Link Level Library (3L) Interface Specification

For 3Com Network Adapter Products

3165 Kifer Road
Santa Clara, California 95052-8145
Printed in the U.S.A.

Manual Part No. 4205-01
Published January, 1989

Copyright Statement

Disclaimer

Contents

Figures

Chapter 1: Introduction

This document describes the 3Com Link Level Library (3L) Interface. It is intended to assist engineers developing or modifying network software to run on 3Com network adapters that have a 3L implementation available.

3L consists of a set of generic routines that provide the protocols with a flexible means of transmitting and receiving packets over the network, while at the same time isolating the protocols from the specific network adapter hardware in use. Each 3L implementation for 3Com's various adapters interface to any protocols that network 3L to "talk" to the network. 3L implementations allow both Ethernet and token ring physical layers to support the same network operating system.

Chapter 2 describes the concepts and facilities of 3L, and outlines the architecture by identifying general usage of interface routines.

Chapter 3 specifies each routine that is to reside on either side of the 3L interface. For each routine, the input parameters, output parameters, and function are defined, and special notes regarding usage are included, as appropriate.

Chapter 4 describes the error handling conventions to be used by all routines defined in the specification.

Appendix A describes the specific issues for the several 3Com 3L implementations. The appendix has a single section for each 3Com network adapter for which the interface has been implemented.

Appendix B discusses the specifics of linking the routines using the Microsoft OBJ format conventions.

Appendix C provides an example of interfacing protocol side routines with 3L routines.

Chapter 2: Architecture

The following specification is technical in nature, and is intended for experienced 8086 programmers with specific understanding of the architecture of 808x-based and 80x86-based microcomputers, and of programming interrupt driven device drivers. An assembler should be used that generates object modules that are Microsoft/Intel compatible.

The 3L routines are intended to allow protocols to communicate with one or more network adapters. The routines are designed to be directly linked to protocol software. These routines provide four areas of service for the protocols running "above" the 3L-based driver. These are:

- Network adapter initialization
- Delivery of received packets to higher level protocols
- Transmission of packets onto the network
- Control functions

The 3L interface is specified in a manner so as to support the range of network hardware interface architectures in use and under development today. In particular this architecture allows for the synchronous/asynchronous operation of the transmission process. Each implementation of 3L supports one or both of these approaches. Protocol software interfacing to 3L routines may select either one, if both approaches are implemented. Protocol software may determine which transmission approach is available through the **WhoAmI** function call at initialization time. Availability and selections will be based on the following:

- Network controller hardware design
- Specific 3L implementation design
- Performance requirements
- Capabilities built into the protocol software

Synchronous vs. Asynchronous Operation

To provide the most flexibility and performance, the 3L interface presented to the protocols appears to be asynchronous. "Appears to be" is appropriate because a specific 3L implementation may or may not implement an asynchronous interface. Asynchronous drivers will either start the requested operation, or queue up the operation for starting later, as appropriate for the current state of the driver. In either case, an asynchronous driver will return immediately to the routine requesting the operation. At a later time, the asynchronous driver interrupts and informs the requesting routine that the operation previously requested has been completed. This method allows the 3L routines and the protocols to continue simultaneously, improving overall performance. To ease implementation, a synchronous hardware driver may be used in place of a true asynchronous hardware driver. Synchronous 3L routines start the requested operation, wait for it to complete, and inform the requesting routine that the operation has completed before returning to the protocols. The net result is that the synchronous driver appears to be a very fast asynchronous driver to the routine requesting the service, since the requested operation is completed before the called routine returns. Truly asynchronous operation of receive and transmit functions may or may not be able to be provided in a given 3L implementation for a specific network hardware interface. Also, protocols linking to and using the 3L implementation may or may not choose to implement the logic required to use this asynchronous operation capability. Two design features are provided for protocol developers wanting to use this capability. These are the Request Identifier and the NO-WAIT mode post routine use.

Since the interface may support asynchronous operations, a Request Identifier (1-byte long) is provided by the 3L routines to keep track of which request is being referred to in a given routine call. A unique request identifier is generated by the 3L routines at the beginning of each transmit or receive function sequence. The protocols must keep this identifier and pass it to the 3L routines on any subsequent routine call during the processing of that particular transmit or receive packet.

Most of the routines defined in this document comprise the 3L. There are two additional routines defined, **RxProcess** and **TxProcess**, which may be provided by the protocols. Of these, **RxProcess** is required and **TxProcess** provides a capability of executing transmit operations in a no-wait mode. This two-way interface greatly simplifies the interaction of the 3L routines and the protocol routines.

The general flow of control between the 3L routines and the protocols is charted in the following diagrams. The diagrams are split into two halves: protocols and 3L routines. The connections between the two halves of the code are through a small number of well-defined interface routines.

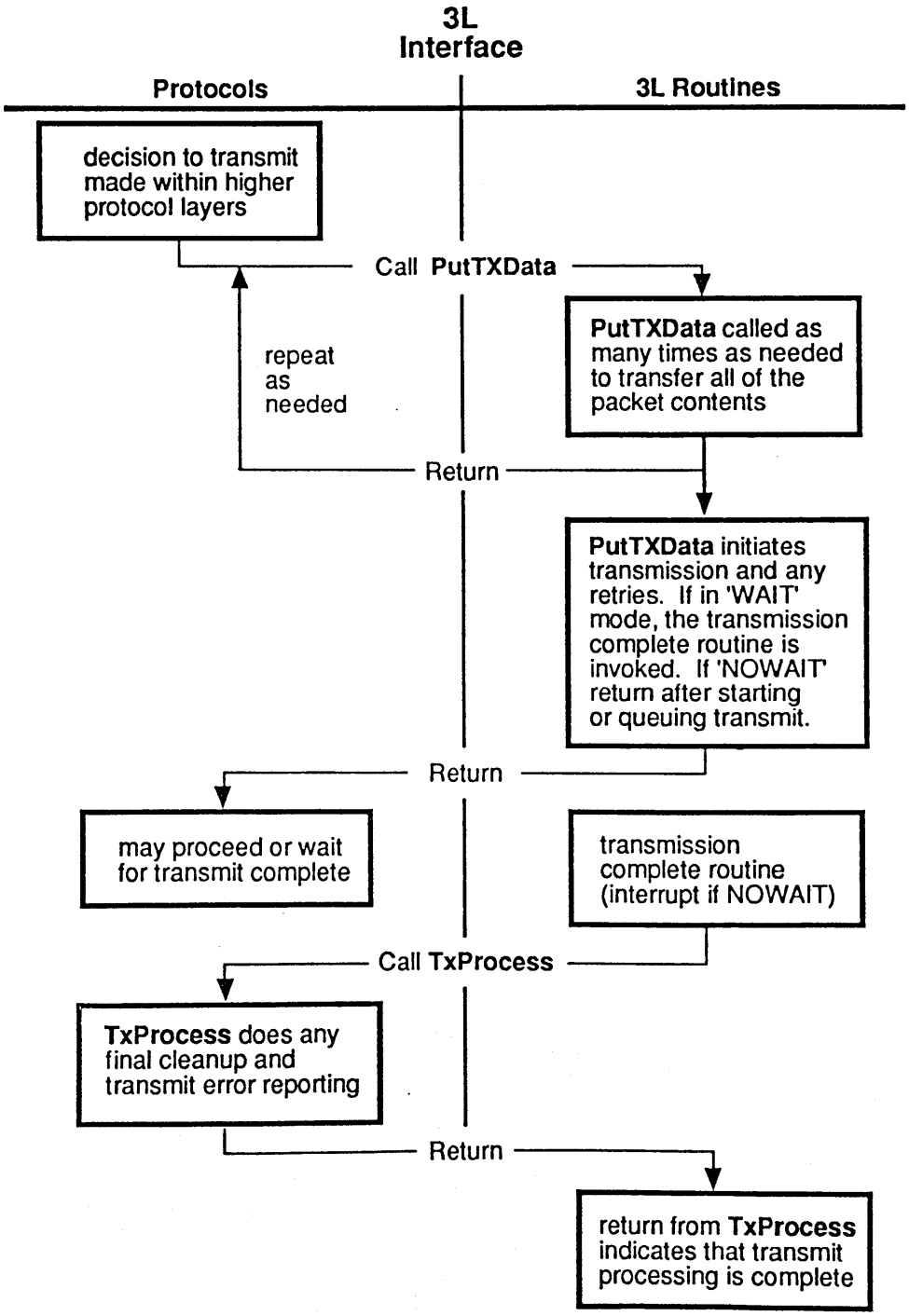

Figure 2-1. Transmit Processing

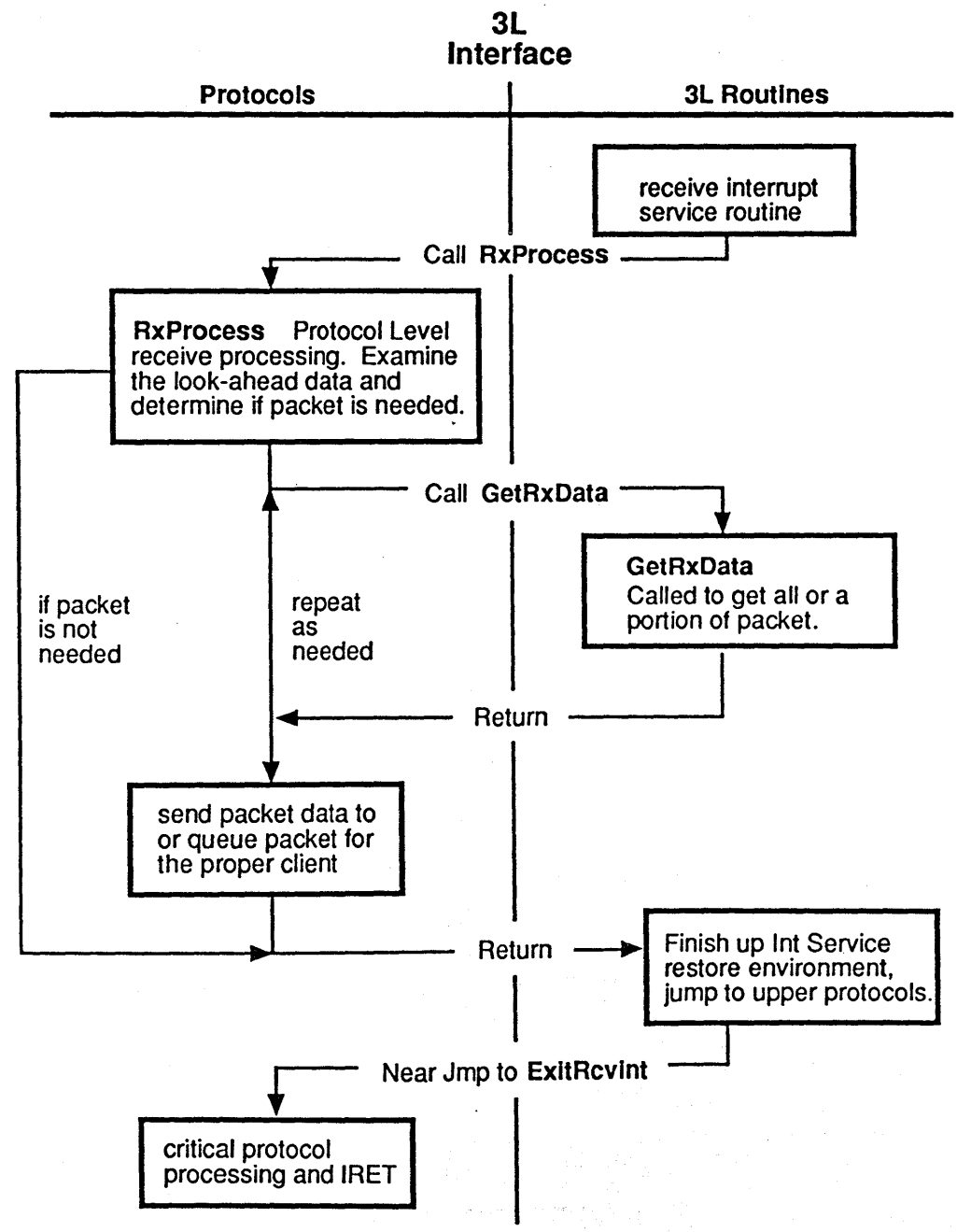

Figure 2-2. Receive Processing

Chapter 3: Interface Routine Specifications

This chapter contains two sections. The first section consists of a list of the interface routines by category. The second section contains detailed descriptions of the routines.

Listing by Category

Initialization Routines

Routine Name	Locus	Description
InitAdapters	3L	Initializes the adapter hardware into a fully configured, but not yet enabled, state.
InitParameters	3L	Sets up data structures for the 3L code with configuration information connecting the adapter with the host computer.
ResetAdapter	3L	Resets an adapter to its configured, but not yet enabled, state.
WhoAmI	3L	Returns identification and current status information about the adapter and the 3L software to the upper protocols.

Control Routines

Routine Name	Locus	Description
RdRxFilter	3L	Informs upper protocols of the types of packets.
WrRxFilter	3L	Enables/disables the adapter and specifies thepacket type.

Receive Packet Routines

Routine Name	Locus	Description
ExitRcvInt	protocol	Accomplishes final protocol side time-critical processing and performs the IRET that exits the interrupt context. NOTE: **ExitRcvInt** is defined as a near label, rather than as a PROC.
GetRxData	3L	Allows data from the received packet to be read by the protocols. Also allows buffers to be released for use in storing packets upon reception.
RxProcess	protocol	Posts the receipt of the packet or accomplishes the receive processing at the protocol level itself.
SetLookAhead	3L	Specifies an initial portion of all packet headers that are to be presented to the protocol side when the call to **RxProcess** is made from the 3L receive handler.

Transmit Packet Routines

Routine Name	Locus	Description
PutTxData	3L	Transfers data to be transmitted from the protocol.
TxProcess	protocol	Posts the completion of a transmission or accomplishes the transmit processing at the protocol level itself.

Routine Descriptions

Note Regarding General Use of Registers
In the following routine descriptions, registers DI, SI, BP, and DS are presumed to be preserved by each function, unless otherwise noted. Contents of other registers are not guaranteed, except as specifically noted. Many routines require DL=0. This was intended as the adapter number but only one adapter (adapter 0) is supported in this version. Each routine specified here is to be coded as a near procedure (PROC NEAR routine name).

InitAdapters - Initialize Network Adapters

Procedure type:	Near.
Locus:	Hardware.
Call with:	DI = Offset address of the **RxProcess** routine within CS.
Return with:	AX = Completion code. CX = Number of adapters of this type found.

This function initializes the adapter hardware, runs any diagnostics on the adapter hardware, initializes any interrupt vector(s) to be used by the adapter(s), sets up any DMA channel(s) to be used by the adapter(s), reads the network address of the board, writes the network address to the controller (as appropriate), initializes the transmit data counter to zero, and initializes the receive packet filter to no packets.

At this point, the adapter is fully initialized and prepared for use, but has its receiver disabled. See the routines **RdRxFilter** and **WrRxFilter** for information about setting the packet filter to enable and disable reception on the adapters.

InitParameters - Initialize Parameters

Procedure type:	Near.
Locus:	Hardware.
Call with:	ES:BX= Address of the DOS device driver INIT call request header.
Return with:	AX = Completion code. ES:BX = Unchanged.

This is the lowest level init function and should be called first during initialization. This function sets up internal data structures that are needed to interact with the adapter. This involves defining values specific to this board and this host computer environment, and may include interrupt channel(s) to use, DMA channel(s), DMA modes for input and for output, type of host, I/O memory addresses, etc. Also, various special hardware mode condition variables may be set.

This routine may also perform certain adapter checking and initialization functions. The parameter in ES:BX is the request header provided by DOS. The command line parameters from the line in CONFIG.SYS are at offset 18 decimal from ES:BX. If routines are not being used in a DOS device driver, a DOS INIT header must be simulated.

ResetAdapter - Reset Adapter

Procedure type:	Near.
Locus:	Hardware.
Call with:	DL = 0
Return with:	AX = Completion code.

This function resets and reinitializes the adapter. The routine is used to get an adapter back into a known good state after a catastrophic error. The state of the adapter is the same as it was after a call to **InitAdapters**. No other adapters are affected.

WhoAmI - Who Am I Hardware Self Identification/Status Report

Procedure type:	Near.
Locus:	Hardware.
Call with:	DL = 0
Return with:	AX = Completion code. ES:DI = Pointer to N byte structure holding the interesting information.

Offset	Description	Size
[ptr + 0]	Adapter Network Address/ Extended Structure Indicator.	6 bytes
[ptr + 6]	Software major version number.	1 byte BCD
[ptr + 7]	Software minor version number.	1 byte BCD
[ptr + 8]	Software sub version.	1 byte ASCII char
[ptr + 9]	Software type d or s (development or ship).	1 byte ASCII char
[ptr + 10]	Adapter type. Bits 0-5: adapter type ID (0 - 63) 1 = EtherLink (IE1) 2 = EtherLink (IE1, IE2, or IE4) 3 = EtherLink Plus 4 = EtherLink/MC 5 = TokenLink Plus 6 = EtherLink II 7 = TokenLink 8 = Reserved 9 = Reserved 10 = IBM Token Ring 11 = 3Station, 3Station/2E 12 - 63 = reserved Bit 6: hardware interface code location 0 = host processor based 1 = adapter processor based Bit 7: network type - not used in extended structure form 0 = Ethernet 1 = Token Ring	1 byte
[ptr + 11]	Initial status of hardware.	1 byte
[ptr + 12]	Reserved	1 byte
[ptr + 13]	Number of transmit buffers.	1 byte
[ptr + 14]	Size of transmit buffers.	1 word
[ptr + 16]	Number of transmissions since reset.	2 words

Offset	Description	Size
[ptr + 20]	Number of transmit errors since reset.	2 words
[ptr + 24]	Number of transmit timeouts since reset.	2 words
[ptr + 28]	Number of receptions since reset.	2 words
[ptr + 32]	Number of broadcasts received since reset.	2 words
[ptr + 36]	Number of receive errors since reset.	2 words
[ptr + 40]	Number of retries since reset.	2 words
[ptr + 44]	Data transfer mode control. Bit 0 = 0 gather pieces must start on bytes. Bit 0 = 1 gather pieces must start on word address. Bit 1 = 0 gather piece length is any number of bytes. Bit 1 = 1 gather piece length is even number of bytes.	1 byte
[ptr + 45]	WAIT/ NOWAIT transmission capability. 0 = only WAIT option is available in calls to **PutTxData.** 1 = both WAIT and NOWAIT options are available.	
[ptr + 46]	Start of hardware specific data.	

This function returns information about the current state of the hardware and software for a specific adapter.

The routine returns an address of a data structure that may be shared by the hardware and the protocols. Several statistics may be maintained from within the hardware portion and the protocol side may at any time read and/or reset these data items. For performance reasons, these statistics may be maintained only by versions of hardware side implementations that have been created for this purpose. (The fastest driver may not take time to count up things such as received packets.)

Note that two types of structures are defined. The standard structure contains 46 bytes of information and may be followed by hardware specific data. This structure is used for Ethernet and token ring networks. The extended structure allows for other network types. It is indicated by the Adapter Network Address field ([ptr + 0], 6 bytes in length) having all bits set: address = 255-255-255-255-255-255. In this case, location [ptr + 46] contains the 2-byte offset of an extension to this structure, allowing for network addresses of lengths other than 6, and network types other than Ethernet and token ring.

See Appendix A for further information.

RdRxFilter - Read Receive Filter

Procedure type:	Near.
Locus:	Hardware.
Call with:	DL = 0
Return with:	AX = Completion code. BX = Filter setting.
See related routines:	**WrRxFilter.**

This function retrieves the current filter setting from the LAN hardware controller for the specified adapter.

Note Regarding Packet Filter Settings:
The receive filter setting referred to in this routine is used to set the receive mode of the network adapter. There are four basic parts to the filter setting, each one represented by one bit in the filter setting word. Combinations of settings 1, 2, and 4, may be used as appropriate.

Receive no packets (receiver disabled).	0000h
Receive packets for the adapter address.	0001h
Receive multicast or group packets.	0002h
Receive broadcast packets.	0004h
Receive all packets.*	0008h

* Promiscuous mode: all physically addressed packets are received.

Specific handling of these settings may vary somewhat from implementation to implementation: i.e., not all settings may be available on any particular network adapter controller. Token ring implementations define these differently from Ethernet implementations. For token ring, certain control functions provide analogous capability regarding multicast and/or group addresses.

WrRxFilter - Write Receive Filter

Procedure type:	Near.
Locus:	Hardware.
Call with:	AX = Filter setting. DL = 0
Return with:	AX = Completion code.
See related routines:	**RdRxFilter.**

This function sets a new filter setting and enables or disables the adapter. It is the responsibility of the protocol software to keep track of the old filter (if it is needed later) by using **RdRxFilter.** If the filter setting is zero, no packets are to be received, and this function call will disable packet reception on the adapter. If the filter setting is a valid non-zero value, the receiver will be enabled, and the filter setting will be provided to hardware to accomplish the filtering, or will filter in software, per specific hardware capabilities.

Note Regarding Packet Filter Settings:
The receive filter setting referred to in this routine is used to set the receive mode of the network adapter. There are four basic parts to the filter setting, each one represented by one bit in the filter setting word. Combinations of the first three settings may be used as appropriate. (See Rd Rx Filter.)

Receive no packets (receiver disabled).	0000h
Receive packets for the adapter address.	0001h
Receive multicast or group packets.	0002h
Receive broadcast packets.	0004h
Receive all packets.*	0008h

* Promiscuous mode: all physically addressed packets are received.

Specific handling of these settings may vary somewhat from implementation to implementation, i.e., not all settings may be available on any particular network adapter controller. Token ring implementations define these differently from Ethernet implementations. For token ring, certain control functions provide analogous capability regarding multicast and/or group addresses.

ExitRcvInt - Protocol Time Critical Receive Processing

Label type:	Near.
Locus:	Protocol.
Jump to with:	Machine context exactly as established at the time the receive interrupt from the adapter occurred. (In other words, all registers, excepting CS, IP, and Flags, contain the values of the interrupted process. CS, IP, and the Flags register contents are on the interrupted process stack as in standard 80x86 interrupt context.)
Return:	Return is made to the interrupted process directly via an IRET. Does not return to the 3L routine that jumped to this label.

This label is reached via a jump made by the interrupt service routine (which exists in the 3L code). The hardware side has restored the context in which the interrupt was received, and is ready for return to it. The jump to the protocol side is made to allow the protocols to accomplish any additional time-critical processing prior to returning to the interrupted process.

Example Usage:
In the 3Com XNS environment, **ExitRcvInt** performs certain critical functions that would otherwise be done the next time the XNS process manager got control. In this case, the process manager gets control via the next timer tick, and this interface allows the initiation of its tasks immediately upon completion of the link layer receive functions, avoiding a potential average latency of 50 percent of the timer tick period.

In the absence of critical functions to be executed, this label may address a single IRET instruction.

GetRxData - Get Received Data

Procedure type:	Near.
Locus:	Hardware.
Call with:	CX = Count of bytes to transfer or size of buffer to release. DL = Adapter number and flags. Bit 0, 1 = 0 Bit 6 (Buffer Release) = 1 if buffer may be released. = 0 to retain the packet in the hardware buffer. Bit 7 = 0 DH = Request identifier ES:DI = Address of buffer to place packet data. Return with: AX = Completion code. CX = Actual count of bytes transferred. ES:DI = Preserved.
See related routine:	**RxProcess.**

This function obtains the requested amount of data from the packet indicated by the request identifier and also serves the function of releasing buffers for use by the hardware in storing packets received from the network medium. The Request Identifier is supplied when Rx Process is called.

ES:DI contains an address to which data is to be copied, with the amount to be copied specified in CX. The first call to **GetRxData** for a packet will obtain data from the start of the packet. Subsequent calls to **GetRxData** will return data that immediately follows the portion of the packet that was previously obtained. The buffer release bit operates independent of any request to copy data, and CX may be 0, indicating release only.

Buffer Release
The release bit is to be specified on or following the last call to **GetRxData** obtaining data from the given packet. Release of the buffer allows new packets coming in from the network medium to be stored in this hardware, or 3L, side buffer. Until the buffer is explicitly released via a **GetRxData** call with buffer release bit set, the buffer is considered in use.

RxProcess - Post the Received Packet for Protocol Side

Procedure type:	Near.		
Locus:	Protocol.		
Call with:	AX	=	Receive status.
	AH	=	0 success.
	AH	=	1 failure
	AL	=	Reserved for future use expanding receive status.
	CX	=	Size of received packet.
	DL	=	0
	DH	=	Request identifier.
	ES:DI	=	Address of virtual packet header (see below).
Return with:	DS, SI, DI destroyed.		
See related routines:	**SetLookAhead** **GetRxData.**		

This is a protocol-level receive packet processing routine. The routine is called by a hardware side receive routine, which is either an interrupt handler responding to a packet received interrupt or a subroutine of a polling control module which has noted a received packet. The basic receive packet processing sequence is outlined in the introduction section to this document. The main task for **RxProcess** is to obtain information about what to do with the packet, and what additional processing by protocol software is required to either complete that processing, or to enqueue the task for protocol software to complete it when it is no longer in the system interrupt context. The Request Identifier that is provided is used in subsequent calls to GetRxData.

Note Regarding Time Within Interrupt Context:
It is anticipated that overall system performance will be best enhanced by remaining in the interrupt context for a minimum period of time and, as such, protocols will generally attempt minimum functions in this routine to get the packet and set it up for later processing. When the processing of the received packet is finished, **RxProcess** simply returns to the receive interrupt service routine that called it.

Note Regarding Buffer Usage:
The return of **RxProcess** signifies to the interrupt service routine that all processing of the hardware buffer copy of the received packet is complete. Therefore, the hardware packet buffer may be freed, and the network adapter should be rearmed for reception, if needed.

Note Regarding Virtual Header:
ES:DI points to a copy of packet header data. This structure provides protocol software a "look ahead" at bytes from among the first 64 bytes of the received packet. It is provided to allow **RXProcess** greater intelligence in the selection of a buffer into which **GetRxData** will copy the packet data. The contents of the virtual header are defined by one or more calls to **SetLookAhead** prior to the receipt of the packet in question. Each such call specifies the number of bytes from among the first 64 in each packet header which **RxProcess** will have available for use in its initial processing. The virtual header itself contains a copy of the beginning of the packet, of a length equal to the largest number ever requested in a call to **SetLookAhead**, up to a maximum of 64 bytes.

Example Usage:
XNS protocols can determine whether or not it needs the packet by the IDP socket number and the packet type. These two words can be specified by a single call to **SetLookAhead**, by specifying a virtual header length which includes both of these words. In this case, the second of the two "interesting" words occurs at byte offsets 30 and 31; therefore calling **SetLookAhead** with a parameter of 32 (or greater) will guarantee that virtual headers passed to **RxProcess** will contain these two words. Thereafter, whenever a packet is received, a call to **RxProcess** is generated and ES:DI will address a string of 32 bytes at the beginning of the packet.

SetLookAhead - Set Packet Header Length

Procedure type:	Near.
Locus:	Hardware.
Call with:	CX = Number of bytes in list. DL = 0
Return with:	AX = Completion code.
See related routine:	**RxProcess.**

This function allows protocol software to specify the number of bytes within received packet headers that will be provided by the hardware side receive routine (either interrupt handler or subroutine of polling control module) for inspection by the protocol side **RxProcess** routine. This is information which the protocol routine **RxProcess** may use to "look ahead" into the packet to determine where to route the packet among its own clients. Use of this mechanism (to specify "look ahead" data that is provided to **RxProcess** in the form of the "virtual header") is to allow a performance gain in **RxProcess**. **RxProcess** is able to use look ahead data to determine which client the packet is intended for, and can make its calls to **GetRxData** using addresses of protocol buffers most appropriate for the particular packet. An example is the MINDS/XNS protocol using the packet type and IDP socket number to determine which client's buffer to have the packet transferred into, directly from the hardware buffer.

PutTxData - Put Transmission Data

Procedure type:	Near.
Locus:	Hardware.
Call with:	BX = Total length of packet (first call only). CX = Count of bytes to transfer. DL = Flags: Bit 0-1 = 0 Bit 4 NOWAIT transmit: = 1 to start transmission after this data transfer and return to caller as soon as transmit is initiated. = 0 to just transfer data. Bit 5 WAIT transmit = 1 to start transmit after this data transfer and await transmit completion before returning. = 0 to just transfer data. Bit 6 First data transfer call = 1 if first data transfer call for this packet . = 0 if subsequent data transfer call for this packet. Bit 7 = 0 DH = Request identifier (unused if first data transfer). DS:SI = Address of data to transmit. DI = Address of TxProcess post routine (first call only).
Return with:	AX = Completion code. DH = Request identifier (if first call).
See related routine:	**TxProcess.**

This function transfers a block of data that is to be transmitted. This function uses the transmit packet data counter to determine how far into the transmit packet to place the new data and adjusts the transmit packet data counter appropriately to account for the data just transferred. This ensures that subsequent calls to **PutTxData** will continue building the packet to transmit where the previous call left off. The transmit packet data counter is reset only when a packet transmit is started.

TxProcess Address
The address of the **TxProcess** routine received in register DI is used to notify the protocol side when the transmit is completed. If the appropriate flag bit is set, start transmitting the packet just set up. If the 3L is asynchronous, this call might only queue the packet for later transmission and not actually start a transmission immediately. If DI is -1 (= FFFFh), the hardware side assumes no protocol side completion post routine call is necessary.

WAIT/NOWAIT

When a transmission is initiated by a call to **PutTxData**, the caller may specify either WAIT or NOWAIT. In the case of WAIT, this hardware side routine initiates the transmission, and waits (probably via a loop while polling hardware) until indication of completion of the transmission is available from the network adapter. (Note: For Ethernet, in cases of collision, the standard 16 retries is attempted.) The return from **PutTxData** indicates final status of the transmission attempt. In the case of NOWAIT, a minimum of actions to initiate transmission or simply to queue the packet for transmission, is done in **PutTxData**, and control is returned to the caller. Actual completion of the transmission later results in notification to the protocol side via a call to the **TxProcess** routine address (if one has been provided). Use of NOWAIT and of the **TxProcess** interface are complementary, but independent, since either can be done without using the other.

TxProcess - Transmit Complete Processor

Procedure type:	Near.
Locus:	Protocol.
Call with:	AX = Transmit status: AH = 0 success. AH = 1 failure. AL = Reserved. DL = 0 DH = Request identifier.
Return with:	AX, BX, CX, DX, ES destroyed. BP, SI, DI, DS preserved.
See related routine:	**PutTxData.**

This function is a completion processing routine existing on the protocol side of the interface. It is called when hardware has completed a transmission initiated by a call to **PutTxData.** The address of this routine was passed by the protocol side when the call to **PutTxData** was made. Thus, several distinct **TxProcess** routines may exist in the protocol software for different control requests or for other distinct purposes. The hardware side may queue transmission requests, distinguishing them via request identifiers.

This is the protocol-level transmit complete processing routine and is called by the transmit packet complete interrupt service routine. **TxProcess** does any final cleanup necessary to inform the upper protocol levels that the requested transmission has finished. The protocol-level drivers should be able to handle a call to **TxProcess** from the hardware level at any time immediately following the call to **PutTxData.** In fact, a synchronous transmit routine calls **TxProcess** before the hardware level driver returns from **PutTxData.** In this case, request identifiers will not be required or be in use, as only a single transmission may be in progress at any time in such a synchronous driver.

Chapter 4: Error Handling

Error Handling Notes

A. All of the hardware dependent routines defined here return a completion or error code in register AX upon exiting. The value of this return code signifies that either the routine completed its appointed duties successfully, or that a problem of some sort was encountered.

The method used here to assign completion codes provides a simple means of checking for and processing errors on three different levels of detail:

- The code may be checked simply for successful or unsuccessful completion of the request.
- The code may be checked to determine the general type of error encountered, such as a bad parameter or transmission failure.
- The code may be checked to determine the specific error encountered, such as an expired retry counter.

B. The method used to provide this flexibility in error handling is simple. The total range of completion codes is first split into several subranges. Each subrange is then associated with a specific category of error. Finally, individual codes within each subrange or category of codes are assigned to specific error conditions, as required.

C. By using this method, and by choosing an error subrange size of 256, the three types of error checking discussed above may be implemented by the following methods:

- The simple success/failure test is done by testing AX for a zero or nonzero value.
- The general error category test is done by testing the value in AH. Since an error subrange size of 256 was chosen, each subrange will provide a unique value in AH. For example, an incorrect parameter passed to a routine would return a completion code in the 100h-1FFh range, giving a value of 01h in register AH.
- The specific error test is done by comparing the value in AX to a specific completion code. If the test for the general error category has already been done, only the value in register AL need be tested.

- Since the completion code is returned in a 16-bit register, the total number of completion codes is rather large (65000+), allowing a generous initial allocation of completion code subranges. The allocation of completion codes and categories is defined in the "Completion Code Assignment" section of this document.

D. For consistency and convenience, the first completion code in each range is defined as a general failure of the given category. In this way, a routine need not allocate specific completion codes for generic errors.

Completion Code Assignment

The allocation of completion code ranges to error categories follows:

Code range	Error category
0	No error
0100h-01FFh	Parameter errors
0200h-02FFh	Miscellaneous host errors
0300h-03FFh	Adapter initialization errors
0400h-04FFh	Adapter transmission errors
0500h-05FFh	Adapter receive errors
0600h-06FFh	Miscellaneous adapter errors
0700h-07FFh	Reserved
0800h-08FFh	Reserved

Specific completion code definitions follow:

No error encountered

0000h	No error. Request completed successfully.

Parameter errors

0100h	General parameter error
0101h	Reserved
0102h	Invalid data transfer method

Miscellaneous host errors

0200h	General host error

Adapter initialization errors

0300h	General adapter initialization error
0301h	Problem while locating adapters
0302h	Unable to initialize adapter
0303h	Adapter diagnostic failure
0304h	Unable to read network address from adapter
0305h	Unable to write network address to adapter
0306h	Invalid network address
0307h	Invalid receive filter setting
0308h	Unable to read receive filter setting

Adapter transmission errors

0400h	General adapter transmission error
0401h	All transmit buffers in use, try again later
0402h	Maximum packet length exceeded
0403h	Unable to load packet data to adapter
0404h	Unable to start transmission
0405h	Transmission retry count exceeded

Adapter receive errors

0500h	General adapter receive error
0502h	Receive interrupts already enabled
0503h	Unable to disable receive interrupts
0504h	Receive interrupts already disabled
0505h	End of packet data reached
0506h	Unable to read packet information
0507h	Unable to read packet data
0508h	Unable to rearm adapter for reception

Miscellaneous adapter errors

0600h	General adapter error
0601h	Shared buffers not available
0603h	Command rejected
0604h	Command timed out with no response
0605h	Unable to initiate data transfer
0606h	Unable to complete data transfer

Appendix A: Hardware Implementation Specifics

EtherLink (3C501)

Package Contents

The package for this adapter consists of the source code and a "lib'd" version of the 3L library. The source files are:

501CTRL.ASM	- **RdRxFilter** and **WrRxFilter**
501DATA.ASM	- Data declarations
501INIT.ASM	- **ResetAdapter, WhoAmI, InitAdapters** and **InitParameters**
501INTR.ASM	- Interrupt service routine
501RECV.ASM	- **GetRxData**
501UTIL.ASM	- Various hardware utility routines
501XMIT.ASM	- **PutTxData**
501VER.ASM	- Version string data declaration

Transmission Capabilities

Only WAIT mode transmits are supported.

Initialization Parameters

InitParameters interprets the DOS device driver INIT request header pointed to by ES:BX. At offset 18 from the start of this structure is a pointer to an ASCII string which is the parameter list after the 'device=' statement in the CONFIG.SYS file. The parameter list must have the following format:

<filename> <i> <bbb> <d> <t>

where:

<filename>	is the name of the device driver file
<i>	is the interrupt level
<bbb>	is the I/O base address (in hex)
<d>	is the DMA channel
<t>	is the DMA transfer mode: t = 1 DMA single byte mode t = 2 Programmed I/O loop t = 3 DMA block or demand mode as supported by the adapter t = 4 Programmed I/O "rep"

All of the parameters are position dependent.

WhoAmI Statistics

No statistics in the **WhoAmI** data structure are maintained.

Other

The only valid **RxFilter** settings are:

0	Receive no packets
5	Receive station address and broadcast packets
7	Receive station address, multicast and broadcast packets
8	Receive all packets (promiscious mode)

EtherLink II (3C503)

Package Contents

The package for this adapter consists of the source code and a "lib'd" version of the 3L library. The source files are:

EHWINIT.ASM	- **InitParameters**
EHWRECV.ASM	- **GetRxData** and the interrupt service routine
EHWXFER.ASM	- **SetLookAhead**
EHWRTN.ASM	- **ResetAdapter, RdRxFilter, WrRxFilter, WhoAmI,** and **InitAdapters**
EHWDATA.ASM	- Data declarations
EHWXMIT.ASM	- **PutTxData**
TIMER.ASM	- Optional timer routines

Transmission Capabilities

Both WAIT and NOWAIT transmissions are supported.

Initialization Parameters

InitParameters interprets the DOS device driver INIT request header pointed to by ES:BX. At offset 18 from the start of this structure is a pointer to an ASCII string which contains the driver name, optionally followed by one or more of the following parameters:

/I:x	Use interrupt level 'x'
/A:xxx	Use 'xxx' for the I/O base address of the adapter, where 'xxx' is a three-digit hex number.
/D:x	Use DMA channel 'x'
/T:x	Use transceiver 'x': x = 1 BNC/Onboard x = 2 DIX/External
/M:x	Use data transfer mode 'x': x = 1 DMA single byte mode x = 2 Programmed I/O loop x = 3 DMA block or demand mode as supported by the adapter x = 4 Programmed I/O "rep" (default for 286-based machines)

WhoAmI Statistics

Four of the statistics counters in the **WhoAmI** data structure are implemented. The other statistics counters have not been implemented for performance reasons and will remain at zero. The counters which are kept are:

[ptr+16] Number of transmissions
[ptr+24] Number of transmit timeouts
[ptr+28] Number of receptions
[ptr+36] Number of receive errors

These statistics are not cleared during a reset. The hardware specific portion of the **WhoAmI** structure is defined as follows:

[ptr+46]	Number of jams	2 words
[ptr+50]	Number of receive overflows	2 words
[ptr+54]	Number of short packets (runts) received	2 words

Although all three of these counters are defined, only the third one (Number of short packets received) is actually implemented.

EtherLink/MC (3C523)

Package Contents

The package for this adapter consists of the source code and a "lib'd" version of the 3L library. The source files are:

523CTRL.ASM	- **RdRxFilter** and **WrRxFilter**
523UTIL.ASM	- Various hardware dependent routines
523DATA.ASM	- Data declarations
523INIT.ASM	- **ResetAdapter, WhoAmI, InitAdapters** and **InitParameters**
523INTR.ASM	- Interrupt service routine
523RECV.ASM	- **GetRxData** and **SetLookAhead**
523XMIT.ASM	- **PutTxData**

Transmission Capabilities

Only WAIT mode transmits are supported.

Initialization Parameters

There are no initialization parameters.

WhoAmI Statistics

Statistics are kept in the **WhoAmI** data structure if the DEV compiler flag is turned on.

Other

The only valid **RxFilter** settings are:

0	Receive no packets
1	Receive station address packets
3	Receive station address and multicast packets
5	Receive station address and broadcast packets
7	Receive station address, multicast and broadcast packets
8	Receive all packets (promiscious mode)

TokenLink (3C603)

Package Contents

The package for this adapter consists of the source code and a "lib'd" version of the 3L library. The source files are:

TOKENS.ASM	- All 3L source code
TIMER.ASM	- Optional timer interface routines

Transmission Capabilities

Both WAIT and NOWAIT transmissions are supported. If more than eight NOWAIT transmits are queued up, awaiting transmission, **PutTxData** will return an error indicating that no more buffers are available.

Initialization Parameters

InitParameters interprets the DOS device driver INIT request header pointed to by ES:BX. At offset 18 from the start of this structure is a pointer to an ASCII string which may be composed of the following position dependent parameters:

<filename>.sys <i> <bbb> <d> <w>

where:

<filename>.sys	is the name of the device driver. This must end with the letters "SYS".
<i>	is the interrupt level
<bbb>	is the I/O base address
<d>	is the DMA channel
<w>	is the number of wait states

WhoAmI Statistics

The **WhoAmI** data structure is only partially implemented. The network address, transmit counts and receive counts are the only fields that are maintained in the entire structure.

Other

The only valid **RxFilter** settings are:

0	No packet are received
5	Receive station address and broadcast packets

3Station (3C1100)

Package Contents

The package for this adapter consists of the source code and a "lib'd" version of the 3L library. The source files are:

1100CTRL.ASM	- **RdRxFilter, WrRxFilter** and **SetControl**
1100DATA.ASM	- Data and structure declarations
1100INIT.ASM	- **InitAdapter, ResetAdapter, WhoAmI** and **InitParameters**
1100INTR.ASM	- Receive interrupt service routine
1100RECV.ASM	- **GetRxData** and **SetLookAhead**
1100UTIL.ASM	- Utility routine used by other modules
1100VER.ASM	- Version number string
1100XMIT.ASM	- **PutTxData**

Transmission Capabilities

Only WAIT mode transmits are supported. If NOWAIT request are made to **PutTxData**, they will be handled as WAIT mode transmits. Due to the nature of the hardware, when the receiver is enabled/disabled the transmission is also enabled/disabled.

Initialization Parameters

InitParameters receives in ES:BX the address of the DOS device driver INIT call request header, but does not use any run-time parameters. The routine does verify that the code is running on a 3Station. It also checks the hardware version number of the machine. The hardware version number is used to differentiate between the 3Station version released in April 1987 and future versions. There are hardware bugs in the 3Station version released in April which should not be present in future versions of the 3Station. The variable "lite_flag" is set to 1 to allow the "xfer" routine (see 1100util.asm) to take advantage of the improved performance possible due to the hardware fixes.

WhoAmI Statistics

No statistics are maintained in the **WhoAmI** data structure.

Other

A special note is needed here to explain how the selection of the internal/external transceiver in the hardware is made. To boot a 3Station, the PROM code runs the hardware up until the device driver version takes control. Since the PROM code has made the selection in order to make the hardware work (in order to boot), the code in "InitGA" (part of **InitAdapters**) does not alter the current setting of the transceiver and it does not reset the gate array since this also resets the transceiver selection.

Appendix B: Implementation Syntax and Naming

3Com currently implements both hardware and protocol side routines in 8086/80286 assembler source compatible with the Microsoft macro assembler, version 4.0. Hardware and protocol routines are maintained in separate modules, and linked with Microsoft's LINK, version 3.05.

Define each hardware side procedure as a public near procedure in source files where procedure code is included, and declare each as external near procedures in your source files where calls are made.

Implementation as a configurable DOS device driver allows specification of an end address for the code to DOS at the time configuration initialization completes. In the current MINDS implementations, this driver is run at configuration initialization time, and then cut off by specifying a code end address to DOS which is just prior to this "use one time and throw away" type code. In this MINDS implementation, the module ethdr.obj must be linked first (as it contains the DOS device driver header). Three SEGMENTS are defined to the linker, and are all grouped together so that all data and entry points are relative to the same start location for the driver.

The statements to declare statement in a module are:

```
        ;align all 3 segments
CODE GROUP RCODE, DATA, ICODE
        ;specify default label
        ;and variable registers
ASSUME cs:CODE,ds:CODE,ss:NOTHING, es:NOTHING

RCODE SEGMENT WORD PUBLIC
        ;DOS device driver header structure
        ;all executable code which is to be retained after configuration initialization time
RCODE ENDS

DATA SEGMENT WORD PUBLIC
        ;define data items which are to be retained
DATA ENDS

ICODE SEGMENT WORD PUBLIC
near label CUTOFFPOINT:  ;address to pass DOS for throwaway after init
        ;all executable code which is top be discarded after configuration initialization time
        ;all data items which are to be inserted
ICODE ENDS
```

Appendix C: Interfacing to a 3L Compliant Driver Example

This chapter contains listings providing example use of the 3L Interface. These listings are for illustration only and are not intended for direct inclusion in a developer's program.

This module, whose source file is named CTO3L.ASM, consists of a set of assembly language interface modules used to provide linkage between C language code and the assembly language-based Link Level Libraries. It is expected that in most applications, the 3L routines will be linked directly into installable MS-DOS® drivers.

```
title cto3l.asm

;*****************************************************************
;
;File: CTO3L.ASM
;
;Description:  This file contains subroutines which provide a
;              C program with an interface to the 3L 1.0 routines.
;
;*****************************************************************

PUBLIC  _getds

PUBLIC  _cInitParameters
PUBLIC  _cInitAdapters
PUBLIC  _cResetAdapter
PUBLIC  _cWhoAmI
PUBLIC  _cRdRxFilter
PUBLIC  _cWrRxFilter
PUBLIC  _cPutTxData
PUBLIC  _cGetRxData
PUBLIC  _cSetLookAhead
PUBLIC  _etext

extrn   _myExitRcvInt   :near
extrn   _myRxProcess    :near
extrn   _myTxProcess    :near


PUBLIC  ExitRcvInt
PUBLIC  RxProcess
PUBLIC  TxProcess


extrn   InitParameters  :near
extrn   InitAdapters    :near
extrn   ResetAdapter    :near
extrn   WhoAmI          :near
extrn   RdRxFilter      :near
extrn   WrRxFilter      :near
extrn   PutTxData       :near
extrn   GetRxData       :near
extrn   SetLookAhead    :near


CODE    GROUP   _TEXT, DATA, ICODE

_TEXT   segment byte public 'CODE'
DGROUP  group   _DATA, _BSS
        assume  cs:_TEXT, ds:DGROUP, ss:DGROUP
_TEXT   ends

DATA    segment word public 'CODE'
DATA    ends

ICODE   segment word public 'CODE'
ICODE   ends
```

```
DATA    segment
his_ds  dw      ?
_etext  db      ?
DATA    ends

_DATA   segment word public 'DATA'
_d@     label   byte
_DATA   ends
_BSS    segment word public 'BSS'
_b@     label   byte
_BSS    ends
_DATA   segment word public 'DATA'
_s@     label   byte
_DATA   ends


_TEXT   SEGMENT
        ASSUME  CS:_TEXT, DS:DGROUP, SS:DGROUP


_getds  proc    near
        mov     ax,ds
        mov     cs:his_ds,ax
        ret
_getds  endp


;-------------------------------------------------------------------------
;
;   _cInitAdapters:  This procedure provides the glue between a C
;                    program and the 3L 1.0 InitAdapters function.
;
;   Calling Sequence:
;      int cInitAdapters( &nAdapters )
;
;   Input Parameters:
;      None
;
;   Output Parameters:
;      int  nAdapters
;
;   Returns:
;      The return value of the InitAdapters function
;
;-------------------------------------------------------------------------

_cInitAdapters  proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     ax,cs
        mov     ds,ax
```

```
        mov     di,offset CODE:RxProcess

        call    InitAdapters

        pop     ds
        mov     di,word ptr[bp+4]
        mov     word ptr[di],cx

        pop     di
        pop     si
        pop     bp
        ret
_cInitAdapters  endp


;-------------------------------------------------------------------------
;
;   _cInitParameters:  This procedure provides the glue between a C
;                      program and the 3L 1.0 InitParameters function.
;
;   Calling Sequence:
;      int cInitParameters( Parms )
;
;   Input Parameters:
;    char *Parms - Pointer to a structure with overrides of default
;                  parameters.
;
;   Output Parameters:
;      None
;
;   Returns:
;      The return value of the InitParameters function
;
;-------------------------------------------------------------------------

_cInitParameters        proc near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     bx,[bp+4]

        mov     ax,ds
        mov     es,ax
        mov     ax,cs
        mov     ds,ax
```

```
        call    InitParameters

        pop     ds
        pop     di
        pop     si
        pop     bp

        ret
_cInitParameters        endp


;-------------------------------------------------------------------------
;
;   _cResetAdapter:  This procedure provides the glue between a C
;                    program and the 3L 1.0 ResetAdapter function.
;
;   Calling Sequence:
;      int cResetAdapter( )
;
;   Input Parameters:
;      None
;
;   Output Parameters:
;      None
;
;   Returns:
;      The return value of the ResetAdapter function
;
;-------------------------------------------------------------------------

_cResetAdapter  proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     dx,0
        mov     ax,cs
        mov     ds,ax

        call    ResetAdapter

        pop     ds
        pop     di
        pop     si
        pop     bp
        ret
_cResetAdapter  endp
```

```
;-------------------------------------------------------------------------
;
;    _cWhoAmI:  This procedure provides the glue between a C
;               program and the 3L 1.0 WhoAmI function.
;
;    Calling Sequence:
;       int cWhoAmI( &WhoPtr )
;
;    Input Parameters:
;       None
;
;    Output Parameters:
;       struct WhoStruct far *WhoPtr - Far pointer to the WhoAmI
;       structure.
;
;    Returns:
;       The return value of the WhoAmI function
;
;-------------------------------------------------------------------------

_cWhoAmI                proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     dx,0
        mov     ax,cs
        mov     ds,ax

        call    WhoAmI

        pop     ds
        mov     si,[bp+4]
        mov     word ptr [si],di
        mov     word ptr [si+2],es

        pop     di
        pop     si
        pop     bp

        ret
_cWhoAmI                endp
```

```
;-------------------------------------------------------------------------
;
;    _cRdRxFilter:  This procedure provides the glue between a C
;                   program and the 3L 1.0 RdRxFilter function.
;
;    Calling Sequence:
;       int cRdRxFilter( &RxFilter )
;
;    Input Parameters:
;       None
;
;    Output Parameters:
;       int  RxFilter - The receive filter value
;
;    Returns:
;       The return value of the RdRxFilter function
;
;-------------------------------------------------------------------------

_cRdRxFilter    proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     ax,cs
        mov     ds,ax

        mov     dx,0
        call    RdRxFilter

        pop     ds
        mov     di,[bp+4]
        mov     [di],bx

        pop     di
        pop     si
        pop     bp
        ret
_cRdRxFilter    endp
```

```
;-------------------------------------------------------------------------
;
;    _cWrRxFilter:  This procedure provides the glue between a C
;                   program and the 3L 1.0 WrRxFilter function.
;
;    Calling Sequence:
;       int cWrRxFilter( RxFilter )
;
;    Input Parameters:
;       int  RxFilter - The new receive filter value
;
;    Output Parameters:
;       None
;
;    Returns:
;       The return value of the WrRxFilter function
;
;-------------------------------------------------------------------------

_cWrRxFilter    proc    near
        push    bp
        mov     bp,sp
        push    ds
        push    si
        push    di

        mov     ax,cs
        mov     ds,ax

        mov     dx,0
        mov     ax,[bp+4]
        call    WrRxFilter

        pop     di
        pop     si
        pop     ds
        pop     bp
        ret
_cWrRxFilter    endp
```

```
;-------------------------------------------------------------------
;
;    _cSetLookAhead:  This procedure provides the glue between a C
;                     program and the 3L 1.0 SetLookAhead function.
;
;    Calling Sequence:
;       int cSetLookAhead( NumBytes )
;
;    Input Parameters:
;       int NumBytes - The number of bytes of look ahead data
;
;    Output Parameters:
;       None
;
;    Returns:
;       The return value of the SetLookAhead function
;
;-------------------------------------------------------------------

_cSetLookAhead  proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
 push    ds

 mov    ax,cs
        mov     ds,ax
         -
        mov     dx,0
        mov     cx,[bp+4]
        call    SetLookAhead

        pop     ds
        pop     di
        pop     si
        pop     bp
        ret
_cSetLookAhead  endp
```

```
;-------------------------------------------------------------------
;
;    _cPutTxData:  This procedure provides the glue between a C
;                  program and the 3L 1.0 PutTxData function.
;
;    Calling Sequence:
;       int cPutTxData( TotalPacketLen, NumBytes, Flags, RequestID,
;                       PacketAddr, &NewRequestID )
;
;    Input Parameters:
;     int TotalPacketLen - The total packet length (first call only)
;       int NumBytes - The number of bytes to transfer this call
;       int Flags - The DL flags
;       int RequestID - Used if not the first call
;       char far *PacketAddr - A far pointer to the packet
;
;    Output Parameters:
;       int NewRequestID - Returned after first call
;
;    Returns:
;       The return value of the PutTxData function
;
;-------------------------------------------------------------------

_cPutTxData     proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     ax,ds
        mov     es,ax

        mov     bx,[bp+4]
        mov     cx,[bp+6]
        mov     dl,byte ptr[bp+8]
        mov     dh,byte ptr[bp+10]
        mov     si,[bp+12]
        mov     di,offset CODE:TxProcess
        call    PutTxData

        pop     ds
        xchg    dh,dl
        xor     dh,dh
        mov     di,[bp+16]
        mov     [di],dx

        pop     di
        pop     si
        pop     bp
        ret
_cPutTxData     endp
```

```
;-------------------------------------------------------------------
;
;    _cGetRxData: This procedure provides the glue between a C
;                 program and the 3L 1.0 GetRxData function.
;
;    Calling Sequence:
;       int cGetRxData( &NumBytes, Flags, RequestID, PacketAddr )
;
;    Input Parameters:
;       int NumBytes - The number of bytes to transfer this call
;       int Flags - The DL flags
;       int RequestID - The request identifier
;       char far *PacketAddr - A far pointer to the packet to copy
;                              the data
;
;    Output Parameters:
;       int Numbytes - The actual number of bytes transferred
;
;    Returns:
;       The return value of the GetRxData function
;
;-------------------------------------------------------------------

_cGetRxData     proc    near
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        mov     di,[bp+4]
        mov     cx,ss:[di]
        mov     dl,byte ptr[bp+6]
        mov     dh,byte ptr[bp+8]
        mov     di,[bp+10]
        mov     es,[bp+12]
        call    GetRxData

        pop     ds
        mov     di,[bp+4]
        mov     ss:[di],cx

        pop     di
        pop     si
        pop     bp
        ret
_cGetRxData     endp
```

```
;-------------------------------------------------------------------
;
;    TxProcess: This procedure is the protocol-side routine
;               which is called when a packet has finished
;               transmitting (see _cPutTxData). It provides the
;               glue between the 3L 1.0 routines and a C routine
;               called myTxProcess.
;
;    myTxProcess Calling Sequence:
;       void myTxProcess( Status, RequestID )
;
;    myTxProcess Input Parameters:
;       int Status - Transmit status
;       int RequestID - The request identifier
;
;    myTxProcess Returns:
;       Nothing
;
;-------------------------------------------------------------------

TxProcess       proc    near
        push    bp
        push    si
        push    di
        push    ds
        push    es

        push    ax
        mov     ax,cs:his_ds
        mov     ds,ax
        mov     es,ax
        pop     ax

        xor     cx,cx
        mov     cl,dh
        xor     dh,dh

        push    cx
        push    ax
        call    _myTxProcess

        add     sp,4

        pop     es
        pop     ds
        pop     di
        pop     si
        pop     bp
        ret
TxProcess       endp
```

```
;-------------------------------------------------------------------
;
;    ExitRcvInt: This procedure is the protocol-side routine
;                which is called when the 3L has completed a
;                receive interrupt. It provides the
;                glue between the 3L 1.0 routines and a C routine
;                called myExitRcvInt.
;
;    myExitRcvInt Calling Sequence:
;       void myExitRcvInt( )
;
;    myExitRcvInt Input Parameters:
;       None
;
;    myExitRcvInt Returns:
;       Nothing
;
;-------------------------------------------------------------------

ExitRcvInt:
        push    bp
        push    ds
        push    es
        push    si
        push    di

        push    ax
        mov     ax,cs:his_ds
        mov     ds,ax
        mov     es,ax
        pop     ax

        call    _myExitRcvInt

        pop     di
        pop     si
        pop     es
        pop     ds
        pop     bp
        iret
```

```
;-------------------------------------------------------------------------
;
;   RxProcess: This procedure is the protocol-side routine
;              which is called when a packet has been received
;              (see _cInitAdapters). It provides the
;              glue between the 3L 1.0 routines and a C routine
;              called myRxProcess.
;
;   myRxProcess Calling Sequence:
;     void myRxProcess( Status, PacketSize, RequestID, PacketHeader
;
;   myRxProcess Input Parameters:
;      int Status - Receive status
;      int PacketSize - Size of the received packet
;      int RequestID - The request identifier
;      char far *PacketHeader - Address of the virtual packet
;                               header
;
;   myRxProcess Returns:
;      Nothing
;
;-------------------------------------------------------------------------

RxProcess       proc    near
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es
        pushf

        push    es    ; push FAR ptr to Virtual pckt hdr
        push    di

        push    ax
        mov     ax,cs:his_ds
        mov     ds,ax
        mov     es,ax
        pop     ax

        xor     bx,bx
        mov     bl,dh
        xor     dh,dh

        push    bx
        push    cx
        push    ax

        call    _myRxProcess
        add     sp,10
```

```
        popf
        pop     es
        pop     ds
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx

        ret
RxProcess       endp

_TEXT   ends
        end
```