registers are restored. All hardware processing should have been completed by the driver, and the ISR must be ready to accept a new interrupt.

The ServiceEvents routine will route all received packets to the correct protocol stack and fill in the StackID field for them. The stack is responsible for making the ReturnECB call to release the ECB. (See the description of the ReturnECB call earlier in this chapter for more information.)

The Receive Entry Points, Default Receiver Entry Points, and PreScan Entry Points of the protocol stacks will be called with interrupts disabled, and the program stack will be that of the interrupted process (or the interrupt routine, if the ISR swaps stacks). The protocol stack should turn interrupts on as soon as possible. If the protocol stack is running under an operating system that supports program stack swapping at interrupt time, the protocol stack should swap to its own internal program stack when it processes the received packets.

Assumes

- BX = 11
- Registers preserved: DS, SS, SP and BP
- Interrupts: unspecified
- The system may spend a large amount of time in this routine

Returns

- Interrupts: Return disabled but interrupts will have been enabled during the course of processing
- AX = 0

EnqueueSend

This command allows the driver to place the Send ECB in the Link Support Layer's send queue. The Send ECB remains in the queue until the driver requests it by calling GetNextSend. The ECB may be copied by the Link Support Layer under some operating systems. SendComplete should not be called for the ECB that was sent to the Link Support Layer through EnqueueSend (see the note in GetNextSend, next.)

Assumes

- BX = 12
- Registers preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry
- ES:SI points to the Send ECB to queue

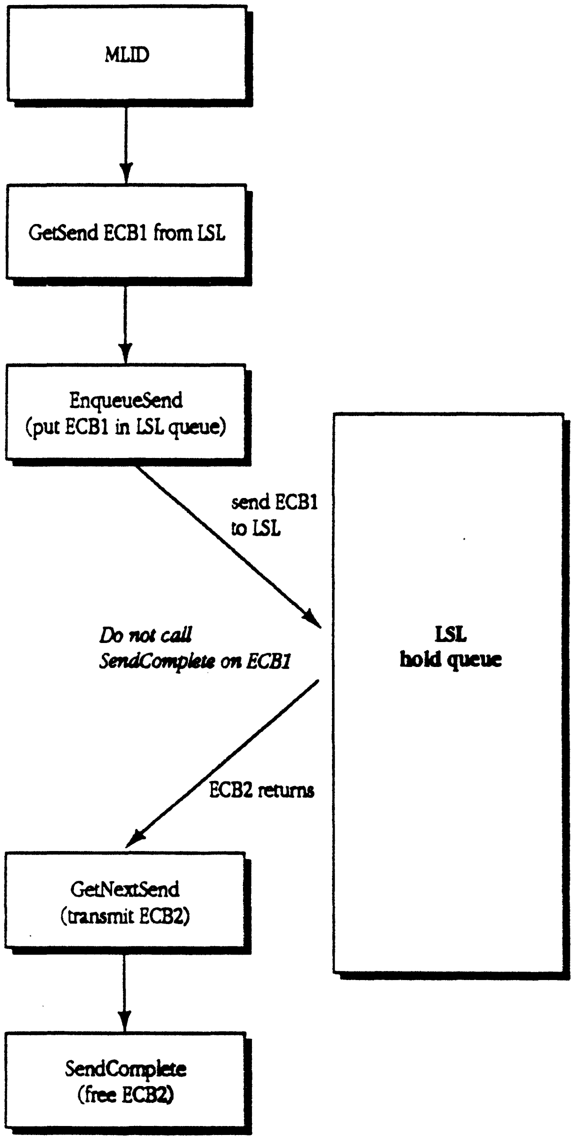
Returns

- Interrupts: Remain disabled

- AX = 0; the ECB was successfully queued
- AX < 0; an error occurred:

AX = OUT_OF_RESOURCES	the ECB needs to be copied and there was not a free ECB available to create the copy

Transmitting an ECB with the LSL hold queue



GetNextSend

This command determines if there are any Send ECBs on the Link Support Layer's send queue. If there are, the address of the first ECB is returned in ES:SI. If not, a NO_MORE_ITEMS error code is returned. (For more information, refer to the previous description of EnqueueSend.)

Assumes

- BX = 13
- Registers preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry
- AX contains the board number making the request

Returns

- Interrupts: Remain disabled
- AX = 0. ES:SI will return an ECB queued up with the EnqueueSend command
- AX < 0 if an error occurred:

AX = NO_MORE_ITEMS	no ECBs are queued

◆ *Note:* The ECB returned by this command may not be the same ECB that was queued using the EnqueueSend command, but may be a copy that has been defragmented. The driver must call SendComplete when it is through using this ECB.

SendComplete

SendComplete must be called every time a packet is transmitted from an ECB. A pointer to this ECB is passed in registers ES:SI.

This routine must be called any time the driver is finished using a Send ECB. The Link Support Layer calls the ESR of the SendECB with interrupts disabled and ES:SI pointing to the ECB. The ESR routine must not enable interrupts; it must execute quickly since it is called at interrupt time. Always call SendComplete when you are done with an ECB returned by GetNextSend (even if it is the same ECB given to an EnqueueSend), but never call SendComplete when queuing a Send with EnqueueSend. Normally, a driver is finished using a SendECB after it sends the data to the interface card, or after calling GetNextSend in the driver's ISR routine and sending the data to the interface card. However, if the driver used the EnqueueSend command (described earlier) to transmit the ECB at a later time, SendComplete should *not* be called for the ECB given to the EnqueueSend command.

Assumes

- BX = 14

- Registers preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry
- ES:SI contains a pointer to an ECB associated with a completed Send event.

Returns

- Interrupts: Remain disabled
- AX = 0; no errors are possible.

AddProtocolID

This command allows the driver to tell the Link Support Layer the names and protocol IDs of the protocol stacks the driver can support.

Assumes

- BX = 15
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry
- ES:SI points to the 6-byte protocol ID being added
- CX:DI points to a string (no more than 15 characters long) containing the name of the protocol stack for this protocol ID. The string *must* have a leading length byte and a trailing zero byte.
- AX contains the media ID for which the new protocol ID is being added

Returns

- Interrupts: Enabled on exit
- AX = 0; the Link Support Layer successfully added the new protocol ID
- AX < 0; an error occurred:

AX = OUT_OF_RESOURCES	The Link Support Layer has no resources to register another protocol ID
AX = DUPLICATE_ENTRY	There is already a protocol ID registered for the given media/stack combination
AX = BAD_PARAMETER	The name of the parameter is illegal (undefined) and the length field of this parameter should be less than or equal to 15

◆ *Note:* This call should only be made for PID info in the NET.CFG file because of a limited number of protocol stack slots.

Part II Writing Protocol Stacks for the MPI

Chapter 8 **Protocol Stack Operations**

This chapter briefly describes the operation of a protocol stack. The stack receives packets from the Link Support Layer and then processes these received packets. The protocol stack also creates outgoing packets. The stack delivers the outgoing packets to the Link Support Layer. From the Link Support Layer, the packets are delivered to the MLID that was requested by the protocol stack.

The processing of these packets by the protocol stack allows higher-level services (such as registration and lookup of entity names and transaction processing) to exist. Each stack maintains its own set of higher-level services so the availability of a particular service will vary from stack to stack. ■

The stack normally consists of the following four handlers:

- The Protocol Stack Control Entry Point handler
  This handler allows applications and stacks to obtain valuable information about the stack. The entry point is detailed further in Chapter 10, Protocol Stack Control Procedures.
- The Application Entry Point handler
  This handler services requests by applications.
- The Transmit Packet handler
  This handler is responsible for creating an ECB, filling in its fields, and passing the ECB to the Link Support Layer for transmission.
- The SendComplete ISR handler
  This handler is called whenever a Send ECB has been processed by the MLID. The handler allows the stack to do any required processing after an ECB has been transmitted

  One or more of the following three handlers must also be included in the stack:
- The Receive Entry Point handler
  This handler processes received packets delivered by the Link Support Layer. Stacks receive packets if the BindStack call was used to bind with an MLID.
- The Default Receiver Entry Point handler.
  This is another handler for receiving packets delivered by the Link Support Layer. The Default Receiver Entry Point receives all incoming packets from the MLID that can not be routed to other stacks. For example, the PID of the packet is not registered. A stack receives packets if it was registered as the default receiver by using the RegisterDefaultStack call. This handler could be the same as the Receiver Entry Point handler.
- The PreScan Entry Point handler
  This entry point allows a special purpose stack to filter or preview incoming packets from an MLID before they are routed by the Link Support Layer.

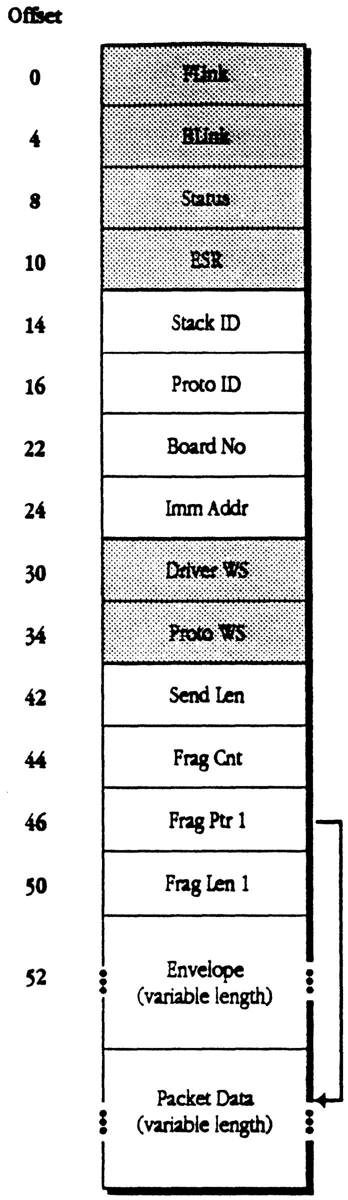
The process of registering and binding with MLIDs is discussed in Chapter 9, *Protocol Stack Initialization.* This chapter provides detailed descriptions of the following:

- The Receive Entry Point
- The Default Receiver Entry Point
- The PreScan Entry Point
- The Transmit Packet handler

The Receive Entry Point

The Link Support Layer passes the Receive Entry Point a pointer to an ECB in ES:SI with the following fields filled in:

A Receive ECB



StackID contains the stack ID of the protocol stack for which this packet is destined. This value need only concern stacks that are attempting to handle two different protocol IDs.

ProtoID contains the protocol ID of the incoming packet. Most stacks do not need this value.

BoardNo contains the board number of the MLID that received this packet.

ImmAddr contains the physical address of the source node of this packet.

FragCnt always contains a 1.

SendLen contains the number of bytes in the packet (not counting the envelope).

FragPtr1 contains a pointer to the start of the packet's data (immediately following the envelope).

FragLen1 contains the number of bytes in the packet's data (not counting the envelope).

Envelope immediately follows FragLen1 and has a variable length, depending on the media.

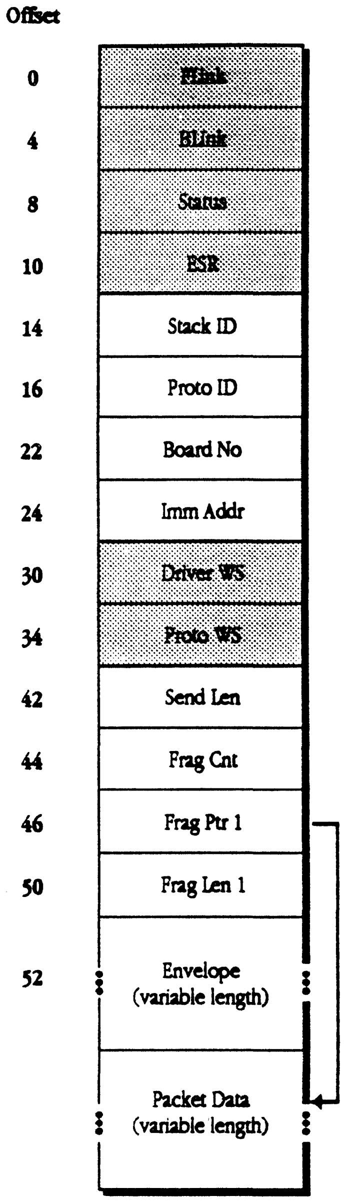
The Receive Entry Point may destroy all registers except SS, SP, BP, and DS.

The protocol stack is responsible for making the ReturnECB call; this returns the ECB to the Link Support Layer. The ECB should be returned after the protocol stack has finished processing the received packet.

The Default Receiver Entry Point

The Link Support Layer passes the Default Receiver Entry Point Handler a pointer to an ECB in ES:SI with the following fields filled in:

A Receive ECB



StackID contains 0FFFFH.

ProtoID contains the protocol ID of the incoming packet.

BoardNo contains the board number of the MLID that received this packet.

ImmAddr contains the physical address of the source node of this packet.

SendLen contains the number of bytes in the packet (not counting the envelope).

FragCnt contains a 1.

FragPtr1 contain a pointer to the start of the packet's data (immediately following the envelope).

FragLen1 contain the number of bytes in the packet's data (not counting the envelope)

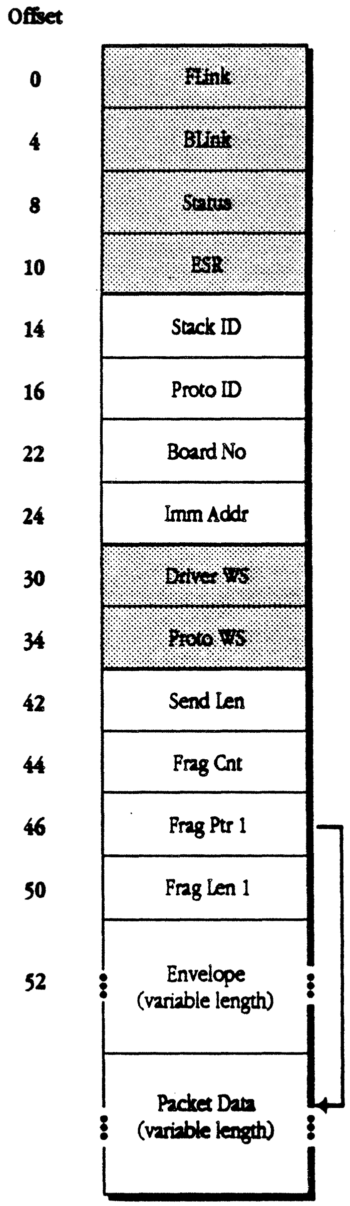
Envelope immediately follows Fraglen1 and has a variable length, depending on the media.

The Default Receiver Entry point handler may destroy all registers except SS, SP, BP, and DS. The protocol stack is responsible for making the ReturnECB call; this returns the ECB to the Link Support Layer after it has finished processing the received packet.

The PreScan Entry Point

The Link Support Layer passes the PreScan Entry Point Handler a pointer to an ECB in ES:SI with the following fields filled in:

A Receive ECB



Offset	
0	FLink
4	BLink
8	Status
10	ESR
14	Stack ID
16	Proto ID
22	Board No
24	Imm Addr
30	Driver WS
34	Proto WS
42	Send Len
44	Frag Cnt
46	Frag Ptr 1
50	Frag Len 1
52	Envelope (variable length)
	Packet Data (variable length)

ProtoID contains the protocol ID of the incoming packet

BoardNo contains the board number of the MLID that received this packet. A protocol stack may want to reject a packet depending on the origin of its board number

ImmAddr contains the physical address of the source node of this packet

SendLen contains the number of bytes in the packet (not counting the envelope)

FragCnt always contains a 1.

FragPtr1 contains a pointer to the start of the packet's data (immediately following the envelope).

FragLen1 contains the number of bytes in the packet's data (not counting the envelope).

Envelope immediately follows FRAG_LEN1 and has a variable length depending on the media.

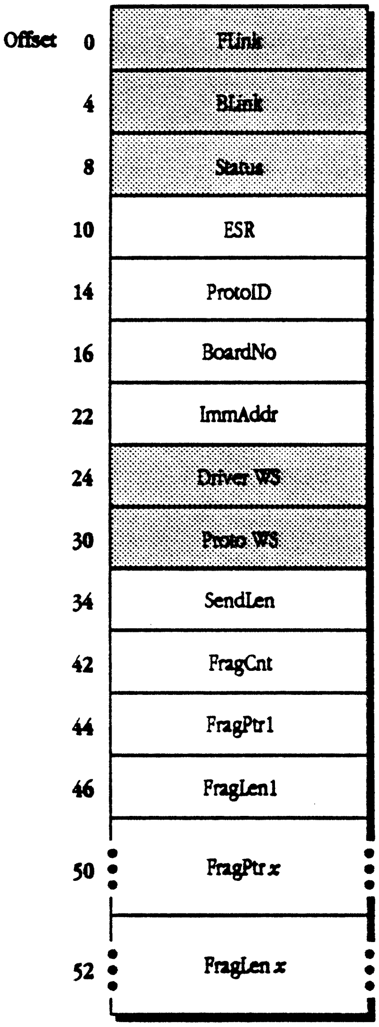
The PreScan Entry Point Handler may destroy all registers except SS, SP, BP, and DS. The stack should return AX = 0 if the Link Support Layer should not route the packet. The stack should return AX = 1 and ES:SI pointing to the receive ECB if the Link Support Layer should route the receive ECB. If the stack returns AX = 0, the stack is responsible for making the ReturnECB command on the received ECB.

The Transmit Packet Handler

To send a packet on the network, the protocol stack must create an ECB and provide the following information in the ECB:

Transmit Packet Handler

Transmit



Offset	Transmit
0	FLink
4	BLink
8	Status
10	ESR
14	ProtoID
16	BoardNo
22	ImmAddr
24	Driver WS
30	Proto WS
34	SendLen
42	FragCnt
44	FragPtr1
46	FragLen1
50	FragPtr *x*
52	FragLen *x*

StackID contains the stack ID of the stack sending the packet or 0FFFFH if the stack intends to send a "raw" packet.

ESR contains the address of an Event Service Routine which will be called when the ECB is freed. This field *must* contain a valid address.

ProtoID contains the protocol ID for the stack ID/board number for which the packet is destined. This field is ignored if a "raw" packet is sent.

BoardNo contains the board number for which the packet is destined.

ImmAddr contains the physical address of the destination node for this packet. This field is ignored if a "raw" packet is sent.

Proto WS may be filled in with any needed information (will not be modified by the MLID or the Link Support Layer).

SendLen contains the complete length of the packet, counting all fragments, but not counting the envelope (unless the feature to send a "raw" packet is used)

FragCnt contains the number of fragments for the packet

FragPtrX contains the pointer to the X'th fragment of the packet

FragLenX contains the number of bytes in the X'th fragment of the packet

After filling in the ECB, the packet is transmitted using the SendPacket call. The transmit command is asynchronous, and the stack may not reuse the ECB until the ESR is called. The ESR is called even if an error occurs. A stack must not depend on receiving an error if the transmit fails. Some MLIDs call SendComplete immediately after the data from the ECB has been transferred to the memory on the interface card.

The ESR may destroy all registers except SS, SP, BP, and DS, but may not enable interrupts.

If the transmit command waits for the ESR to be called before continuing (for the caller this means turning the transmit into a synchronous command), the transmit handler should make the RelinquishControl call while waiting. However, if the stack is operating under a multitasking operating system such as OS/2, the stack should yield to the operating system instead, normally by waiting on a semaphore.

If the StackID field is set to 0FFFFH, then a "raw" send is performed by the MLID. Normally, the MLID encapsulates the data given by the ECB in a link-level envelope before transmitting the packet. For a raw transmit, the stack is responsible for building the entire packet including media-specific headers. In this case the media envelope must be contained entirely in the first fragment.

◆ *Note:* The ESR may be called before control is returned from the SendPacket call.

Chapter 9 **Protocol Stack Initialization**

Protocol stack initialization occurs when the stack loads itself in the computer's system. This initialization process, described in this chapter, must take place before the stack can send and receive packets. ■

Stack initialization stages

The initialization process of a stack occurs in the following stages:

1. The protocol stack first establishes a connection with the Link Support Layer. When using OS/2, installation is accomplished when the stack sends an IOCTL command to the LINKSUP$ device using the general IOCTL command (DosDevIOCtl) with a function category of 0A1H and a function code of 2. However, under MS-DOS, the LSL is a TSR program, and the Link Support Layer's Initialization Interface Entry Point is found using the INT 2FH multiplexing address. The exact procedure for finding this entry point is described in Appendix I.

2. The stack uses the following information to call the Link Support Layer's Initialization Entry Point. Using this call, the stack can obtain further information about the Link Support Layer.

   BX = 2; protocol stack initialization function code

   ES:SI; address of 8 bytes in memory into which the Link Support Layer fills the addresses of the following two entry points:

Offset	Bytes	Description
0	4	Ring 0 address of the Protocol Stack Support Entry Point of the Link Support Layer.
4	4	Ring 0 address of the General Services Entry Point of the Link Support Layer.

   These two Link Support Layer entry points are described in more detail in later chapters.

3. The stack reads the NET.CFG file. (See Appendix H for the format of the NET.CFG file).

4. A stack must complete one or both of the following registration operations before it can receive packets from an MLID:

   □ Register with the Link Support Layer by name and bind with an MLID

   □ Register as the default handler for an MLID

   □ Register as a PreScan stack for an MLID

   Each of these registration operations is described in the next section.

◆ *Note:* With OS/2, the eight bytes of entry point information is returned in the IOCTL data buffer.

Registering a protocol stack

A protocol stack must be registered with the Link Support Layer in order to receive packets from an MLID. Registration provides the Link Support Layer with the information it requires to route packets from MLIDs to protocol stacks.

When an MLID receives a packet, the MLID places the board number of the MLID, the protocol ID from the link-level envelope (or 0 if no protocol ID exists) and the packet into a receive ECB and passes the ECB to the Link Support Layer. The Link Support Layer uses the board number and protocol ID to route the packet.

The Link Support Layer first calls the PreScan stack for the MLID if present. If a PreScan stack does not exist or indicates that the Link Support Layer should route the packet, the Link Support Layer searches for any stack that is registered and bound to the MLID. If no suitable stack is found, then the Link Support Layer will call the stack that has registered as the default for the MLID. If no stack has registered as the default, the packet is ignored. The ECB is then returned to the Link Support Layer's ECB pool by the Link Support Layer.

A stack can receive a packet in three ways. The stack can bind with an MLID to receive packets that have a particular protocol ID. The stack can also be registered to receive all packets from an MLID if no other stack claims them, or the stack can be registered as a PreScan stack. As a PreScan stack, it can receive all packets. A stack can use all three methods of registering to receive packets from the same MLID. A stack can also be bound with any number of MLIDs.

To register by binding with an MLID

A stack can bind with an MLID to receive specific packets (those with a specific protocol) by making at least two calls. The stack must first make the RegisterStack call to obtain its StackID. The RegisterStack call takes the following as parameters:

- the name of the protocol stack
- a pointer to a table containing the following:

  a pointer to the Receive Entry Point

  a pointer to the Protocol Stack Control Entry Point

ES:SI points to a table containing the previously mentioned parameters, as shown in the RegisterStack call described later in this chapter.

Using the StackID obtained from the RegisterStack call and the board number of the desired MLID, the protocol stack makes the BindStack call to complete the binding process.

To register as the default stack

A protocol stack can become the default receiver for an MLID by making a single RegisterDefaultStack call. See Chapter 11, *Link Support Procedures for Protocol Stacks*, for a full description of the call. Only one protocol stack can be the default receiver of an MLID. The stack is implicitly bound to the MLID by making this call.

Register as the PreScan stack

A stack can receive all packets from an MLID by making a single RegisterPreScanStack call. See Chapter 11, *Link Support Procedures for Protocol Stacks*, for a full description of the call. Only one stack can be the PreScanStack for an MLID. The stack is implicitly bound to the MLID by making this call.

In order to make the registration calls, the stack must determine the board number of the desired MLID. Board numbers are assigned dynamically by the Link Support Layer as each MLID registers. A stack should not depend on an MLID having any particular board number; the board number should be determined every time the stack initializes.

Finding an MLID by name

A protocol stack can find an MLID's board number from the name of the MLID. The name of the MLID can be read from the NET.CFG file when the stack initializes, or the name of the MLID can be hard coded. The following algorithm (in pseudo-C) can be used for finding an MLID's board number from its name.

```
/*
************************************************************************
**  GetBoardNofromName
**
** Inputs:
**      name   Pointer to name of MLID being searched for.
**      The first byte of the string must have its length,
**      and the string must be null-terminated.
** Outputs:
**      BoardNo or -1 if none found.
************************************************************************
*/
int GetBoardNofromName(char  *name)
{
        int                         board;
        PtrToFunction               MLIDEntry;    /* MLID Control Entry Point */
        MLIDConfigurationTable      *tbl;         /* Ptr to MLID's config tbl */
        int                         status;       /* Status of calls to LSL */
```

```
for (board = 0;   ; board++)

{

        /*

        ** Call Link Support Layer to get the Control Entry

        ** Pcint for MLID

        **

        */

        status = GetMLIDControlEntry(board, &MLIDEntry);

        if (status == NO_MORE_ITEMS)      /* Didn't find MLID */

                return -1;

        if (status == ITEM_NOT_FOUND)     /* No MLID with this ID */
                        continue;




        /*

        ** Call MLID with function code 0 to get its

        ** Configuration table.

        **

        ** The syntax below will not work exactly under most

        ** C's since they pass the parameters on the stack.

        ** Pseudo-C, remember?

        ** (An assembly language glue routine will be required.)

        */

        status = MLIDEntry(0, &ConfigTbl);

        if (status)                       /* Should never happen... */

                continue;

        /*

        ** Compare the name of the MLID in its configuration
```

```
                ** table with the desired name
                */
                if (stricmp(name, ConfigTbl->ShortName) == 0)
                        return board;               /* Found our MLID */
        } /* for */
}
```

Chapter 10 **Protocol Stack Control Commands**

This chapter describes commands that must be provided by a protocol stack in order for it to support the MPI interface. ■

To call a protocol stack control command, place a function code into BX and call the Protocol Stack Control Entry Point. The address of this entry point is obtained by making a GetProtocolControlEntry command call to the Link Support Layer for the desired protocol stack. The return value in AX will always be generated so that the Z and S flags are set correctly. AX will be 0 (and the Z flag set and S flag clear) if the call is completed with no error. AX will be less than 0 (and the Z flag clear and S flag set) if the call completed with an error. The value of AX will indicate the error. The following support commands are available through this interface:

- GetProtocolStackConfiguration
- GetProtocolStackStatistics
- BindToMLID
- UnbindFromMLID
- MLIDDeRegistered

GetProtocolStackConfiguration

This command allows a protocol stack to read the name and version information of another protocol stack.

Assumes

- BX = 0
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- ES:SI returns a pointer to the Protocol Stack Configuration Table (see Appendix B for a description of this table)
- AX = 0, no errors are possible

GetProtocolStackStatistics

This command returns a pointer to the Protocol Stack Statistics Table. The table describes statistics of the protocol stack.

Assumes

- BX = 1

- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry

Returns

- Interrupts: Enabled on exit
- ES:SI points to a statistics table whose format is described in Appendix E
- AX = 0, no errors are possible

BindToMLID

This command provides a consistent method to instruct a protocol stack to bind with an MLID.

The protocol stack is expected to issue the BindStack call to the Link Support Layer as well as perform any other maintenance commands required to bind to an MLID.

Assumes

- BX = 2
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- CX contains the board number to which the protocol stack should bind
- ES:SI points to a parameter string that is dependent on the implementation

Returns

- Interrupts: Enabled on exit
- AX = 0; the bind completed successfully (other error codes are dependent on the implementation)

UnbindFromMLID

This command provides a consistent method to instruct a protocol stack to unbind from an MLID.

Assumes

- BX = 3
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Enabled on entry
- CX contains the board number from which the protocol stack should unbind.
- ES:SI points to a parameter string that is dependent on the implementation.

Returns

- Interrupts: Enabled on exit
- AX = 0; the unbind call is completed successfully (other error codes are dependent on the implementation)

MLIDDeRegistered

This command allows the Link Support Layer to inform all protocol stacks bound to an MLID that the MLID has deregistered. As a result, the MLID will no longer be available. This call is used strictly to inform stacks. The stack may use the information any way it chooses and may even ignore it.

Assumes

- BX = 4
- Registers preserved: DS, SS, SP and BP
- Interrupts: Enabled on entry
- CX contains the board number which has deregistered from the Link Support Layer.

Returns

- Interrupts: Enabled on exit
- AX has no return value for this call

Chapter 11 Link Support Commands for Protocol Stacks

This chapter describes support commands found in the Link Support Layer that can be called by the protocol stacks. ■

To call the Link Support Layer's protocol support commands, place a function code in BX and call the Protocol Stack Support Entry Point. AX will be 0 (and the Z flag set and S flag clear) if the call completed with no error. AX will be less than 0 (and the Z flag clear and S flag set) if the call completed with an error. The value of AX will indicate the error.

The following support commands are available to protocol stacks:

- GetECB
- ReturnECB
- DeFragmentECB
- ScheduleAESEvent
- CancelAESEvent
- GetIntervalMarker
- RegisterStack
- DeRegisterStack
- RegisterDefaultStack
- DeRegisterDefaultStack
- RegisterPreScanStack
- DeRegisterPreScanStack
- SendPacket
- HoldPacket
- GetHeldPacket
- ScanPacket
- GetStackIDfromName
- GetPIDfromStackIDBoard
- GetMLIDControlEntry
- GetProtocolControlEntry
- GetLinkSupportStatistics
- BindStack
- UnbindStack
- AddProtocolID
- RelinquishControl ■

GetECB

This command allows a protocol stack to obtain and use an ECB. The ECB is normally used when a protocol stack has to *simulate* a received packet. Because MLIDs filter incoming packets sent by other MLIDs, protocol stacks that need to receive their own transmissions can simulate a received packet by getting an ECB from this command, copying the packet to be received into it, and calling HoldPacket. The protocol stack retrieves the packet later by calling GetNextHeldPacket.

In OS/2, ECBs are in the GDT and addressable by all Ring 0 processes. In MS-DOS and OS/2, the packet does not cross a 64K DMA boundary.

Assumes

- BX = 0
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Disabled on entry

Returns

- Interrupts: Remain disabled
- AX = 0; an ECB was available
- AX < 0; an error occurred:

  AX = OUT_OF_RESOURCES   no ECBs were available
- ES:SI points to the returned ECB, if one was available

ReturnECB

ReturnECB returns an ECB that was allocated through the GetECB command. The ECB is returned to the Link Support Layer's ECB pool for reuse.

Assumes

- BX = 1
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Disabled on entry
- ES:SI points to an ECB to be returned to the Link Support Layer's ECB pool for reuse

Returns

- Interrupts: Remain disabled
- AX = 0
- AX < 0; an error occurred:

  AX = BAD_PARAMETER   the ECB does not belong to the Link Support Layer's ECB pool

DeFragmentECB

This command allows a protocol stack to defragment an ECB quickly. The resulting ECB will be a copy of the original. The only exception is that there will be only one fragment, and the data from the original fragments will be placed in AX bytes after the FragLen1 field in the destination ECB. (AX refers to the ECB at an offset specified by AX past the FragLen1 field).

If AX contains 0FFFFH on entry to this command, then ES:SI points to only a data buffer, and not an ECB. In this case, the header information from the source ECB will not be copied, but defragmented data will be moved to the data buffer at ES:SI.

Assumes

- BX = 2
- Register preserved: DS, SS, SP and BP, ES and SI
- Interrupts: unspecified
- CX:DI points to an ECB to defragment
- ES:SI points to an ECB to hold the defragmented copy
- AX contains the number of bytes past the FragLen1 field in the destination ECB where the defragmented data should be stored. If this offset is 0FFFFH, then ES:SI only holds a data buffer, and ECB header information will not be copied

Returns

- Interrupts: returned the same way they were entered, and are not enabled inside the routine
- AX = 0
- AX < 0; an error occurred:

AX = BAD_PARAMETER	the ECB to be defragmented contains invalid fields describing the data contents

◆ *Note:* Interrupts should be left on if at all possible.

ScheduleAESEvent

This command schedules a driver-defined event to occur at the end of a specified time interval. An ECB must be supplied for the AES timing system. When the timeout occurs, the Status field is set to 0 and the ESR is called, with ES:SI pointing to the ECB. An ECB which is already in use should not be passed again to this command. To reset the timer of an AES Event, use CancelEvent, then issue a new ScheduleAESEvent call.

Assumes

- BX = 3
- Registers preserved: DS, SS, SP, BP, ES, and SI
- Interrupts: unspecified
- ES:SI points to an ECB whose Blink field is filled in with an unsigned 32-bit number. This number indicates how many milliseconds should elapse before the Status field is set to 0 and the ESR is called. The ESR field of the ECB must contain a valid ESR pointer
- In OS/2, the ECB is in the GDT

Returns

- Interrupts: returned the same way they were entered, and are not enabled inside the routine
- AX = 0; no errors are possible

CancelEvent

This command cancels the ECB to which ES:SI was pointing. The Status field of the ECB will be set to the CANCELLED error code. The ESR will not be called. This command will cancel an outstanding AES ECB or SendPacket ECB.

Assumes

- BX = 4
- Register preserved: DS, SS, SP, BP, ES, and SI
- Interrupts: Disabled on entry
- ES:SI contains a pointer to an ECB to be cancelled

Returns

- Interrupts: remain Disabled
- AX = 0; the cancel was completed successfully
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	the ECB could not be cancelled

GetIntervalMarker

This command returns a timing marker in milliseconds. The timing marker can be used, for example, for timing retry events. The value of this marker has no relation to any real-world absolute time. However, when time marker values are compared with each other, the difference between them is elapsed time in milliseconds.

Assumes

- BX = 5
- Registers preserved: all except DX, AX, BX
- Interrupts: unspecified

Returns

- Interrupts: are returned the same way they were entered and are not enabled inside the routine
- DX:AX returns the current interval time in milliseconds

◆ *Note* AX does not return an error code

RegisterStack

A protocol stack can transmit packets and communicate with MLIDs even if it has not registered. Either the RegisterStack, RegisterDefaultStack, or RegisterPreScanStack call must be made for the protocol stack to receive incoming packets from the Link Support Layer.

In addition to a RegisterStack call, the protocol stack must issue BindStack calls for those MLIDs from which the stack wants to receive packets.

Assumes

- BX = 6
- Registers preserved: DS, SS, SP, and BP
- Interrupts: unspecified
- ES:SI points to a table with the following information:

Offset	Bytes	Description
0	4	Pointer to the name of the protocol stack. This pointer needs to be valid only at the time the call is made
4	4	Ring 0 Receive Entry Point for the protocol stack. All incoming packets will be dispatched to this address for processing
8	4	Ring 0 Address of the Protocol Stack Control Entry Point

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- BX = Stack ID
- AX = 0; no error occurred
- AX < 0; an error occurred:

AX = OUT_OF_RESOURCES	there are too many protocol stacks already registered
AX = DUPLICATE_ENTRY	a protocol stack with that name is already registered
AX = BAD_PARAMETER.	length of the protocol stack name is 0 or greater than 15

DeRegisterStack

This command removes a protocol stack from the Link Support Layer's list of protocol stacks. After making this call, a protocol stack will not receive any more incoming packets (unless the protocol stack has an outstanding RegisterDefaultStack or RegisterPreScanStack), and must make the RegisterStack call again to start receiving incoming packets.

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

Assumes

- BX = 7
- Register preserved: DS, SS, SP and BP
- Interrupts: disabled on entry
- AX contains the stack ID that the protocol stack is de-registering.

Returns

- Interrupts: remain disabled
- AX = 0; the protocol stack was deregistered
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	no protocol stack is registered with that stack ID

RegisterDefaultStack

This call can be made for a protocol stack which needs to accept all incoming packets that are not bound for other protocol stacks.

This call implicitly binds the protocol stack to the MLID whose board number is specified in AX. No call to BindStack is possible (there is no stack ID associated with a default stack).

The RegisterDefaultStack command allows a protocol stack that recognizes the link-level envelope to receive packets unwanted by other protocol stacks.

Assumes

- BX = 8
- Register preserved: DS, SS, SP and BP
- Interrupts: Unspecified
- AX contains the board number from which the protocol stack will receive all packets not specifically sent to any other protocol stack
- ES:SI points to a table with the following information:

Offset	Bytes	Description
0	4	Ring 0 Default Receiver Entry Point for the protocol stack. All incoming packets will be dispatched to this address for processing
4	4	Ring 0 Address of the Protocol Stack Control Entry Point

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- AX = 0; no error occurred
- AX < 0; an error occurred:

AX = DUPLICATE_ENTRY	There is already a default stack registered for the desired board number
AX = BAD_PARAMETER	The MLID corresponding to the requested board number does not exist

◆ *Note:* RegisterStack, RegisterDefaultStack and, RegisterPreScanStack are separate calls and handled independently by the Link Support Layer. Both calls can be used by a protocol stack, depending on the particular need.

DeRegisterDefaultStack

This command removes the protocol stack associated with a specific MLID from the Link Support Layer's list of default stacks. After making this call, a protocol stack will not receive incoming packets from the specified MLID unless the protocol stack still has an outstanding RegisterStack call.

Assumes

- BX = 9
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Disabled on entry
- AX contains the board number for the default protocol stack being deregistered

Returns

- Interrupts: Remain disabled
- AX = 0; the protocol stack was deregistered
- AX < 0; an error occurred:

AX = BAD_PARAMETER.	the MLID corresponding to the requested board number does not exist
AX=ITEM_NOT_PRESENT	there is no default stack registered for this MLID

RegisterPreScanStack

This call can be made by a protocol stack which needs to filter or examine all incoming packets before they are routed to other protocol stacks.

This call implicitly binds the protocol stack to the MLID whose board number is specified in AX. No call to BindStack is possible (there is no stack ID associated with a PreScanStack).

The RegisterPreScanStack command allows a protocol stack to determine whether a packet should be routed by the Link Support Layer or discarded.

Assumes

- BX = 10
- Register preserved: DS, SS, SP and BP
- Interrupts: Unspecified
- AX contains the board number from which the protocol stack intends to receive all packets
- ES:SI points to a table with the following information:

Offset	Bytes	Description
0	4	Ring 0 PreScanStack Entry Point for the protocol stack. All incoming packets will be dispatched to this address for processing. This routine will return AX = 1 to allow the Link Support Layer to route the incoming packet, or AX = 0 if the Link Support Layer should not route the incoming packet. ES:SI must remain unchanged to permit the Link Support Layer to route the packet
4	4	Ring 0 Address of the Protocol Stack Control Entry Point

Returns

- Interrupts: Returned the same way they were entered, and are not enabled inside the routine
- AX = 0; no error occurred
- AX < 0; an error occurred:

AX = DUPLICATE_ENTRY	there is already a PreScanStack registered for the desired board number
AX = BAD_PARAMETER.	the MLID corresponding to the requested board number does not exist

RegisterStack, RegisterDefaultStack, and RegisterPreScanStack are separate calls, and they are handled independently by the Link Support Layer. Both calls can be used by a protocol stack, depending on the particular need.

◆ *Note:* PreScan stacks are intended to be used to implement a "security-monitoring stack" that keeps sensitive packets from being routed. They can also be used to preview packets before they are routed to other protocol stacks in the system.

DeRegisterPreScanStack

This command removes the protocol stack name associated with a specific MLID from the Link Support Layer's list of default stacks. After making this call, a protocol stack will not receive incoming packets from the specified MLID unless the protocol stack still has an outstanding RegisterStack or RegisterDefaultStack call.

Assumes

- BX = 11
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Disabled on entry
- AX contains the board number for the default stack being deregistered

Returns

- Interrupts: remain disabled
- AX = 0; the protocol stack was deregistered
- AX < 0; an error occurred:

AX = BAD_PARAMETER.	the MLID corresponding to the requested board number does not exist

AX = ITEM_NOT_PRESENT — there is no PreScanStack registered on the requested board number

SendPacket

This command sends a packet to one of the registered MLIDs. The ESR field of the ECB must be filled in with the address of a routine to call when the send is complete. Until the ESR is called, the ECB and all its data areas belong to the Link Support Layer and ***must not*** be modified.

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

The ESR of the SendPacket ECB will be called with ES:SI pointing to the ECB that was sent. This call will be made at interrupt time with interrupts disabled. The ESR should not re-enable interrupts, and should complete quickly. The Status field of the ECB will indicate any errors that were detected. The ESR can destroy all registers except DS, SS, SP, and BP.

Assumes

- BX = 12
- Register preserved: DS, SS, SP, and BP
- Interrupts: unspecified
- ES:SI contains a pointer to the Send ECB. The SendECB contains the following information:

ESR. The address of a routine that is called when the ECB is free (after the packet has been transmitted or copied). A pointer to the ECB is passed to this routine in ES:SI; the ECB's Status field contains the result of the send

StackID. The stack ID of the protocol stack sending the packet. If this field is 0FFFFH, then the packet is raw and the first fragment of the send list contains the full header of the packet

ProtoID. The protocol ID that the MLID is to use when encapsulating the data. This field is ignored if "raw packet" is sent

BoardNo. The board number of the MLID sending this packet

ImmAddr. The network address to which this packet is destined, unless the packet is raw, in which case this field is undefined

SendLen. The total length of all fragments sent must be stored here

FragCnt. The number of fragments in the packet to be sent

FragPtrX. A pointer to the data of the X'th fragment

FragLenX. The number of bytes present in the Xth fragment

Returns

- Interrupts: Will return enabled
- AX = 0; there was no error; however, until the ESR is called, the ECB belongs to the MLID
- AX < 0; an error occurred:

AX = NO_SUCH_DRIVER	the BoardNo in the ECB does not exist
AX = BAD_PARAMETER	the SendECB was not completed correctly
AX = OUT_OF_RESOURCES	there were not enough resources to handle the send request
AX = FAIL	the MLID could not send the packet

◆ *Note:* If an error occurred, the error code will be placed in the status field of the ECB and the ESR will be called. Be aware that the ESR can be called before the call to SendPacket returns.

HoldPacket

This command allows a protocol stack to queue an incoming packet for later processing. The GetHeldPacket and ScanPacket commands can be used to find and remove packets from this queue.

Assumes

- BX = 13
- Registers preserved: DS, SS, SP, BP, ES, and SI
- Interrupts: Disabled on entry
- ES:SI contains a pointer to a Receive ECB to be held.

Returns

- Interrupts: Remain disabled
- AX = 0; no errors are possible

❖ *Note:* ECBs in the hold queue may be reused by the Link Support Layer if it runs out of ECBs.

GetHeldPacket

This command allows a protocol stack to remove an ECB from the hold queue; it was placed there with a call to HoldPacket. ES:SI may point to an ECB to remove it from the queue (the ECB is usually obtained from a ScanPacket call). Otherwise, ES:SI can be set to 0:0 to allow a limited search of the queue. For more powerful searches, use the ScanPacket call.

Assumes

- BX = 14
- Register preserved: DS, SS, SP, and BP
- Interrupts: Disabled on entry
- AX = stack ID
- ES:SI = 0:0

  CX = match word

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

  □ The ECB's stack ID matches the value in AX

  □ The first word of the protocol workspace matches the value in CX

◆*Note:* If CX = 0FFFFH, the match on the protocol workspace will be ignored.

- ES:SI <> 0:0

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

Returns

- Interrupts: Remain disabled
- AX = 0; ES:SI will contain a pointer to the desired ECB. The ECB will have been removed from the queue.
- AX < 0; an error occurred:

AX = BAD_PARAMETER	the ECB passed in ES:SI did not belong to the protocol stack (the StackID was in AX)
AX = ITEM_NOT_FOUND	the requested ECB was not found in the hold queue or the hold queue is empty

◆ *Note:* Link Support Layer ECBs in the hold queue are "at risk" of being reused if the Link Support Layer runs short of ECBs for incoming packets. Refer to Appendix C for details on the ECB format.

ScanPacket

This command scans the hold queue in search of ECBs that correspond to the stack ID in AX and have CX matching the first word of the ProtoWS field. A new scan is started by passing ES:SI as 0:0. The scan is continued by calling the command with ES:SI which still contains the return value of the previous invocation.

Interrupts must remain disabled while scanning the hold queue. To remove an ECB from the hold queue, call GetHeldPacket with ES:SI containing the value returned from ScanPacket but leave interrupts off until GetHeldPacket returns.

Assumes

- BX = 15
- Register preserved: DS, SS, SP and BP
- Interrupts: Disabled on entry
- ES:SI contains a pointer to the previous ECB returned or 0:0 for a first-time call.
- AX contains a stack ID to use as a filter to scan for ECBs
- CX contains a value to match with the first word of the ProtoWS field of the ECB. If CX = 0FFFFH, then the match with the ProtoWS value is ignored

Returns

- Interrupts: Remain disabled
- AX = 0; ES:SI returns the next ECB in the hold queue with the desired stack ID
- AX < 0; an error occurred:

AX = NO_MORE_ITEMS	there are no more matching items in the hold queue

GetStackIDfromName

This command allows a protocol stack or application to obtain its own or any other stack ID. With this information and the board number of an MLID, a stack can obtain the protocol ID, which is used for sending packets. Once the stack ID and board number are known, the stack uses the GetPIDfromStackIDBoard call to obtain the protocol ID.

For the ASCII character set only (in other words, when the high bit is clear), the match of the stack name will be case insensitive . Any other characters that have the high bit set must match exactly.

Assumes

- BX = 16
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Unspecified

- ES:SI contains a pointer to a string containing the name of a protocol stack

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- AX = 0; a protocol stack corresponding to the name was found. BX returns the stack ID for this stack
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	the protocol stack name passed is not registered with the Link Support Layer
AX = BAD_PARAMETER	the length of the name is 0 or greater than 15

GetPIDfromStackIDBoard

This command returns a protocol ID corresponding to a protocol stack ID and a board number. A protocol stack uses this information to fill in the ProtoID field in a send ECB.

Assumes

- BX = 17
- Registers preserved: DS, SS, SP, and BP
- Interrupts: unspecified
- AX contains a stack ID.
- CX contains a board number
- ES:SI points to a 6-byte area into which the protocol ID is returned

Returns

- Interrupts: returned the same way they were entered, and are not enabled inside the routine
- AX = 0; a match was found. The 6 bytes corresponding to the Protocol ID at ES:SI are filled in
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	there is no protocol ID associated with the parameters passed (this probably means that no MLID can use this protocol)
AX = BAD_PARAMETER.	either the protocol stack ID or the board number does not exist

GetMLIDControlEntry

This command returns the MLID Control Entry Point for the MLID corresponding to the board number passed in AX. This command allows a protocol stack to communicate directly with an MLID and obtain information such as the addresses of the MLID Configuration Table and the MLID Statistics Table.

Assumes

- BX = 18
- Registers preserved: DS, SS, SP, and BP
- Interrupts: unspecified
- AX contains the board number for the desired MLID Control Entry Point.

Returns

- Interrupts: are returned the same way as they were entered, and are not enabled inside the routine
- AX = 0; the board number in AX exists. ES:SI contains the MLID Control Entry Point
- AX < 0 ; an error occurred:

AX = ITEM_NOT_PRESENT	there is no MILD with a board number of AX, but there may be others at a higher AX value
AX = NO_MORE_ITEMS	the board number in AX does not exist, and there are no MLIDs registered at higher AX values

GetProtocolControlEntry

This command allows a protocol stack or application to communicate directly with another stack and to obtain information from the Link Support Layer's list of known protocol stacks.

Assumes

- BX = 19
- Registers preserved: DS, SS, SP, and BP
- Interrupts: unspecified
- AX contains a stack ID, starting at 0. If AX contains 0FFFFH, then CX contains the board number for which the Protocol Control Entry Point of the default protocol stack is desired. If AX contains 0FFFEH, then CX contains the board number for which the Protocol Control Entry Point of the PreScan stack is desired

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- AX = 0; a stack exists corresponding to the stack ID in AX. ES:SI will contain the address of the Protocol Stack Control Entry Point.

- AX < 0; an error occurred:

  AX = NO_MORE_ITEMS — there is no MLID with a board number of CX, and there are no others at a higher CX value; or there is no stack with a stack ID of AX, and there are no others at a higher AX value

  AX = ITEM_NOT_PRESENT — there is no MLID with a board number of CX, but there may be others at a higher CX value; or there is no stack with a stack ID of AX, but there may be others at a higher AX value

GetLinkSupportStatistics

This command obtains a pointer to the Link Support Layer's statistics table. See Appendix F for a description of the format of this table.

Assumes

- BX = 20
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Unspecified

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- AX = 0
- ES:SI returns a pointer to the Link Support Layer's statistics table

BindStack

This command binds a protocol stack to an MLID allowing a protocol stack to receive packets from the MLID.

Assumes

- BX = 21
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Unspecified
- AX contains the stack ID to bind
- CX contains the board number to bind

Returns

- Interrupts: Are returned the same way they were entered and are not enabled inside the routine
- AX < 0; an error occurred:

AX = BAD_PARAMETER.	the MLID corresponding to the requested board number or the protocol stack corresponding to the requested stack ID does not exist
AX = DUPLICATE_ENTRY	the specified binding already exists

UnbindStack

This command unbinds a protocol stack from an MLID. After this call is completed, the protocol stack will no longer receive packets from the MLID to which it was once bound unless the protocol stack is also registered as a default receiver for the MLID.

Assumes

- BX = 22
- Registers preserved: DS, SS, SP, and BP
- Interrupts: Unspecified
- AX contains the stack ID to unbind
- CX contains the board number to unbind

Returns

- Interrupts: Are returned the same way they were entered, and are not enabled inside the routine
- AX < 0; an error occurred:

AX = ITEM_NOT_PRESENT	the specified binding does not exist
AX = BAD_PARAMETER.	the MLID, corresponding to the requested board number or the protocol stack corresponding to the given stack ID does not exist

AddProtocolID

This command allows a protocol stack to add a new protocol ID for a given media.

Assumes

- BX = 23
- Registers preserved: DS, SS, SP and BP

- Interrupts: Enabled on entry
- ES:SI points to the 6-byte protocol ID being added
- CX:DI points to a string containing the name of the protocol stack for this protocol ID. The maximum length of the string is 15 characters. The string *must* have a leading length byte and a trailing zero byte.
- AX contains the media ID for which the new protocol ID is being added

Returns

- Interrupts: Enabled on exit.
- AX = 0; the Link Support Layer successfully added the new protocol ID
- AX < 0; an error occurred:

AX = OUT_OF_RESOURCES	the Link Support Layer has no resources to register another protocol ID
AX = DUPLICATE_ENTRY	there is already a protocol ID registered for the given media type/protocol stack combination
AX = BAD_PARAMETER.	the specified parameter is an illegal (unknown) name. The field length will be equal or less than 15

◆ *Note*. This call should only be made for PID information in the NET.CFG file because of the limited number of protocol stack slots.

RelinquishControl

This command allows a protocol stack to yield control to the Link Support Layer, allowing the Link Support Layer to perform any necessary background processing. The background processing includes polling MLIDs if the poll bit is set in the Mode Flags field of the MLID Configuration Table (see Appendix A). If the bit is set, the DriverPoll command of the MLID will be called. This call should be made by any protocol stack that is waiting for an event to occur. For example, the protocol stack could be waiting for a SendPacket to complete.

This call need not be made under a multitasking operating system. A protocol stack should yield to the operating system or allow concurrent processing while waiting for an event. Under a multitasking operating system, this call does nothing.

Assumes

- BX = 24
- Registers preserved: DS, SS, SP, and BP

- Interrupts: enabled on entry

Returns

- Interrupts: Enabled on exit
- AX = 0. No errors are possible

Appendix A **MLID Configuration Table**

The MLID Configuration Table contains information on the MLID and its configuration. Variables in the file can include items such as the interrupt number and port I/O address.

◆ *Note:* All data strings in the configuration table consist of a one-byte length (not counting the terminating 0 byte), the data string itself, and a terminating 0 (null) byte.

Offset	Name	Bytes	Description
0	Signature	26	Contains the ASCII Data "HardwareDriverMLID" padded with trailing spaces.
26	CFG_MajorVersion	1	The major version number of the configuration table (1 for this specification).
27	CFG_MinorVersion	1	The minor version number of the configuration table (0 .. 99 decimal, for example, 31 indicates version X.31). This number should be 0 for this specification.
28	NodeAddress	6	Contains the node address of the MLID in network order, padded on the left (lower addresses) with 0s. For example, a one-byte address xx would be filled in as 00 00 00 00 00 xx from low to high memory.
34	ModeFlags	2	Bit mask for MLID information: Bit 0 Set if a real driver Bit 1 Set if driver uses DMA Bit 2 Set if driver is 100% reliable on transmit Bit 3 Set if driver supports multicast Bit 4 Set if driver supports promiscuous mode Bit 5 Set if driver needs polling Bit 6 Set if driver supports raw send mode
36	BoardNumber	2	The board number identifier supplied by the Link Support Layer (0 ...?)
38	BoardInstance	2	Represents the board instance number. This contains an instance identifier to identify which of several boards (supported by an MLID) correspond to this particular table
40	MaxDataSize	2	Contains the maximum number of bytes that the driver can transmit, not counting the link-level envelope. This must be a number greater than 585. If hardware cannot support at least 586 bytes, the driver must be written to send multiple packets and reconstruct them
42	MaxRecvSize	2	Best case room in ECB for received packets. This corresponds to the LSL's ECB size less the size of the smallest link-level header for the media.
44	RecvSize	2	Worst case room in ECB received packets. This corresponds to the LSL's buffer size less the size of the largest link-level header for the media.
46	CardName	4	Pointer to data string containing a name uniquely identifying the interface card hardware
50	ShortName	4	Pointer to data string (7 characters max) containing a short name uniquely identifying the MLID in the NET.CFG file

54	MediaType	4	Pointer to data string containing the media type for the MLID (for example, "\010EtherNet\0")
58	CardID	2	Word containing a Novell/Apple–administered ID for the interface card
60	MediaID	2	Word containing a Novell/Apple-administered ID for the media/link-level envelope combination

Offset	Name	Bytes	Description
62	TransportTime	2	Number of milliseconds required to transmit a 586-byte packet. (*Note*: This number must be set to at least 1 millisecond)
64	Reserved	16	Must be set to 0
80	MLID_MajorVersion	1	The major version number of the MLID
81	MLID_MinorVersion	1	The minor version number of the MLID (0 .. 99 decimal; for example, 31 indicates version X.31)
82	Flags	2	This field contains flags that are operating-system dependent. For the MS-DOS-OS/2 environment the bits are defined as follows: Bit 0 = 1 if MLID can operate in *real* Mode Bit 1 = 1 if MLID can operate in *protected* Mode Bits 2-3 = 00 for MicroChannel architecture if the MLID is to scan for the card. Bits 2-3 = 01 when dual support for At Bus and MicroChannel architecture support is provided Bits 2-3 = 10 for MicroChannel architecture if the card slot is fixed. Bits 2-3 = 11 for non-MicroChannel configuration.
84	SendRetries	2	Number of retries the MLID should perform before failing a transmit (the number of actual retries may be hardware-dependent).
86	Link	4	Link pointer field for use by the Link Support Layer
90	ShareFlag	2	Bits indicating sharing capability of the MLID Bit 0 = set if MLID is shut down Bit 1 = set if MLID can share I/O port #1 Bit 2 = set if MLID can share I/O port #2 Bit 3 = set if MLID can share memory range #1 Bit 4 = set if MLID can share memory range #2 Bit 5 = set if MLID can share interrupt #1 Bit 6 = set if MLID can share interrupt #2 Bit 7 = set if MLID can share DMA channel #1 Bit 8 = set if MLID can share DMA channel #2 All other bits are undefined and must be set to 0
92	Slot	2	Contains the slot number of the interface card in configurations where a slot number is appropriate (for example, MicroChannel architecture)
94	IOAddr1	2	Contains the primary I/O address for the interface card

Offset	Name	Bytes	Description
96	IORange1	2	Contains the number of I/O ports used at IOAddr1
98	IOAddr2	2	Contains the secondary I/O address for the interface card
100	IORange2	2	Contains the number of I/O ports used at IOAddr2
102	MemAddr1	4	Contains the primary memory address used by the interface card
106	MemSize1	2	Contains the number of paragraphs used by the interface card at MemAddr1
108	MemAddr2	4	Contains the secondary memory address used by the interface card
112	MemSize2	2	Contains the number of paragraphs used by the interface card at MemAddr2
114	Int1Line	1	The IRQ number for the first interrupt that the MLID uses. Set to 0FFH if not used
115	Int2Line	1	The IRQ number for the second interrupt that the MLID uses. Set to 0FFH if not used
116	DMA1Line	1	The DMA channel number for the first DMA channel that the MLID uses. Set to 0FFH if not used
117	DMA2Line	1	The DMA channel number for the second DMA channel that the MLID uses. Set to 0FFH if not used

Appendix B **Protocol Stack Configuration Table**

The following table provides information that helps you reference stacks by name and version. This table is especially helpful if you want to search for a particular stack by name.

Offset	Name	Bytes	Description
0	CFG_MajorVersion	1	Major version number of the configuration table (1 for this specification)
1	CFG_MinorVersion	1	Minor version number (0..99 decimal) of the configuration table (0 for this specification)
2	Name	4	Address of a string with the name of the protocol stack. This name may be longer than the name used to register the stack with the Link Support Layer.The name is preceded by a length byte and terminated with a 0 byte
6	ProtocolName	4	Address of the protocol name (15 characters maximum) used to register the stack with the Link Support Layer; also used in the AddProtocolID and in the NET.CFG file. The name is preceded by a length byte and terminated with a 0 byte
10	Stack_MajorVersion	1	Major version number of the protocol stack
11	Stack_MinorVersion	1	Minor version number (0..99 decimal) of the protocol stack
12	Reserved	16	must be set to 0

Appendix C **ECB Format**

The following table lists the fields used in the ECB.

Offset	Name	Bytes	Description
0	FLink	4	Forward link for queueing
4	BLink	4	Backward link for queueing. For AES Event ECBs this dword is the number of milliseconds that need to elapse before the ESR routine is called
8	Status	2	Status word: >0 in progress 0 completed successfully <0 completed with error)
10	ESR	4	Address of a FAR Procedure to call when the ECB has completed.
14	StackID	2	Stack ID (or 0FFFFH for a Send ECB to indicate a "raw" transmit)
16	ProtoID	6	Protocol ID
22	BoardNo	2	A board number
24	ImmAddr	6	Network address of source node (on receives) or destination node (on transmits)
30	DriverWS	4	A 4-byte work area for the MLI
34	ProtoWS	8	An 8-byte work area for protocol stacks. This area will not be modified by MLIDs or the Link Support Layer.
42	SendLen	2	Word containing the full data length of a send buffer, counting all fragments
44	FragCnt	2	Number of fragments in following fragment list
46	FragPtr1	4	Pointer to the first fragment
50	FragLen1	2	Length of the first fragment
52	Envelope	xx	In receive ECBs, the incoming packet is stored here

Appendix D MLID Statistics Table Format

All MLID modules must keep a statistics table for network management purposes. The following is the format of an MLID statistics table.

Offset	Name	Bytes	Description
00	STAT_MajorVersion	1	Major version number of the statistics table (1 for this specification)
01	STAT_MinorVersion	1	Minor version number of the statistics table (0..99 decimal, 0 for this specification)
02	GenericCnts	2	Number of 4 byte counters in fixed portion of table
04	ValidCntsMask	4	Bit mask indicating which counters are valid. The value, 0 indicates Yes. The value 1 indicates No. The bit/counter correlations are determined by shifting left, as you move down the counters in the table. So bit 7 of the 4th byte corresponds to the first counter, as shown in the following illustration. Similarly, Bit 0 of the first byte corresponds to the 32nd counter (if present).

Valid Counters Mask

Bits	7	6	5	4	3	2	1	0
Byte 1	25	26	27	28	29	30	31	32
Byte 2	17	18	19	20	21	22	23	24
Byte 3	9	10	11	12	13	14	15	16
Byte 4	1	2	3	4	5	6	7	8

Counters

08	TotalTXPackets	4	Total number of packets that were requested to be transmitted (whether they were actually transmitted or not)
12	TotalRXPackets	4	Total number of incoming packets received
16	NoECBsAvail	4	Number of incoming packets that were lost because of unavailable ECBs
20	TXTooBig	4	Number of requested packets for transmission that were too big to send
24	TXTooSmall	4	Number of requested packets for transmission that were normally too small to be transmitted. The packets may still have been sent if the MLID does padding
28	RXOverflow	4	Number of incoming packets that were lost because they were bigger than the ECB buffer size
32	RXTooBig	4	Number of incoming packets that were bigger than the maximum legal size for the media
36	RXTooSmall	4	Number of incoming packets that were smaller than the minimum legal size for the media
40	TXMiscErr	4	Number of transmissions requests that were not sent because of errors other than those explicitly listed in this table
44	RXMiscErr	4	Number of incoming packets that were lost because of errors other than those explicitly listed in this table
48	TXRetryCount	4	Total number of retries invoked to send packets
52	RXChksumErr	4	Total number of incoming packets lost due to Checksum/CRC errors
56	RXMismatch	4	Total number of incoming packets lost due to conflicting information given by the hardware and the packet internals
60	NumCustom	2	Total number of custom variables following this word

There are NumCustom dwords starting at offset 56 that correspond to the custom statistics for the MLID. Following these dwords, there are NumCustom pointers (4 bytes each) that point to strings describing the custom statistics. The strings have a leading length byte and a terminating 0 byte.

Appendix E Protocol Stack Statistics Table Format

All protocol stacks must keep a statistics table for the purpose of network management. Any statistics that are not appropriate for a given protocol stack should be set to 0FFFFFFFFH. The following is the format of a Protocol Stack Statistics Table.

Offset	Name	Bytes	Description
00	STAT_MajorVersion	1	Major version number of the statistics table (01h for this specification)
01	STAT_MinorVersion	1	Minor version number of the statistics table (0..99 decimal, 00h for this specification)
02	GenericCnts	2	Number of 4-byte counters in fixed portion of table
04	ValidCntsMask	4	Bit mask indicating which counters are valid. A value of 0 indicates Yes. A value of 1 indicates No. The bit/counter correlations are determined by shifting left, as you move down the counters in the table.
08	TotalTXPackets	4	Total number of packets that were requested to be transmitted (whether they were actually transmitted or not)
12	TotalRXPackets	4	Total number of incoming packets received
16	IgnoredRXPackets	4	Total number of incoming packets that were ignored by the stack
20	NumCustom	2	Total number of custom variables following this word

There are *NumCustom* dwords starting at offset 16 that correspond to the custom statistics for the protocol stack. Following these dwords are *NumCustom* pointers (4 bytes each) that point to strings describing the custom statistics. The strings have a leading length byte and a terminating 0 byte.

Appendix F Link Support Layer Statistics Table Format

The Link Support Layer will keep a statistics table for the purpose of network management. The following is the format of the Link Support Layer Statistics Table.

Offset	Name	Bytes	Description
00	STAT_MajorVersion	1	Major version number of the statistics table (1 for this specification)
01	STAT_MinorVersion	1	Minor version number of the statistics table (0..99 decimal, 0 for this specification)
02	GenericCnts	2	Number of 4-byte counters in fixed portion of table
04	ValidCntsMask	4	Bit mask indicating which counters are valid. A value of 0 indicates Yes. A value of 1 indicates No. The bit/counter correlations are determined by shifting left, as you move down the counters in the table.
08	TotalTXPackets	4	Total number of packets that were requested to be transmitted (whether they were actually transmitted or not)
12	GetECBBfrs	4	Total number of GetECB requests
16	GetECBFails	4	Number of GetECB requests that failed because of unavailable resources
20	AESEventCounts	4	Number of completed AES events
24	PostponedEvents	4	Number of events that were postponed because of StartCriticalRegion functions
28	ECBCxlFails	4	Number of ECB cancel events that failed
32	ValidBfrsReused	4	Number of ECBs on the hold queue that were reused before they were removed from the hold queue

36	EnqueuedSendCnt	4	Number of EnqueueSend events that have occurred
40	TotalRXPackets	4	Number of incoming packets dispatched
44	UnclaimedPackets	4	Number of incoming packets that were not claimed by any protocol stack
48	NumCustom	2	Total number of custom variables following this word

There are *NumCustom* dwords starting at offset 44 that correspond to the custom statistics for the Link Support Layer. Following these dwords, there are *NumCustom* pointers (4 bytes each) that point to strings describing the custom statistics. The strings have a leading length byte and a terminating 0 byte.

Appendix G System Error Codes

The following error codes are defined for the network system.

OUT_OF_RESOURCES	8001H	There are no resources available to execute the desired function
BAD_PARAMETER	8002H	One of the parameters passed to this function is unclear
NO_MORE_ITEMS	8003H	There are no more items to return
ITEM_NOT_PRESENT	8004H	The item that you requested was not found
FAIL	8005H	An unspecified failure occurred
RX_OVERFLOW	8006H	The received packet was an overflow packet and may be in error
CANCELLED	8007H	The ECB associated with this error code was cancelled by an MLIDShutdown or an explicit cancel call
BAD_COMMAND	8008H	The value passed in BX does not correspond to a legitimate command
DUPLICATE_ENTRY	8009H	The command or address you tried to add is already present
NO_SUCH_HANDLER	800AH	The protocol stack you tried to send a command to has been deregistered
NO_SUCH_DRIVER	800BH	The MLID you tried to send a command to has been shut down

Appendix H NET.CFG Configuration File Format

The NET.CFG file contains the configuration information for the network system. It is a control file that contains section headings and subsidiary information. This appendix documents the section headings and the currently defined subsidiary information. However, protocol stacks and MLIDs may define new keywords and information to be stored in the file that are specific to the MLIDs and protocol stacks. All keywords and main headings are case insensitive.

The following is the format of the NET.CFG file:

```
Main heading
          Sub info1
          sub info 2
```

The main headings always start in the first column of the file, and all subsidiary information starts in any column other than the first. Not all MLIDs or protocol stacks need to understand all of the possible keywords. The MLID or protocol stack designer determines which keywords need to be understood.

The following main headings are allowed:

Link support The subsidiary information up to the next main heading describes parameters for the Link Support Layer.

Protocol <name> The subsidiary information up to the next main heading describes information for the named protocol stack. This name corresponds to the protocol name given to the Link Support layer at initialization time by the protocol stack.

Link driver <name> The subsidiary information up to the next main heading describes information for the named MLID. This name corresponds to the ShortName field in the MLID Configuration Table.

A # in column one indicates a comment line and will be ignored.

In the following definitions, the following conventions are used:

[]	optional element inside brackets
n#	decimal number, digit # is for differentiation
h#	hexadecimal number, digit # is for differentiation

Link support keywords

Buffers n1 n2

Configures the number of receive buffers (n1) and their size (n2) that the Link Support Layer will create. N2 must be at least 586

MemPool n1[k]

Configures the size of the memory pool that the Link Support Layer will maintain. The "k" notation has the usual meaning (multiply by 1,024)

Protocol stack keywords

Sessions n1

Configures the number of sessions that the protocol stack will be required to maintain at one time

Bind <Name>

Requests the protocol stack to bind with the MLID <Name>

Default <Name>

Requests the protocol stack to bind with the MLID <Name> and sets this MLID to the default MLID if appropriate

Link driver keywords

DMA [#n1] n2

Configures DMA channel n1 (where n1 is 1 or 2, and is assumed 1 if absent) to be channel n2

INT [#n1] n2

Configures the n1th interrupt number (where n1 is 1 or 2, and is assumed 1 if absent) to be interrupt number n2

MEM [#n1] h1 h2

Configures the n1th memory range (where n1 is 1 or 2, and is assumed 1 if absent) to be at address h1 for h2 paragraphs. The h2 is assumed to be 1 if not present

PORT [#n1] h1 h2

Configures the n1th I/O port range (where n1 is 1 or 2, and is assumed 1 if absent) to be at I/O port address h1, for h2 ports. The h2 is assumed to be 1 if not present

PS/2 Slot n1

The n1 is a number that indicates the PS/2 slot containing the card for this MLID. (Slot number is 1-based. In other words, the first slot number is 1, not 0)

PS/2 Slot ?

Indicates that the MLID must scan for the PS/2 slot that contains its card. This is the default if not present

Node Address h1

Overrides any hard-coded node address in the MLID's hardware, if the hardware allows it

Protocol <name> h1

Tells the MLID that the named protocol has a protocol type of h1. This allows new protocols to be handled by existing MLIDs

SendRetries n1

Configures the MLID to attempt n1 retries on transmitted packets. n1 may be 0

Envelope type name

Configures which link-level envelope type the driver will create if there is a possibility of more than one type (see appendix J for list of names)

Appendix I **Finding the Link Support Layer in DOS**

When running with MS-DOS, the Link Support Layer is found using the INT 2FH multiplexing interrupt with code similar to the following:

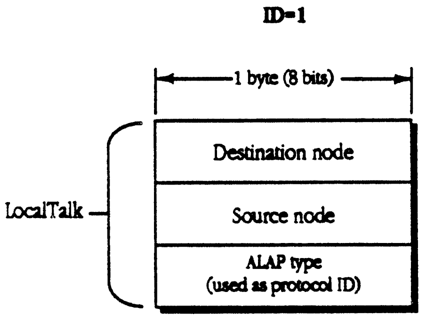
```
cmpr:   db      'LINKSUP$'

;
;   This first piece of code is needed for DOS 2 since DOS did not support the
;   multiplexing interrupt in that version of DOS.  We make sure the vector is
;   not 0 or OFFFFH:OFFFFH.  The Link Support Layer will properly intercept
;   int 2FH even under DOS 2.
;
        sub     ax,ax
        mov     es,ax
        cli
        mov     cx,es:[2FH*4]
        mov     dx,es:[2FH*4+2]
        sti
        mov     ax,cx
        or      ax,dx
        jz      bad                     ; vector was 0 - can't be loaded
        mov     ax,cx
        and     ax,dx
        cmp     ax,-1
        jz      bad                     ; vector was OFFFFH:OFFFFH - can't be loaded
        sub     bx,bx
        mov     es,bx
        mov     ah,0c0h                 ; Scan through the multiplex numbers COH thru OFFH
        sub     al,al
lp1:    push    ax
        int     2FH
        cmp     al,offh                 ; If AL returns OFFH, something is using this mx
                                        number
        pop     ax
        jz      fnd
nxt:    inc     ah
        jnz     lp1
;
```

```
; Here, the Link Support Layer was not found
;
bad:
       Tell the user that the Link Support Layer must be loaded
;
; Here, an int 2FH entity was found - see if it is the Link Support Layer
;
fnd:   mov    ax,dx         ; Did this routine set DX:BX?
       or     ax,bx
       jz     nxt           ; No - try next mx number
       push    ds
       mov di, si           ; ES:DI points to description record
       mov    si,cs
       mov    ds,si
       lea    si,cmpr              ;Set DS:SI to point to comparison String
       cld
       mov    cx,8
       repe   cmpsb
       pop    ds
       jz     fndit         ; Was the signature there?
       jmp    nxt           ;Not the right one  - try again
;
; Here, we found the Link Support Layer, and DX:BX contains its Initialization Entry
; Point.
;
fndit:
```

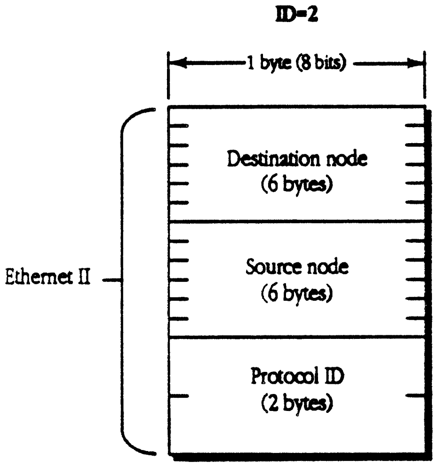
Appendix J **Defined Media IDs**

The following media IDs/names are currently defined:

Media ID	Name	Description
01	LocalTalk	The Apple LocalTalk media

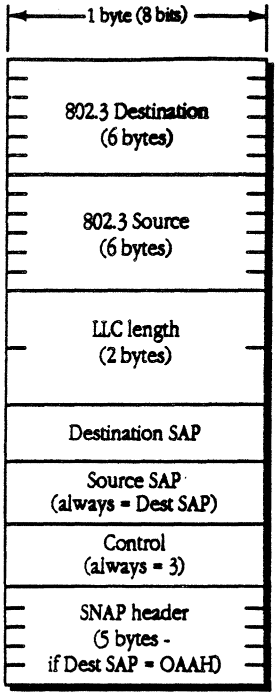


02	Ethernet II	Ethernet using a DEC Ethernet II envelope



03	Ethernet 802.2	Ethernet using an 802.2 envelope

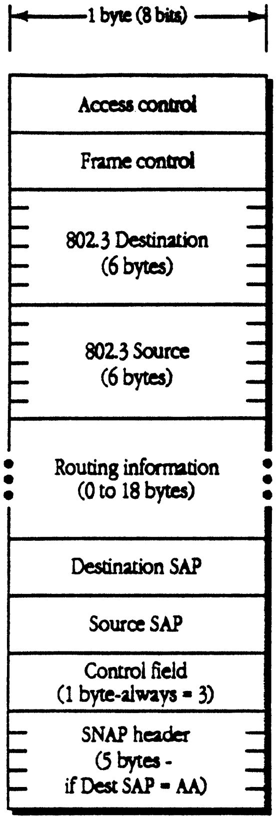
ID=3



Protocol ID =
1 byte Dest SAP
(if not OAAH) or
OAAH + 5 bytes SNAP header

04	TokenRing	Token Ring using an 802.2 envelope

ID=4


1 byte (8 bits)
Access control
Frame control
802.3 Destination (6 bytes)
802.3 Source (6 bytes)
Routing information (0 to 18 bytes)
Destination SAP
Source SAP
Control field (1 byte-always = 3)
SNAP header (5 bytes - if Dest SAP = AA)

Protocol ID =
1 byte Dest SAP
(if not OAAH) or
OAAH + 5 bytes SNAP header

Appendix K **Defined Card IDs**

The following card IDs and card names are currently understood by the MLI/MPI. In this table each particular card ID has to be mapped to its particular card.

Card ID	Name	Description
01	Apple LocalTalkPC	The Apple LocalTalk PC card
02	EtherLink I	The 3Com EtherLink I adapter
03	EtherLink II	The 3Com EtherLink II adapter
04	EtherLink/MC	The 3Com EtherLink/MC adapter
05	IBM 802.2	MLID for the IBM 802.2 interface using IBM Token Ring cards

Using the DOS MLID Startup Modules

This document provides direction on using the DOS MLID Startup Modules and should be used with the *ODI Developer' Guide*, available from APDA (Apple Programmer's and Developer's Association) and Novell, Inc. The modules consist of executable code and help you get started writing your own MLIDs. The modules provide much of the pedestrian work involved in writing MLIDs (such as registering the drivers and protocol stacks, reading the NET.CFG file, and calling Service Events). You only need to link your own code to the modules.

Your DOS MLID Startup modules consist of two files: DSTARTUP.OBJ and DRIVER.OBJ. By linking these two files with some of your well defined routines, you can write an MLID faster and easier than writing all of the interface code required to implement an MLID.

△ Important

The file DSTARTUP.OBJ must be linked as the first object module, since it contains the segment declarations; these segments must also be in the following order.

The following are the segment declarations in DSTARTUP.OBJ:

DGROUP	group _TEXT, _DATA, CONST, _BSS, LDATA, IDATA, ICNST, IBSS, ITEXT
_TEXT	segment word public 'CODE'
_DATA	segment word public 'DATA'
CONST	segment word public 'DATA'
_BSS	segment word public 'DATA'
LDATA	segment para public 'DATA'
IDATA	segment word public 'INIT'
ICNST	segment word public 'INIT'
IBSS	segment word public 'INIT'
ITEXT	segment word public 'INIT'

◆ *Note:* All segments are part of the GROUP. This allows you to create a .COM file, if this is desirable.

Variables Declared by DSTARTUP.OBJ/ DRIVER.OBJ

The following variables are declared public by the startup code. The only exception is that _errmsg are filled in with the proper values by the startup code before _driver_init_config_ is called.

- _MyDgroup in segment _TEXT

  This variable is a WORD that contains the value of DGROUP. This WORD is in the _TEXT segment so that a CS-relative reference can fetch the value. Although the code and data are in DGROUP together, the code becomes very difficult to port to OS/2 if you rely on this fact. For example, you can't assume that CS always equals DS because this may not be true when you port this driver to OS/2. There are only a few variables in the _TEXT segment, and these variables are filled in by the startup code.

- _Link_Support in segment _TEXT

  This is a DWORD that contains the far address of the MLID Support Entry Point of the Link Support Layer. This variable is called by loading BX with the desired support function code, and executing a FAR CALL through this variable.

- _max_ecb_size in segment _DATA

  This is a WORD that contains the maximum size of the data portion of an ECB.

- _errmsg in segment IDATA

  This is a WORD that you can fill in with a near pointer to an error message if your hardware initialization fails (See the *Initialization* section next).

Permanent Variables Declared by the MLID code

The following variables must be declared public by your MLID code:

- _swap_stack in segment _DATA

  This variable is a WORD that contains a 0 if you do not want to swap stacks when an interrupt occurrs. Otherwise, it should contain the DGROUP-relative offset of the initial SP value which should be loaded whenever an interrupt occurrs.

- _card_name in segment CONST

This is the address of a data string (preceded by a length byte, and terminated with a 0-byte) that describes the interface card name. A pointer to this string will be stored in the CardName field of the configuration table for each instance of the MLID.

- _driver_name in segment CONST

  This is the address of a data string (preceded by a length byte, and terminated with a 0-byte) that corresponds to the short name of your driver. A pointer to this string will be stored in the ShortName field of the configuration table for each instance of the MLID. This is the name that the user must specify in the NET.CFG file.

- _card_id in segment CONST

  This is your Card ID. This value will be stored in the CardID field of the configuration table for each instance of the MLID.

- _ws_offsets in segment _DATA

  This is a table of four WORDs containing the offsets of the four workspace areas for up to four instances of the MLID. These offsets must be relative to DGROUP, and must have a value other than 0 to be valid. Use 0's for instances that the driver does not support. (For example, if the MLID only has two instances, set the third and fourth words of this array to 0.) These workspaces *must* be in the segment LDATA so the data memory for the unused instances can be freed when the MLID terminates and stays resident.

- _driver_ctl_tab in segment _DATA

  This is a table that contains a list of offsets (relative to DGROUP) to the various routines which implement the MLID Control Routines. There should be 'NUM_DRIVER_CTLS' entries in this table. Any calls that have a BX value greater than or equal to 'NUM_DRIVER_CTLS' will be dispatched through '_driver_ctl_dflt_'. Any routines dispatched by this table will have DS:BX pointing to the appropriate instance data.

Public Constants by the MLID code

The following constants must be declared public by the MLID code:

VER_MONTH The month of this MLID revision (1 ... 12).

VER_DAY The date of this MLID revision (1 ... 31)

VER_YEAR The year of this MLID revision (0 ... 99)

VER_MAJOR The major version number of this MLID.

VER_MINOR The minor version number of this MLID revision (0 ... 99 decimal).

NUM_DRIVER_CTLS The total number of procedure address contained in '_driver_ctl_tab'.

OFFSET_0 As a first option, this constant should contain 0 if you want your instance data described as DS = DGROUP, BX = offset value. As a second option, this constant should be set to 1 (or a value other than 0) if you want your instance data described as DS = <<desired value>>, BX = 0. For this second option to work correctly, all offsets in the '_ws_offsets' table must be on paragraph boundaries. For this reason, your instance data should be in the LDATA segment (it's paragraph aligned).

Temporary Variables Declared by the MLID code

The following variables should be declared public by the MLID code:

- _signon_msg in segment ICNST

  This is a string (not preceded by a length byte), terminating with a '$' character which is your sign-on message. This string will be output using DOS Function 09H as the first action of the startup code.

- _loaded_msg in segment ICNST

  This is a string (not preceded by a length byte), terminating with a '$' character which is the message you want to output to the console if the MLID is already loaded. This string will be output using DOS Function 09H if the startup code determines that the MLID is already loaded.

Public Procedures in the MLID code

This section describes the required routines you must implement to have a functional MLID. The following public procedures should be present in the MLID code in the _TEXT segment:

- _driver_send_

  This near procedure is called whenever a packet is to be sent. DS:BX will point to the appropriate instance data, and ES:SI will contain the send ECB. Your routine must preserve only BP.

- _driver_ctl_dflt_

  This near procedure is called whenever a Driver Control call is dispatched that is not in '_driver_ctl_tab'. DS:BX will point to the appropriate instance data, and BP will contain the value of BX that caused the call. Your routine does not need to preserve any register.

- _driver_isr1_

  This near routine is called whenever a card interrupt occurs on Int1Line. All registers except BP have already been saved, the stack has been swapped (if that feature was enabled using the _stack_swap variable), and DS:BX is pointing to the instance data for the board that caused the interrupt. Your routine should return:

  AX = 0 if the interrupt was not intended for this driver. The next chained interrupt handler would then be called.

  AX = 1 for all commands other than those to get next handler, call ServiceEvents, or call EndCriticalSection.

  AX = 2 to call ServiceEvents.

  AX = 3 to call EndCriticalSection.

  The startup code keeps track of nested invocations of the ISRs, and will not actually dispatch ServiceEvents until the last nested interrupt is processed.

- _driver_init_config_

  This routine is called to check the validity of the configuration table. The configuration table already has the fields defined in NET.CFG filled in. This routine may also fill in the configuration table with other driver-specific values. When this routine is entered, the DWORD address of the configuration table for this instance is pushed on the stack if there is an error in the configuration. This routine should return the following condition:

  If an error message occured, AX = 0 and _errmsg = DGROUP offset of the error message. If there is no error, this routine should return AX = a value other than 0.

- _driver_isr2_

  This near routine is called whenever a card interrupt occurs on Int2Line. All registers except BP have already been saved, the stack has been swapped (if that feature was enabled using the _stack_swap variable), and DS:BX is pointing to the instance data for the board that caused the interrupt. The startup code keeps track of nested invocations of the ISRs, and will not actually dispatch ServiceEvents until the last nested interrupt is processed. Refer to _driver_isrl_ for return values.

◆ *Note*: This procedure must be present even if the MLID does not use the Int2Line interrupt. In this case, just create a public label somewhere in your code to satisfy the external reference requirement.

- _driver_init_

  This near routine is called to initialize a driver instance. DS:BX points to the configuration table corresponding to the driver instance. If initialization occurred, this routine must return (in AX) a near pointer to the byte after the instance data without any errors. Otherwise, this routine should return a 0, and store a near pointer into '_errmsg' with the error string (terminated with a '$'). The startup code will print this error message, shutdown any other instances that are already initialized, and terminate to DOS without staying resident. The configuration table should be examined for correctness if this has not already been done, and any '_install_ints' calls should be made to install the interrupt handlers for the driver.

Public Procedures in the Startup code

This routine returns CL = Int1Line interrupt disable mask for 8259 PIC Interrupt Mask Register (IMR). CH = Int2Line disable mask, DL = port address of IMR for Int1Line, DH = port address of IMR for Int2Line. The following public procedures are present in the Startup code in the _TEXT segment:

- _install_ints_

This near procedure should be called inside of '_driver_init' to install the interrupts configured in the configuration table. DS:BX must contain the address of the configuration table. The existing interrupt vectors will be saved for restoration in _remove_ints.

- _remove_ints_

  This near procedure should be called to remove the interrupts configured in the configuration table and restore them to their previous values. DS:BX must contain the address of the configuration table.

- _set_IRQ_

  This near procedure is made available for those who prefer not to use _install_ints_ listed earlier, or need additional interrupts installed. For example install_inits may not satisfy the requirements of your particular driver. DX:AX contains the address of the new interrupt handler, ES:BX is the address of a dword where the old interrupt handler should be stored, and CX is the IRQ Number.

- _restore_IRQ_

  This near procedure reverses the results of _set_IRQ_. ES:BX is the address where the old interrupt handler was saved and CX contains the IRQ Number.

- _clear_SendQ_

  This near procedure removes all queued ECBs (queued by the LSL or by using the EnqueueSend command) from the send queue and frees them so that the ECBs can be reused. AX must contain the board number of the MLID making the call.