Implementation of Network Primitives for a Network Monitor/Debugger

Brendan A. Voge

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

ABSTRACT

Computer networks need specialized tools for developing, monitoring, and testing. The network primitives described here will connect a network to a microcomputer system, allowing the development of a network debugger/monitor. A network monitor connects to the network and gathers statistical data about network traffic. A network debugger is similar to a program debugger, except that the programs being debugged are distributed on a network. Together the network monitor/debugger will allow protocol designing and testing, as well as distributed software debugging. In order to determine and control the state of the network, and that of its components, the network monitor/debugger must rely on a set of primitive network operations

I designed a set of network primitives so that a network monitor/debugger can be developed using the software and hardware provided by this project. The network primitives present the user with a standard *UNIX*[1] interface to the network through the use of UNIX device file read, write and i/o control operations. The network interface hardware (an *EXOS*/101[2]) is connected to the host computer (a Z8000-based microcomputer) via network primitives patched to the host operating system (*XENIX*[3]). The network hardware is connected to a 10 Mbit *Ethernet*[4].

1 UNIX is a trademark of Bell Labs

2 EXOS is a trademark of Excelan

3 XENIX is a trademark of Microsoft.

4 Ethernet is a trademark of XEROX Corp.

CONTENTS

1 Introduction..2
1.1 Network Monitor...............................2
1.2 Network Debugger..............................2
1.3 Implementation of the Network Drivers.....3
2 Networking System Overview........................3
2.1 Networking Hardware...........................3
2.2 Networking Driver Functions...............3
2.3 Patching the Drivers to the Host..........3
2.4 Interface to the Networking Board.........4
2.5 Current Layering of the Network Software..4
2.6 Proposed Layering of the Network Software.5
3 Detailed Description of the Networking System.......6
3.1 System Hardware...............................6
3.1.1 Host System.....................6
3.1.2 Networking Board................6
3.1.3 Interrupts......................6
3.2 Memory Allocation and Usage...............7
3.2.1 Message Queues..................7
3.2.2 Data Buffers....................8
3.2.3 Allocation of Memory Space......8
3.2.4 Waiting for Buffers.............8
4 Procedures for Accessing the Network Device File....9
5 Low Level Procedures for Interfacing to the Net....10
7 Conclusion...14
6 Acknowledgements...................................14
8 References...15

1. INTRODUCTION

Computer networking has been suggested as a method for implementing high performance computers, easily extensible computer systems, highly reliable computer systems, and highly available computer systems. Sophisticated tools need to be developed to simplify the development and construction of computer networks. Among the tools that would facilitate the development of networking systems are network monitors and network debuggers. These tools were the incentive for the implementation of the network drivers described in this project.

The network primitives described here are the first step towards implementing a network monitor/debugger. The drivers interface a specific networking board (the Excelan EXOS/101) to a UNIX look-alike operating system (XENIX). The driver software presents a standard UNIX interface to the networking board by supporting the UNIX input/output device operations: read, write, and i/o control. By presenting the standard interface to the operating system, the network monitor/debugger software can be written independently of the driver software and of the networking hardware.

1.1. NETWORK MONITOR

It has been proposed that a network monitor connected to a network and recording network statistics and events would be useful in evaluating network performance. Interesting statistics to be gathered are: the number of packets sent with no errors, the number of packet transmissions aborted due to excess collisions, the number of packets received with no errors, the number of packets received with alignment errors, the number of packets received with checksum errors, and the number of packets lost due to lack of buffer space.

Another possible use of a network monitor would be to gather all packets which pass over the network, and record the message header. A record of message headers would show which protocols are most commonly used, and should be optimized. In addition, the monitor could capture messages with destination addresses which are on a gather list. The gather list would allow the monitor to focus on specific processors, or groups of processors on the network, and the message traffic between them.

The network monitor should also be able to interrogate processors connected to the network to determine the loads on those processors by running a standard benchmark task. This feature could be used to evaluate the efficiency of a load balancing scheme, or to implement load balancing.

Finally, the network monitor should be able to artificially load the network, either by sending messages to the network itself, or by having each of the other processors connected to the net send dummy messages. The network monitor could then observe the network under different load conditions, and gather data that would allow the evaluation of network performance.

1.2. NETWORK DEBUGGER

A network debugger would be valuable for developing computer networks. It has been tentatively proposed that such a debugger should allow the operator to have complete control over the network at the network operation level. That is, the debugger should be able to synchronize or reorder the transmission and reception of network messages. This implies that the network debugger should keep a global time clock to synchronize events, and that all network messages sent over the net should be routed through the network debugger so that the debugger can control the order of message transmission and reception.

The network monitor/debugger should probably reside on a dedicated processor, although this is not absolutely necessary. The other processors connected to the net would have slave debugger software which filters outgoing messages, and accepts commands sent by the network debugger. The slave debugger software should be able to start and stop processes on the host machine, and should keep a local, synchronized version of the global time. This gives the network debugger complete control over all network processors which are running the slave debugger software. Note that this raises some difficult security issues.

1.3. IMPLEMENTATION OF THE NETWORK DRIVERS

The network monitor/debugger has not been implemented, nor are the features described in the above sections anything but preliminary and tentative. The brief description of the network monitor and debugger has been given to motivate the development of the network drivers. What has been designed ard implemented are the drivers which patch the networking board to the host processor.

The drivers interface to the networking board using the standard UNIX device interface commands read, write, and i/o control. The standard interface allows the network monitor/debugger to be independent of the network hardware. The network drivers support efficient passing of any data message to the networking board. The network drivers assume nothing about message contents, so they can be used with any message or protocol.

The network drivers simplify the implementation of the network monitor/debugger. They gather useful network statistics and can perform a non-busy wait when message or data buffers are not available. The drivers also allow the networking board to gather all messages sent over the network so that a network monitor could collect all message headers for evaluation. Further, a list of up to eight addresses for message gathering can be specified so that specific network traffic can be examined.

2. NETWORKING SYSTEM OVERVIEW

2.1. NETWORKING HARDWARE

The system hardware consists of a host system and a networking board (IMP). The host is a Z8000 microcomputer with a hard disk provided by Advanced Micro Devices (AMD). The host operating system is a UNIX look-alike called XENIX written by Microsoft. The networking processor is an 8088 based system with resident software which support fundamental networking operations. The networking board (EXOS/101) was provided by Excelan.

2.2. NETWORKING FUNCTIONS SUPPORTED BY THE DRIVERS

I wrote drivers to interface the networking board to the host. The host operating system uses the EXOS board as an input/output device by reading and writing the network device file. The host can request the EXOS to perform one of the following driver functions:

-initialize the networking board
(shared memory address, queue addresses, and interrupt type)
-set the mode of the networking board
(recognized error conditions, and address filtering)
-set the network addresses of the host processor
(addresses recognized as the host (up to eight))
-enable or disable message receive for an address
(any one of the eight recognized addresses)
-send a message to the network
(transmit the message pointed to by the request message)
-receive a message from the network
(transfer to shared memory a received message)
-interrogate the networking board for network statistics
(returns network statistics gathered by the networking board)
-close the network
(end the networking session)

2.3. PATCHING THE DRIVERS TO THE HOST OPERATING SYSTEM

By allowing the operating system to use the above functions, the EXOS may be treated as just another device by the operating system. Since the operating system is a UNIX look-alike, the

user interface to the device is simple. The user just writes the Ethernet packet to the network device file by calling write(message). The driver software sends a transmit request message to the EXOS, causing the Ethernet packet to be transmitted. Similarly, to read from the network the user reads from the network device file by calling read(buffer). The driver software sends a receive message to the EXOS asking for a pending message. The Ethernet packet is returned in the system buffers provided when read was called. The host interrupts the EXOS to indicate that a new request message has been sent. Note that lost packets are not automatically retransmitted, thus only datagrams are provided.

A reply message from the EXOS interrupts the host. The host interprets the reply message, places any packets to be received into the system buffers provided, and then returns control to the user's program. The host processor then interrupts the EXOS to indicate that the reply buffer has been received and may now be reused.

The user accesses the network device file by using the system calls read and write. These calls transfer data out of (or into) the device file into (or out of) system buffers. The control of the EXOS for tasks such as changing address slots or gathering statistics uses the catch-all system call to i/o control. These three system calls (read, write, and iocntrl) constitute the standard UNIX interface to any device.

2.4. THE INTERFACE TO THE NETWORKING BOARD

The interface between the host system and the networking board (EXOS) uses interrupts and shared memory. When either processor (host or networking board) has a message to transfer (request, reply, or actual Ethernet message), it interrupts the other indicating there is a new request message in the shared memory. The interrupted processor examines the shared memory until the request message is found. If the request message indicates that an Ethernet message is to be transferred, then the request message also contains pointers to the Ethernet message, which is also stored in shared memory. For example, to transfer a message from the EXOS to the host, once the request message has been processed, the networking processor frees the area of shared memory containing the request message by changing an ownership bit in shared memory, and then interrupts the host to indicate that another message buffer is available.

The processors use circular queues to communicate through shared memory. There are two such queues, one for host to EXOS request transfers, and one for EXOS to host reply transfers. The queues are initialized by the host before the EXOS is initialized. The host then indicates to the EXOS where the two queues are located by two address fields in the initialization message.

The shared memory is physically located in the host system. The EXOS has its own local memory which is not acccessable to the host processor. The EXOS accesses the shared memory via the multibus which connects the two processors. Similar to DMA memory transfers, the host is prevented from accessing shared memory while the EXOS accesses it.

Since the EXOS does not know the address of the queues before the initialization message is received, the initialization message is passed in a special way which does not use the command message queues. This will be discussed in more detail later.

2.5. CURRENT LAYERING OF THE NETWORKING SOFTWARE

Since the networking drivers do not interpret the Ethernet message received, any message may be transferred to the EXOS. The current implementation of the drivers make little use of the capabilities of the EXOS. The EXOS is used only as a semi-intelligent network interface, which performs the following functions:

-scan the net for packets to one of the host addresses
or the broadcast address
-capture packets with a proper address and store them in
local memory
-transfer the packets captured to the host system when a
receive is requested
-transmit a packet onto the network with the address specified
in the packet header

The layering of the current implementation of the networking system is shown in figure 1.

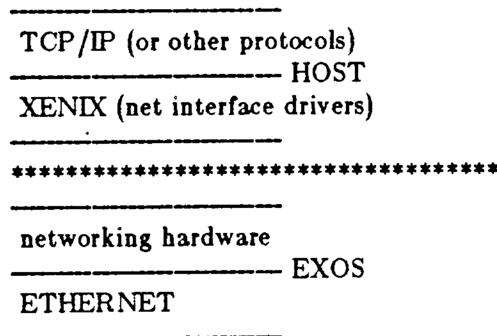

Figure 1. Where the protocol software and drivers reside

2.6. PROPOSED LAYERING FOR THE NETWORKING SOFTWARE

If the protocols could reside on the networking board, host processor time would be saved, and the computing power of the EXOS would be more fully used. Problems such as network message acknowledgement and message retry would be handled by the EXOS rather than by the host processor. The user would transfer Ethernet messages to the EXOS rather than Ethernet packets. This does not prevent the user from using datagrams (Ethernet packets); it merely forces protocol processing onto the networking board. The current implementation of the drivers assume the layering shown in figure 1. Our proposed layering for the networking system is shown in figure 2.

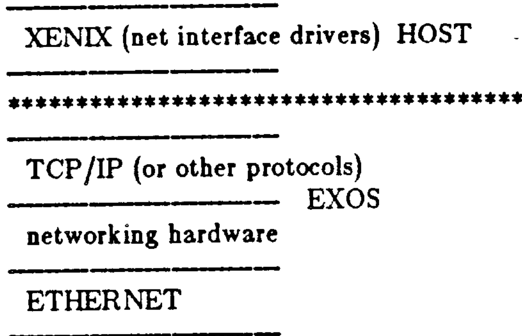

Figure 2. Where the protocol software should reside

3. DETAILED DESCRIPTION OF THE NETWORKING SYSTEM

3.1. SYSTEM HARDWARE

3.1.1. HOST SYSTEM

The hardware consists of a host system and a network interface board (EXOS/101). Any microcomputer or minicomputer system could be the host. We used a Z-8000 based system provided by Advanced Micro Devices (AMD). That system provides adequate hardware power without overkill (a minicomputer probably would have been overkill). The Z-8000 system has a 10 megabyte hard disk, an 8 inch floppy disk, and a *Multibus*[1] offering simple installation for the EXOS. The host operating system is a UNIX look-alike built by Microsoft, called XENIX. UNIX's simple device interface system (through the device file and the system buffer), easily modifiable interrupt scheme, and *C*[2] as the source code language all made the implementation of the networking board drivers simpler.

3.1.2. NETWORKING BOARD

The networking board hardware is a Multibus board designed by Excelan called the EXOS. The EXOS is a sophisticated hardware/software system with many powerful capabilities (few of which were used in this simple driver implementation). The EXOS includes an 8088 based processor system with local memory. The EXOS accesses host memory via the Multibus. The EXOS board also contains hardware to interface to an Ethernet. This includes: hardware filtering for multiple address recognition, timer hardware for timeouts, and CRC hardware for checksum generation and checking. The EXOS has resident software to process all the request messages (described in section 5). EXOS software features not used in this project include: the ability to download code from the host system, multiple process capability, and messages and mailboxes for interprocess communication and synchronization.

3.1.3. INTERRUPTS

Communication between the host system and the networking board is carried out via the shared memory, and synchronized using interrupts. The interrupts are generated by the host whenever new request messages are placed in the queue, or whenever the host has finished

1 Multibus is a trademark of Intel Corp.

2 C is a trademark of Bell Labs.

processing a reply message and is ready to return the reply message buffer. The host system interrupts the networking board by writing to a dedicated port on the networking board. Special hardware on the networking board decodes this write to a port and generates a processor interrupt.

The networking board can generate interrupts in several ways: by lowering one of the interrupt lines on the Multibus, by writing to a dedicated host port, or by writing to a dedicated host memory location. The interrupt type and address are selected by the host in the networking board initialization message. The host verifies EXOS interrupts by reading a dedicated port which provides status information from the networking board. An interrupt bit is in the status word read from the EXOS. Verifying interrupts prevents spurious interrupt problems.

It is possible for several EXOS interrupts to merge into one interrupt. For example, if the EXOS has a command message buffer to return to the host and a reply message to transfer, the two EXOS generated interrupts may be merged by the interrupt hardware into one interrupt. The host system must be prepared to accept several command message buffers, and process several reply messages as the result of one interrupt. The EXOS assumes host interrupts may be similarly merged, and performs request and reply message queue searches accordingly.

3.2. MEMORY ALLOCATION AND USAGE

The host system communicates with the EXOS through interrupts and shared memory. The shared memory is used in three ways. First, before initialization, the host system initializes an area in shared memory with the message queues (both the host to net, and net to host queue), and fills the initialization message with all the appropriate information (addresses of the message queues, interrupt type, etc..). The host then initializes the networking board by writing the address of the initialization message a byte at a time through a dedicated port. The networking board performs a self test, and then initializes itself using the information provided in the initialization message.

Second, once initialized, the host system and the networking board communicate via request and reply messages passed through the message queues which were previously initialized. Finally, if an Ethernet message is to be transferred (from host to EXOS in a network transmit, or from EXOS to host in a net receive) then the message is stored temporarily in shared memory in a system buffer.

3.2.1. MESSAGE QUEUES

Once the networking board has received the initialization message, the address of the message queues is known, and all further request and reply messages are transferred through the queues. There are two message queues, one for request messages from the host to the networking board, and one for the reply messages from the networking board to the host.

Initially all message buffers on the queue from the host to the networking board are owned by the host. This is indicated in each of the message buffers on the queue by a binary lock bit which identifies the processor (host or EXOS) owning the message buffer. For example, if a request message is to be transferred from the host system to the EXOS, first the message buffer is filled with the appropriate fields. Then, the host system changes ownership of the message buffer by changing the lock bit. Finally, the host interrupts the networking board to indicate that a new request message is on the queue. As soon as the networking board services the interrupt, and reads the request message, it changes the ownership bit, returning the message buffer to the host system, and then interrupts the host processor to indicate that another free buffer is now available. The procedure is symmetrical for the reply message buffers from the networking board to the host system.

Since several interrupts can merge, the host and the EXOS must be prepared to have several request and reply message transfers take place as the result of one interrupt.

3.2.2. DATA BUFFERS

Any data transferred between the host system and the networking board is stored in a different area in shared memory. The storage location of the data is unimportant so long as it is in the shared memory (more on this later). When an Ethernet message is to be transferred the host sends a request message. The request message contains the shared memory address of the Ethernet message if a transmit was requested or the buffer space where a received Ethernet message should be stored if a receive was requested. The message can be fragmented into as many as eight pieces called blocks. Pointers for transmit requests point to message blocks already containing a message. Receive request pointers indicate where to store the message blocks after they have been read.

3.2.3. ALLOCATION OF MEMORY SPACE

The shared memory must be accessible to both the EXOS and the host operating system. Further, the shared memory must be protected from other host system users so it is not overwritten. This implies it should reside in the operating system space. The data buffer space in shared memory should not be overwritten until the transfer of data takes place (until the appropriate reply message is received from the networking board for message transmission, or the data is transferred from the buffer space for message reception). A system of locks prevents the reallocation of data buffers in shared memory.

The above implies that both the message queues and the data buffer space should reside in an area in memory which is accessible to both the networking board and the operating system, but protected from host system users. The code for the drivers must reside in the operating system space since the drivers must be allowed to write into operating system space.

If the host system uses virtual memory, we have to guarantee that the pages of memory shared with the networking board have physical addresses identical to their virtual addresses. That is, the operating system must run in an unmapped address space. Further, we must make sure that the shared memory is never swapped out nor moved in memory as the networking board accesses this shared memory by a physical address. The above two conditions require the same solution (running the operating system unmapped), but for different reasons. Fortunately, the host system we used is not a virtual memory system.

3.2.4. WAITING FOR MESSAGE OR DATA BUFFERS

A message buffer must be obtained before a request message can be sent. Similarly, a system buffer must be obtained before an Ethernet packet can be transferred either to or from the host. If either of these buffers is not available when requested, we have a problem. Several solutions are possible.

One solution is to request the buffer, and then continuously test the ownership bit of the next message buffer, waiting until it becomes available. This will work, but causes the host to busy wait, which unnecessarily wastes host processor time.

A better solution is to request the buffer, and then block the process until a message buffer and, if necessary, a system buffer are available. A process is blocked in UNIX by calling sleep(id), where id is the name of the resource the process is waiting on. The process is woken up when a buffer is returned by the EXOS in the interrupt routine. The interrupt routine calls wakeup(id) when a buffer is returned. The call to wakeup unblocks all processes waiting on name id. Thus any process which called sleep(id) must check that the buffer is available, as another processes waiting for the same buffer may have already have woken up and taken the buffer.

4. PROCEDURES FOR ACCESSING AND CONTROLLING THE NETWORK DEVICE FILE

The following are the interface procedures used to interface the user of the network, to the network drivers. The network drivers are described in the next section. The Read procedure reads from the network device file, the Write procedure writes to the device file. The Iocntrl

procedure is the catch-all command for changing address slot assignments, enabling or disabling address slots, opening or closing the network, and gathering network statistics. Note that the device file interface is the UNIX standard interface. Thus, any software written on top of this interface is independent of the networking hardware and of the drivers.

The status returned by each of the procedures listed below indicates how the operation terminated. OK status indicates no errors were encountered. An ERROR status is returned when a faulty request is made, or a transmission or reception error is detected (e.g. checksum error, collision detection, etc..). FATAL_ERROR is returned when the error detected implies faulty initialization or hardware fault. If there is no hardware fault, recovery from a FATAL_ERROR requires that the EXOS be reinitialized.

Read(bufferpointer) takes a pointer to a linked list of three empty system buffers and reads a message from the network. The read routine fills as many of the buffers as required by the Ethernet packet being read. If no Ethernet packet is pending, then the read routine waits until a packet addressed to the host is received. The physical address of the host (unique to each EXOS sold) is used as the default host Ethernet address. Only three system buffers are required as the largest legal Ethernet packet will fit into three buffers (each buffer is 512 bytes long). If fewer than three buffers are provided, the read procedure returns with an error. The read procedure returns a value indicating the status of the read operation (OK, ERROR, or FATAL_ERROR).

Write(bufferpointer) takes a pointer to a linked list of system buffers which contain a valid Ethernet packet complete with destination address, and transmits the packet out onto the Ethernet. The Ethernet packet may consist of no more than three system buffers due to limits specified by Ethernet standards. Note that the EXOS generates the Ethernet preamble and the checksum, and thus these should not be included in the packet sent in the system buffers. The write procedure returns a value indicating the status of the write operation (OK, ERROR, or FATAL_ERROR).

Iocntrl(function, array) is the catch-all procedure to allow the operating system to request the other functions necessary to properly operate the EXOS. The input parameters to iocntrl are, as shown above, a function code to select one of the EXOS procedures, and an array of input parameters to that selected procedure. The procedure is selected by specifying one of the following functions, and filling the parameter array as indicated below (refer to the section on low level procedures for interfacing to the EXOS for details on the specifics of each parameter described below). The iocntrl procedure returns the status of the function selected (either OK, ERROR, or FATAL_ERROR).

Net Initialization – function = NINIT (where NINIT = 07_{16}). Net initialization takes no parameters, so the parameter array is unused.

Net Address – function = NADDRESS (where NADDRESS = 09_{16}). The parameter array is used in the following way:

INPUT

array[0] – read/write request mode.

array[1] – the address slot to be queried or changed or both.

address slot may be:

1-7 user defined address slots

253 physical address slot

255 broadcast address slot

array[2 to 7] – if write, six bytes specifying an Ethernet address.

OUTPUT

array[2 to 7] – if read, Ethernet address corresponding to the address slot.

Net Mode – function = NMODE (where NMODE = 08_{16}). The parameter array is used in the following way:

INPUT

array[0] – read/write request mode.

array[1] – if write selected, new network receive options.

array[2] – if write selected, new address filtering mode.

OUTPUT

array[1] – if read selected, old network receive options.

array[2] – if read selected, old address filter mode of EXOS.

Net Close – function = NCLOSE (where NCLOSE = 06_{16}). Netclose takes no parameters, so the parameter array is unused.

Net Slot Select – function = NRCV (where NRCV = $0A_{16}$). The parameter array is used in the following way:

INPUT

array[0] – the address slot to enable/disable or interrogate.

array[1] – if write, the new mode for the specified address slot.

OUTPUT

array[1] – if read, the old mode of the specified address slot.

(ENABLED, or DISABLED)

Net Statistics – function = NSTSTCS (where NSTSTCS = $0B_{16}$). The parameter array is unused by net statistics.

5. LOW LEVEL PROCEDURES FOR INTERFACING TO THE NETWORKING BOARD

The following procedures support fundamental networking operations. These procedures, called by the host system, cause request messages to be sent to the networking board. The procedures act as drivers, interfacing the existing operating system on the host to the network board. The host system and the networking board communicate through an area in memory shared by both systems. Communication synchronization is controlled by message ownership locks and interrupts.

The command messages support the following operations: open the network, designate network address and mode, transmit message, receive message, close network, and network statistics. The proper reception and acknowledgement of each command message is determined by the net interrupt handler. Neither the host processor interface software, nor the networking board (EXOS) interpret the message data, they only propagate it on the network.

In the following description of procedures for interfacing to the networking board, INPUT refers to parameters passed to the procedure, OUTPUT refers to data structures altered or potentially altered by the call to the procedure, and finally RETURN refers to the value returned by the procedure.

Note that in calling any of the procedures listed below, the input parameters are also used in some cases to return values to the calling program. Thus, the parameters are called by name instead of by value. In other words, to use the routines listed below all calls to the routines must have the following format:

netaddress(&rwrequest, &addresslot, &address)

Where rwrequest, addresslot, and address are variables which are defined in the calling routine, and have been assigned values which properly select the function to be performed by the EXOS. If a read operation was selected in the rwrequest parameter, these procedure returns. Otherwise, they will contain the original values given to them. Not all variables will return values, but for consistency all variables are passed to the network drivers in this manner.

Error(type, explanation) allows all procedures to make a record of errors. Errors fall into two categories: errors and fatal errors. Errors are caused by faulty commands, or insufficient memory space for messages to be stored. These errors can generally be corrected by repeating one or more request messages, though some Ethernet messages may be lost as the result of faulty request messages. Fatal errors are caused by hardware faults or inconsistent requests made of the EXOS. Recovery from fatal errors, if possible, requires that the host system attempt to reinitialize the EXOS. Error messages describe the procedure where the error was detected, and a probable cause for the error. Each error message indicates the type of error which occurred (error, or fatal error), the procedure in which the error was detected, and the type of command message which detected the error.

INPUT: type (ERROR, FATAL_ERROR)
description (80 character array with a short error message)
OUTPUT: NONE
RETURN: NONE

NetInit() sends a net initialization message to the EXOS. The net initialization message inidicates to the EXOS where the shared memory is located, where the message queues are located, what type of interrupt is to be generated, and the address of the network interrupt routine. The NetInit message is unique in that it does not use the message queue structures. Instead, before reception of the NetInit message, the EXOS does not know the location of the shared memory or the message queues.

INPUT: NONE
OUTPUT: NONE
RETURN: NetInit status (OK, ERROR, or FATAL_ERROR)

NetMode(rwrequest, options, mode) sends a net mode message to the EXOS board. The net mode message indicates to the EXOS which types of error packets should be received, and what type of net address filtering should be used. The rwrequest selects if the host is reading the net mode, writing a new net mode, or both. The options allow the reception of packets which are considered to be in error (i.e., alignment error, CRC error, or disable controller without disconnecting from the network). The mode selects the address filtering employed by the EXOS. Net address filtering can be one of three types: perfect filter (hardware and software), hardware filter only, or promiscuous mode (receive all packets on the network). The perfect filter requires both a hardware filter as well as a software filter. Requesting just the hardware filter speeds up address filtering, but may result in the reception of some packets which are not addressed to the host.

INPUT: rwrequest (READ, WRITE, or READ | WRITE (both))
options (ALGN_RECV, CRC_RECV, or NET_DISABLE (or any combination))
mode (P_FILTER, H_FILTER, or PROMISCUOUS)
OUTPUT: options returns old options (if read)
mode returns old mode (if read)
RETURN: NetMode status (OK, ERROR, or FATAL_ERROR)

NetAddress(rwrequest, addresslot, address) sends a net address message to the EXOS. The net address message indicates to the EXOS what address slot (a one byte value) will be used to refer to the Ethernet address supplied. The rwrequest field indicates if the request message is writing to the EXOS, or interrogating the EXOS for the Ethernet address currently filling the specified address slot. Up to eight addresses can be recognized as addresses for the host. Two address slots are treated as special cases. Slot 253 is the physical address slot (unique to each EXOS board produced), and slot 255 is the broadcast address slot. Sending an address to the networking board does not enable reception of messages for that address. The enabling of addresses must be done separately with a NetSlotSelect call.

INPUT: rwrequest (READ, WRITE, or READ | WRITE (both))
addresslot (0 to 7, 253, or 255)
address (any valid multicast Ethernet address)
OUTPUT: address returns the Ethernet address (if read)
RETURN: NetAddress status (OK, ERROR, FATAL_ERROR)

NetClose() disconnects the EXOS board from the network. Any messages sent to the host which pass over the network after a call to NetClose are neither received by the host, nor acknowledged. This is the normal termination to a networking session. Note that there can be no outstanding request messages to the EXOS when NetClose is called. When there are outstanding request messages, NetClose considers a call to itself as a fatal error.

INPUT: NONE
OUTPUT: NONE
RETURN: NetClose status (OK, ERROR, FATAL_ERROR)

NetTransmit() sends a net transmit request message. The net transmit request message takes a packet which has already been fragmented, placed in buffers, and stored in shared memory, and indicates to the EXOS that there is a message to be sent. The message buffers are not reused until the message has been properly sent by the EXOS board over the network, but not acknowledged by the destination node. That is, buffers are not reused until the EXOS returns a reply message indicating that the packet was sent. Retransmission of a message due to no acknowledgement must be handled by the protocol layer. Note that the EXOS attachs an Ethernet preamble and an Ethernet checksum to the message, thus these should not be included. The EXOS assumes that the message in the buffers provided has a valid Ethernet address.

INPUT: NONE
OUTPUT: NONE
RETURN: NetTransmit status (OK, ERROR, FATAL_ERROR)

NetReceive(addresslot) indicates to the EXOS that it should transfer any message addressed to the Ethernet address contained in addresslot (specified by a previous call to NetAddress) into the shared memory so that it can be reassembled and interpreted. The physical Ethernet address assigned to the EXOS may be selected as the address by specifying addresslot 253. The message received is placed in system buffers which are not reused until the message is stored and the system buffers returned. If no message is present, then empty address slot status is returned (address slot zero).

INPUT: addresslot (the addresslot containing the desired Ethernet address)
OUTPUT: the system buffers are filled with any message received
RETURN: NetReceive status (OK, ERROR, FATAL_ERROR)

NetSlotSelect(mode, addresslot) sends a net slot enable request message to the EXOS. The request message allows the host system to query the EXOS for the mode of the address slot specified in the call to NetSlotSelect. The request message may also change the mode of the address slot. Allowable modes are enable receive, and disable receive. In disable receive mode, any messages addressed to the host are not received. In enable receive, mode messages are transferred to the host when requested. If the receive mode is to be changed, then write request mode must be selected. Otherwise, if only the status of the address slot is to be checked, then read request is specified. The result of the NetSlotSelect is returned as status in the net information structure (either ENABLED or DISABLED).

INPUT: mode (READ, WRITE, REC_ENABLE (WRITE | REC_ENABLE to enable))
addresslot (the addresslot to be queried or changed or both)
OUTPUT: mode returns the old addresslot mode (if read) (ENABLE, or DISABLE)
RETURN: NetSlotSelect status (OK, ERROR, FATAL_ERROR)

NetStstcs() sends a net statistics message to the EXOS. A net statistics message causes the EXOS to reply with the network statistics counters which have been gathered. The EXOS keeps eight 32 bit network statistics counters (counter number 3 is unused). The EXOS records the following network statistics: the number of packets sent with no errors, the number of packets transmissions aborted due to excess collisions (more than 16), the unused counter, a time domain reflectometer, the number of packets received with no errors, the number of packets received with alignment errors, the number of packets received with checksum (CRC) errors, and finally the number of packets lost (no buffers available for reception). The time delay reflectometer counts the delay between start of packet transmission and collision detect in 100ns increments. The time delay reflectometer helps detect hardware problems with the physical Ethernet cable. The network statistics are returned in a reserved memory structure. The statistics counters kept by the EXOS are cleared on every call to NetStstcs.

INPUT: NONE
OUTPUT: stats data structure contains new statistics
RETURN: NetStstcs status (OK, ERROR, FATAL_ERROR)

NetInt() is the network interrupt routine. This interrupt procedure is called when the operating system sees an interrupt from the network hardware. The net interrupt routine checks to see if the interrupt was not spurious, if the proper reply message was received, and if the command completed successfully. The net interrupt cleans up the net information data, moves data as necessary (for NetReceive), and returns the status of the interrupt. Note that all status and data manipulation caused by calling one of the above procedures is returned from the NetInt routine, as this is where proper command completion is first known.

INPUT: NONE
OUTPUT: exos.e_status (OK, ERROR, FATAL_ERROR)
exos.e_mode (if NetMode read, or if NetSlotSelect read)
exos.e_address (if NetSlotSelect read, an Ethernet address)
exos.e_options (if NetMode read, the network options)
and potential system buffers (from NetStstcs, or NetReceive)
RETURN: NONE

6. CONCLUSION

I have listed several reasons why the need for a network monitor/debugger has prompted the development of the hardware/software system described here. We recognize a need for these network development/testing tools. To speed up the development of these networking tools, I designed and implemented a set of general network interface drivers which could be used later to implement the desired set of network tools.

The network drivers were designed to be general in order to allow for their adaption to a variety of problems. This feature of the drivers is also a limitation, since generality is often accompanied by lower performance. The drivers described here should perform reasonably well, and should be relatively easy to improve when the network monitor/debugger requirements are better understood.

Some possible improvements being planned have been mentioned (placing the protocol software on the networking board). Other improvements should wait until the high-level design of the network monitor/debugger is complete. For example, a special fast procedure to send the time for clock synchronization would decrease the clock skew between the network debugger to other processors on the net. This feature cannot be implemented without a format for the time message and the slave debugger software design.

7. ACKNOWLEDGEMENTS

Special thanks to the following companies and people for providing invaluable equiment loans and information. Advanced Micro Devices for the Z-8000 system. Microsoft, for the XENIX operating system and source code, without which the operating system patches could never have been made. Excelan, for the EXOS/101 networking board, documentation, and hours on the phone. Professor Domenico Ferrari, for putting all the connections together, and guiding the network monitor/debugger brainstorming sessions. And last, but certainly not least, Joe Pasquale for his explanation of the XENIX internals, and most important, his work patching the drivers to XENIX.

8. REFERENCES

UNIX programmer's manual seventh edition, March 1983.

Kernighan, B. W., Ritchie, D. M., **The C Programming Language** Prentice-Hall inc. Englewood Cliffs, New Jersey, 1978.

EXOS/101 Ethernet Front-End Processor Reference Manual 2nd edition, 1982.

Ideas for the network monitor/debugger from brainstorming sessions lead by Prof. Domenico Ferrari, Fall semester 1983.