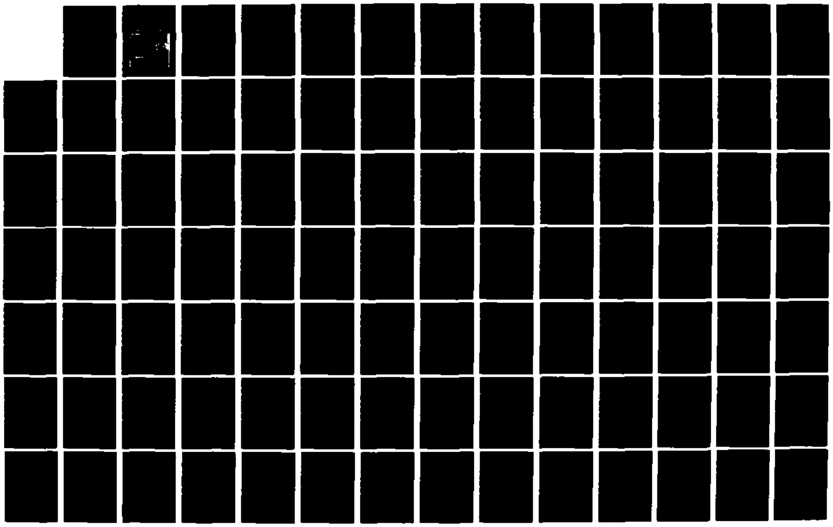
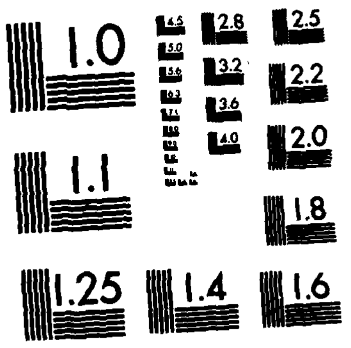
AD-A124 448   A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET   1/2
(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB
D J LILJA AUG 82 R-946 N00039-80-C-0556

UNCLASSIFIED   F/G 9/2   NL



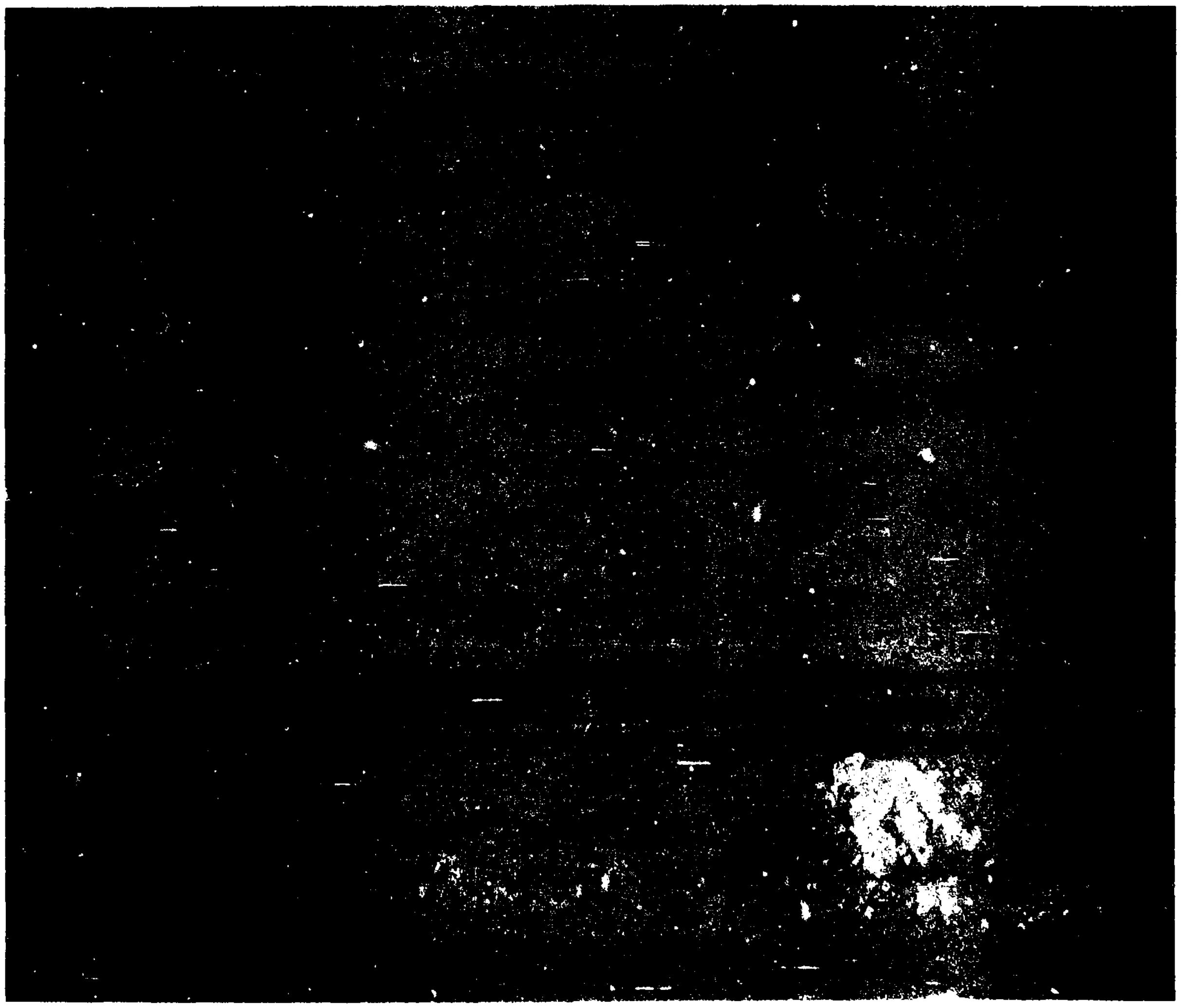

1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

REPORT R-946 AUGUST, 1982 UILU-ENG 82-2212

12

COORDINATED SCIENCE LABORATORY

A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A124448	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER R-946; UILU-ENG 82-2212
7. AUTHOR(s) David John Lilja		8. CONTRACT OR GRANT NUMBER(s) US NAVY N00039-80-C-0556
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS VHSIC		12. REPORT DATE August, 1982
		13. NUMBER OF PAGES 106
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Local networks
Ethernet

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An Ethernet is used to connect an HP 3000 computer with a VAX 11/780 computer to allow the transfer of files from one computer to the other. A user logs in on the VAX computer and uses a one-line command to send a file to or retrieve a file from the HP 3000 computer. Files are transferred between the user's directory on the VAX and either a specified directory or a public "network" directory on the HP 3000. The file transfer system uses a scheme of positive acknowledgment with retransmission to prevent transmission errors from corrupting the file.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UILU-ENG 82-2212

A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET

by

David John Lilja

This work was supported in part by the General Electric Fellowship in Electrical Engineering and in part by VHSIC under Grant US NAVY N00039-80-C-0556.

A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET

BY

DAVID JOHN LILJA

B.S., Iowa State University of Science and Technology, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1982

Urbana, Illinois

A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET

David John Lilja, M.S.
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1982
Advisor: Professor Edward S. Davidson

ABSTRACT

An Ethernet is used to connect an HP 3000 computer with a VAX 11/780 computer to allow the transfer of files from one computer to the other. A user logs in on the VAX computer and uses a one line command to send a file to or retrieve a file from the HP 3000 computer. Files are transferred between the user's directory on the VAX and either a specified directory or a public "network" directory on the HP 3000. The file transfer system uses a scheme of positive acknowledgement with retransmission to prevent transmission errors from corrupting the file.

ACKNOWLEDGEMENT

I would like to thank my thesis adviser, Professor Edward S. Davidson, for his help and advice in preparing this thesis. Also deserving of thanks are the following people: Larry Hanes and Dick Norton for helping me to understand the VAX, Bill Rogers for his help in taming the HP 3000, Howard Pollard for being able to answer an endless barrage of questions, Steve Yates for helping me maintain my sanity through the entire process, and my parents for their continued support and encouragement throughout my academic life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Over the past several years there has been a great deal of interest in interconnecting many geographically distributed computer systems using computer networks. These networks range from long-haul networks connecting computers separated by thousands of miles to multiprocessor computers which are separated by less than a few meters. Local computer networks are generally considered to be those with separations on the order of a few kilometers or less and are usually contained within one building and used by one organization[1]. Local networks are attractive because they allow expensive system resources, such as printers and secondary storage devices, to be shared among several independent computer systems. With the rise in popularity of personal computers, a great deal of work has been done on local networks to connect these small systems and produce some standards[2].

This thesis describes an implementation of a local computer network that allows the transfer of files from one of the host computers on the network to another host. This local network uses an Ethernet to interconnect a VAX 11/780 computer and a Hewlett-Packard HP 3000 computer. The system allows a user to log in on the VAX and use a simple one line command to transfer any file to the HP 3000 or receive any file from the HP 3000.

1.1. Ethernet Description

Ethernet is a registered trademark of the Xerox Corporation used to refer to a particular method of implementing a local computer network[3]. This Ethernet employs packet broadcasting with several stations connected together by a common transmission medium. Every station on the network can receive every transmission of every other station. A station is any device, such as a computer or a printer, that can communicate on the network. The Ethernet concept is predicated on the assumption that computer traffic is bursty. Thus several stations can efficiently share the common transmission medium using time-division multiplexing. This concept of packet broadcasting originated with the Aloha Network which was developed at the University of Hawaii to connect together several computer sites with radio[4]. Ethernet is classified as a "carrier-sense multiple access with collision detection" (CSMA/CD) network which means that multiple stations are able to sense the state of the medium before transmission and also detect if another station is causing interference.

The Ethernet has a central interconnection medium with network control distributed among the stations. This interconnection medium is completely passive and is usually a standard coaxial cable, referred to by the developers at Xerox as the "ether." The name "ether" is from the historical "luminiferous ether" through which electromagnetic radiation was once thought to propagate. The topology of this cable must be an unrooted tree. The tree requirement prevents multiple paths between any two stations on the cable so no transmitted packet can interfere with itself. It is unrooted in the sense that the cable can be extended in any direction by simply tapping into the cable at any point. The ends of the cable must be properly terminated with a passive resistance whose value is the same as the nominal cable impedance to prevent echoes and reflections.

Access to the network is shared among the stations on a contention basis. When a station wants to transmit a packet, it first listens to the ether to see if it is already in use by another station. If it is in use, the new station simply waits until the other packet is finished transmitting. When the ether is clear, the station waiting to transmit seizes the ether and begins transmitting its packet. This sending station monitors its own transmission to make sure that it is not colliding with another station trying to transmit at the same time. If a collision is detected, each station aborts its transmission and waits a random period of time before attempting to retransmit. Once the station is able to seize the ether without a collision, it transmits its entire packet.

Each station on the network is assigned a unique eight-bit station address that is used by the other stations to specify the desired destination of a packet. Using eight bits for the address limits the number of stations on the network to 256. In practice, most Ethernets limit the number of stations to 255 and use one of the addresses, usually the zero address, as a broadcast address. Normally a station examines the destination address of every packet on the ether, but it will respond only to its own address or to the broadcast address. Broadcast is useful for sending all of the stations the same packet, such as time of day information or a system message. In most Ethernet systems, broadcast messages are not acknowledged by the individual stations.

Reliability in Ethernet is based on simplicity of design. By having a passive central interconnect and distributed control, if one station should go down for some reason, it can be disconnected with no effect on the rest of the network, except, of course, that the services provided by that station will no longer be available. Due to the simplicity of this system, packets are correctly delivered to their destinations with only a high probability of success. Some packets will be lost due to collisions with other packets, interference from electromagnetic noise, and being discarded by the intended receiver. Error control for packet transmissions must be dealt with at a higher level and is not a concern of this low-level Ethernet protocol.

1.2. Levels of Protocol

As with most complex systems, this computer network is best developed at several different levels with each level being logically distinct from the others. There are basically four different levels of protocol used to provide communication between the two computers on this network. The lowest level, which is the Ethernet protocol described above, is implemented in a microprocessor based controller, with one controller required for each host. The remaining three levels of protocol are implemented entirely in software on the host computers and represent the main work of this thesis.

The second level of protocol, the host to controller communication protocol, is concerned with how each host sends commands to the slave controller and how the host receives packets from the Ethernet that have been buffered within the controller. The next level of protocol is the file transfer protocol which is used to effect the actual transfer of files between the two host computers and provides the error control mechanism that is missing from the lower-level Ethernet protocol. The highest level of protocol used in this network communicates with the user to allow an easy mechanism to transfer files. Within this level is a sub-protocol that initiates communication between the two host computers before a file can be transferred. When all of these levels are put together, a user can simply log in on the VAX and, using a simple command, can transfer files to and receive files from the HP 3000.

2. HOST TO CONTROLLER COMMUNICATION

The Ethernet concept requires each machine on the network to have an intelligent controller to interface with the ether. This controller can be the host machine itself, but it is desirable to have a separate device dedicated to the network connection. This prevents the host from being burdened with the low-level protocols of the network and allows it to concentrate on higher-level protocols and user processes.

This separation of host and network controller requires that some communication link exist between these two devices. This link could use a dedicated parallel input/output port or a simple serial interface. Also required is the existence of some protocol for communication between the two devices. The complexity of this protocol will vary depending upon the computational power and the relative speeds of the host and the controller.

2.1. General Host/Controller Communication

The local network described in this thesis uses a network controller designed and built by Joseph Ayala at the Coordinated Science Laboratory[5]. This controller uses a microprocessor for its basic intelligence and a specially designed board for the bit transfers on the ether. The controller is connected to the host computer with a standard RS232 serial interface line and appears to the host as a terminal. By using this standard interface, the controller is very portable and can be easily moved to another host with little or no modification. The penalty for this portability is the relatively slow speed of the serial transfers over the communication link, which in this case is limited to 9600 baud.

The controller is configured in a "null modem" fashion, thereby allowing it to be connected directly to the terminal ports of most computers without swapping the transmit and receive wires. The controller requires the DSR (data set ready) signal from the host to be active, but it can be used with three wire systems by connecting pins 6 and 20 together on the 25-pin host-to-controller EIA connector. This jumper connects the controller's DTR (data terminal ready) signal to its DSR connection and thus eliminates the need for the host to supply the DSR signal.

The communication protocol designed into this controller requires the host to initiate all communication by sending a single byte command with the controller always responding by returning two bytes to the host. This initiation responsibility requires a program to be running on the host to communicate with the controller. That is, the controller does not functionally look just like a terminal to the host, but requires a special intelligent process to be running on the host.

Within this protocol, there are four basic commands that the host can issue to the controller. These commands are shown in Table 1. The enquiry command (ENQ) causes the controller to return its station address byte and its current status byte. The station address is the unique eight bit network address to which the controller will respond when there is a packet sent on the ether. The status byte indicates the current status of the controller itself. The bit format of the status

Table 1. Controller Command Bytes

Command	Hex Code	Description
ENQ	05	Enquire
SEND	11	Send a packet
SOH	01	Start of header
ACK	06	Acknowledgement
REC	12	Receive a packet
NAK	15	Negative acknowledgement
CAN	18	Cancel last receive request

byte is shown in Table 2. This command is useful for initializing the station address in the host's software and for checking the status of the controller. Also, since the controller will not respond if it is not turned on and operating, ENQ can be used to test this condition. Once set, the status bits can only be reset by a manual reset of the controller. This fact limits the usefulness of ENQ. The software developed in this research actually uses ENQ only to initialize the sta-

Table 2. Status Byte Description

Bit no.	Meaning when logic 1
0	Always 0.
1	Transmit aborted due to excess collisions.
2	Receive aborted due to "cancel" command.
3	Command from host not recognized.
4	Receiver overrun - error on communication link to host.
5	Checksum error in the received packet.
6	Received packet did not contain multiple of 8 bits.
7	Received packet exceeded 1024 bytes in length.

tion address and to test if the controller is turned on and operating.

The send command (SEND) prepares the controller to send a packet over the net. The packets can be a minimum of two bytes in length (for the source and destination addresses), and a maximum of 1024 bytes (limited by the size of the buffer within the controller). The sending controller hardware appends a two byte CRC (cyclic redundancy code) checksum to the packet, but this is unavailable to the host. After receiving a SEND byte, the controller responds with a "start-of-header" byte (SOH) and a null byte (to pad its response to the standard two bytes). The host then sends the two byte length of its packet, low byte first, and waits for an acknowledgement byte (ACK). When the ACK arrives, the host sends its entire packet to the controller which buffers it and transmits it when the ether is clear. Note that the SEND command is required once for each packet to be sent.

To receive a packet from the net, the host must first issue the receive (REC) command. This causes the controller to listen to the ether, waiting for a packet with its station address or the broadcast station address. When a packet directed to it is received, the controller reads the packet into its buffer and informs the host with either an acknowledgement (ACK), if the packet was received correctly, or a negative acknowledgement (NAK), if the packet was damaged. The host can then either read the packet or cancel the receive request. To read the packet, the host sends the SOH byte to the controller, which then responds with the two byte length, low byte first. The host follows with an ACK byte causing the controller to send the entire packet to the host and return to its idle state waiting for the next command. If instead the host wants to cancel the receive, it can send the controller a cancel command (CAN) which returns the controller to its normal idle state and makes the received packet unavailable to the host. Note that the REC command must be issued once before each packet the host expects to receive.

This simple protocol must be strictly followed for proper transfers to take place between the host and the controller. Any deviation from this protocol may confuse the controller and cause it to "hang" in an indeterminate state. It then must be reset manually.

2.2. Using the VAX 11/780 as a Host

A VAX computer running under the UNIX timesharing operating system is used as one of the hosts in this local network. UNIX is written mainly in the programming language C and allows user programs written in C to interface directly with the operating system[6]. This capability allows the user a great deal of flexibility in interfacing nonstandard devices, such as the network controller, to the computer. UNIX treats all input/output requests as accesses to a file, specifically, requests to access terminal ports are treated exactly as requests to read or write data on disc files. This convention allows one standard interface between the user's program and the operating system.

As mentioned previously, the controller connects to the host through a standard terminal port. To access this port, the user program must open it using the "open" procedure call available in the standard input/output library "<stdio.h>."[7] This procedure expects as parameters the name of the port and the type of access required. The name is specified as "/dev/ttyx" where x is the terminal port number to which the controller is connected, and the access type is the integer 2 which specifies read/write access. This procedure call returns an integer which is then used to identify the port in subsequent read/write requests.

After the port is opened, it must be configured using the "set terminal options" procedure (stty) available in the "<sgtty.h>" library[7]. This procedure allows the user to change default options in the terminal driver. For the network controller the options are changed so that the input and output speeds are set to 9600 baud and the mode is selected as RAW and ANYP. RAW means that all characters are passed to and from the port untouched with no processing done on the characters and no special characters, such as "break," being recognized. The ANYP flag means that any parity is allowed, i.e. the parity bit is ignored.

Once the port is opened and configured, the user's program can read and write to the port using the "read" and "write" procedures available in the standard C input/output library. These procedures allow the program to transfer many bytes at a time from a buffer within the program to the port and to read many bytes from the port into the program's buffer. In most cases, single byte transfers are the simplest to implement and are not much slower than multi-byte transfers since UNIX already buffers the port transparently to the user.

It should be noted that before the port can be properly configured, the user must make sure that no other process is running on that port. Specifically, UNIX normally has a process called "getty" running on each terminal port[7]. This process initiates the login procedure when the port is used for normal interactive sessions. This "getty" process ties up the port and prevents normal access to the controller. Also, since the network controller is not an interactive terminal, the "getty" process will not respond correctly to the controller's protocol. Consequently, the process must be removed by the system management from any port to which the controller is to be connected. Once this is done, the port can be opened and configured as described above.

2.3. Using the HP 3000 as a Host

The other host computer used in this local network is a Hewlett-Packard HP 3000 running the MPE (Multi-Programming Executive) operating system. Systems programming on the HP 3000 is best accomplished using the Hewlett-Packard language SPL (Systems Programming Language) since it allows the most direct interface to the operating system[8]. As in the VAX, the HP 3000 treats all input/output requests as accesses to a file. Thus data transfer to and from a terminal port is accomplished in the same manner as data transfer to and from a file.

The software terminal drivers on the HP 3000 are optimized to work with Hewlett-Packard terminals and present a few problems when trying to interface non-standard devices to the terminal ports. The drivers associate a record size with each terminal which specifies the maximum size of each transfer between the terminal and the computer. Each line is delimited by a carriage return or by this maximum record size. If the record size of the terminal is specified to be greater than eighty characters, then the driver uses an "enquire-acknowledge" protocol for all transfers to the terminal. This means that the computer will send out an enquire byte (ENQ) to the terminal before it transfers any information. It will then wait for an acknowledgement byte (ACK) indicating that the terminal is ready. This protocol is necessary to prevent the computer from overwhelming slow terminals with too much information at once. Unfortunately, the network controller does not respond correctly to this protocol.

Another problem in the terminal driver is that whenever the computer expects input from the terminal, it sends out an ASCII "DC1" byte. This is supposed to alert the terminal that it can now send its information to the computer. The problem in the network controller is that this byte is interpreted as the SEND command and sets up the controller to send a packet over the net. Obviously, this is not compatible with the intended use of this byte by the terminal driver.

Both of of these problems must be eliminated before the controller can be used with the HP 3000. One solution would be to reprogram the microcomputer inside the controller to respond correctly to this protocol. While this could be done relatively easily, it is not a very good solution because the controller would no longer be portable. A better solution, described below, is to eliminate both problems and allow the user's program to handle its own protocol. This solution was adopted in this design.

The enquire-acknowledge problem is eliminated by specifying the record size of the terminal port to be fewer than 80 characters when the port is opened. The "DC1" problem is eliminated by specifying the terminal to be type 18. This terminal type is not documented in the HP 3000 manuals and the information regarding type 18 terminals was obtained through the courtesy of the Hewlett-Packard Company. Type 18 can be specified by using either the MPE intrinsic procedure FCONTROL to allocate the terminal to the program as type 18, or by specifying type 18 for the required port in the system configuration table. Using this terminal type effectively eliminates the built-in protocol in the terminal driver.

Before the controller can be accessed by the user's program, the terminal port to which it is connected must be opened using the FOPEN intrinsic[9]. The required parameters are a bit string indicating the file type, another bit string indicating the access options, an integer specifying the record size, and the name or logical device number of the controller's port. This procedure call returns an integer which is then used to identify the file in subsequent accesses. Once the port is opened, it must be further configured using the FCONTROL intrinsic to select certain options. This intrinsic is used to turn off the automatic echo of input characters; disable the input timer; disable block mode transfers (part of the enquire-acknowledge protocol); enable binary transfers (to pass all eight data bits); and disable the parity, treating each bit as a valid data bit. Also, the FSETMODE intrinsic must be called to inhibit the automatic carriage return/line feed sequence after each input from the controller. After all of this is done, the controller is ready to be accessed.

The MPE operating system provides two intrinsics to read and write files -- FREAD and FWRITE. Each of these allow the user to specify the file number, the buffer to be used (this buffer must be supplied within the user's program), and the number of bytes to be transferred. FWRITE also allows a carriage control byte to be sent with each transfer. This should be specified as "%320" which means that no carriage control should be sent. If the multirecord option was specified in the access options parameter of the FOPEN call, then the number of bytes transferred does not have to be fewer than the specified record size. The read or write will simply transfer as many bytes as are requested in the procedure call. The multirecord option effectively circumvents the 80 character limit set above to avoid the enquire-acknowledge problem. Packets up to the maximum length allowed by the controller can then be sent and received by the HP 3000.

The first problem encountered in reading from and writing to the controller is that the controller responds too quickly. That is, after a command is sent to the controller, it responds before the program running on the HP 3000 can execute the read procedure. This problem arises here since the computer does not automatically buffer data coming from terminals unless it is already expecting some form of input. The solution to this problem is to set up the read request before sending any commands to the controller by specifying the "no-wait I/O option" in the access options parameter of the FOPEN intrinsic. To execute this intrinsic with this option, however, the program must be put into "privileged mode" using the GETPRIVMODE intrinsic. After the FOPEN is executed, the program can return to its normal mode by executing the GETUSERMODE intrinsic.

Now to read and write to the controller, the program must first execute the FREAD. This will immediately return control to the calling procedure which should then execute the FWRITE to send the command to the controller, followed by a call to the IOWAIT intrinsic to finish the read. The result is that the computer is waiting for the controller's response even before the command is sent to the controller. The rapid response is then automatically buffered by the computer to be transferred to the program via the IOWAIT call.

A problem encountered when using this procedure is that the program cannot set up a read followed by a write to the same file without an intervening call to IOWAIT. An IOWAIT here is improper since no read data would arrive until the write has been issued. The obvious solution is to use two different files, one for read requests and the other for write requests. However, MPE will not allow the same terminal port to be referenced by two different files simultaneously. The final solution is to use two different files, each referenced to a physically separate terminal port. Unfortunately, now two ports are required at the HP 3000 for connecting the Ethernet controller.

To connect the controller to two separate ports, a special connector has been fabricated. The wiring for this connector is shown in Figure 1. Notice that the transmit data and receive data wires (pins 2 and

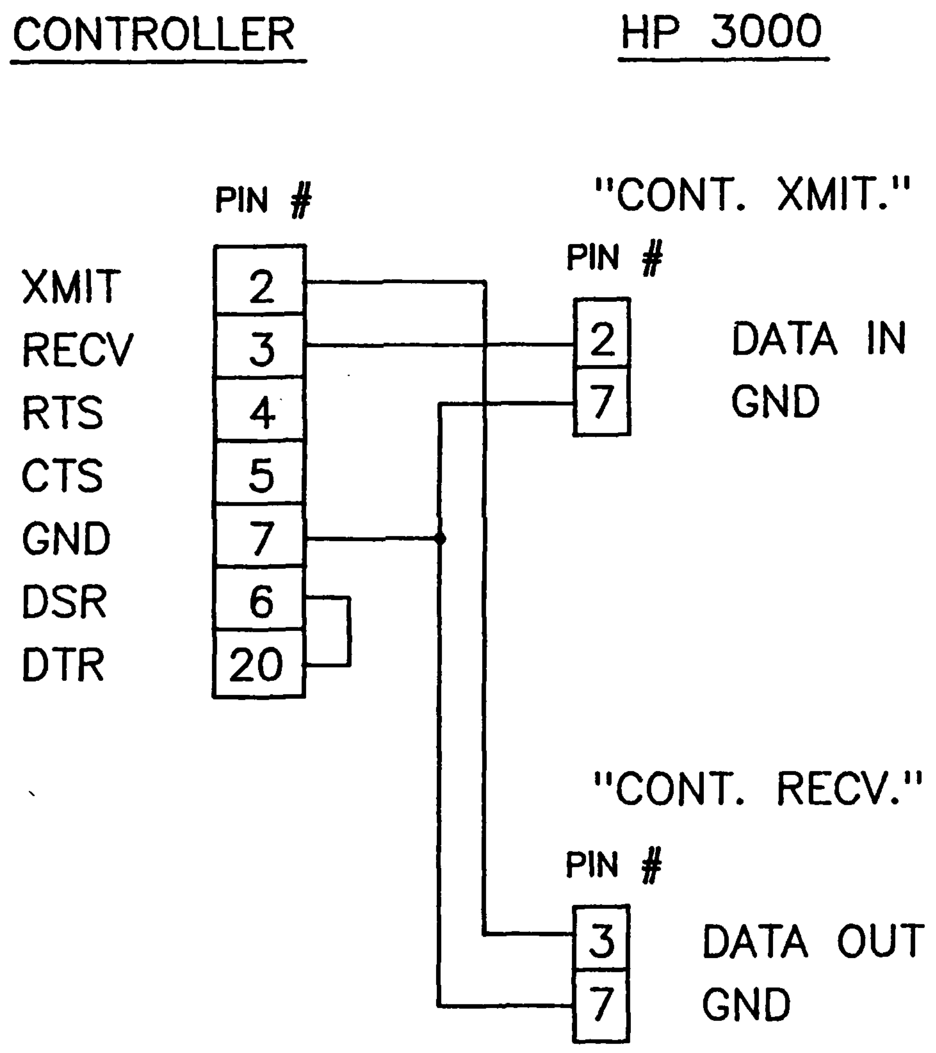


Figure 1. Controller to HP 3000 connection

3) are swapped from the controller's connector to the connector on the computer. This swap is necessary because the terminal connection block on the HP 3000 is also configured in a null modem fashion. Using this connection arrangement requires the controller to occupy two terminal ports, one for sending data to the controller and the other for reading data from the controller. While this arrangement has its disadvantages, it seems to be the best solution available without changing the program within the controller itself.

2.4. The PACKET Program

As a demonstration of the host to controller communication protocol, the program PACKET was written to allow the transmission of packets between the two hosts. A version of this program is written in C to run on the VAX computer (see Appendix 1) and another version is written in SPL to run on the HP 3000 computer (see Appendix 2). The program allows the user to execute all of the commands available to the controller (i.e. SEND, REC, ENQ, and CAN), and allows packets to be sent from a terminal on either machine to a terminal on the other machine.

When the program is run, it first executes the ENQ command to determine if the controller is connected and functioning and to initialize the station address within the PACKET program. It then prints this address on the terminal and requests the user to supply the station address of the controller on the other machine. After this initialization process is completed, the program prints a menu of available commands and requests the user to enter the desired command.

The commands available are "send a packet," "receive a packet," "enquire" (executes the enquire command), "cancel" (executes the cancel command), "help" (prints out the list of commands), and "quit." To send a packet, the user is prompted to enter the packet from the terminal, terminated by a carriage return. This packet is then sent out over the net. For each packet that is to be received, however, the destination controller must be put into the receive mode by using the receive command any time before the packet is sent. When the packet is received, it is simply displayed on the terminal and discarded. If the controller is not in the receive mode, it does not monitor the ether and all packets are simply ignored.

While this program is relatively simple, it does demonstrate the use of the low level host to controller communication protocol and the feasibility of sending packets on the net. It is also useful as a tool to debug more sophisticated network protocols by allowing the user to monitor packet transmissions on the ether. This monitoring is done by running the PACKET program and repeatedly executing the receive command for each packet that is to be received. This program forms a basis for the more complicated programs to follow.

3. THE FILE TRANSFER PROTOCOL

The file transfer protocol builds on the previous host/controller communication protocol to provide for the orderly transfer of files as a sequence of packets. This level of protocol is tolerant of transmission errors and damaged packets and provides a mechanism for recovering from these errors. This mechanism is known as a stop-and-wait protocol with positive acknowledgement and retransmission[1]. What this means is that the sender transmits a packet and waits for the receiver to acknowledge correct reception before sending another. If the receiver never acknowledges the packet, perhaps because it was damaged or never received, the sender times out and retransmits the packet. The receiver only acknowledges the correct reception of packets knowing that unacknowledged packets will be retransmitted by the sender after the time out interval. This protocol provides a simple, relatively error-free method for transmitting files.

3.1. Packet Types and Format

All of the packets used in this protocol have the same format, shown in Figure 2, which uses a six byte header to provide the information required in each packet. The first two bytes of each packet consist of the destination and source addresses with the destination address first. The use and ordering of these two bytes is determined by the Ethernet definition. The controller hardware expects to find the destination address in the first byte. The third byte contains a number that identifies the packet type, described below. The fourth byte is a sequence number that identifies each packet to prevent the receiver from

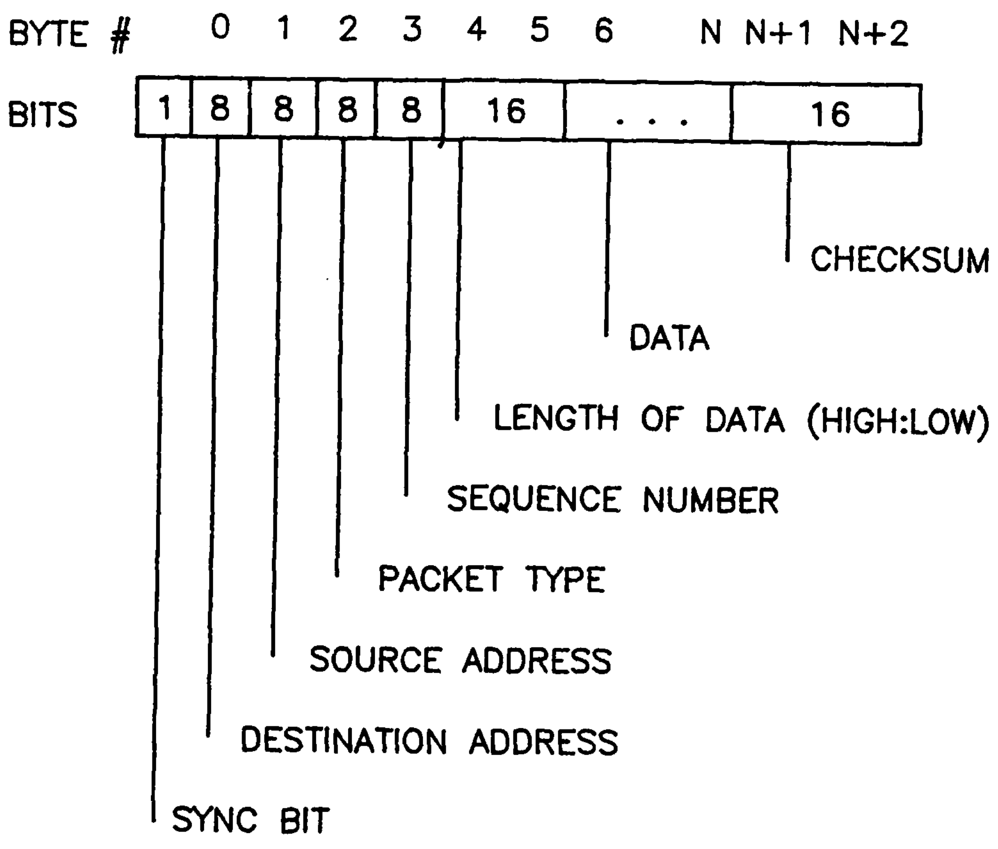


Figure 2. Packet format

accepting duplicates in the event that a packet is retransmitted. The remaining two bytes in the header contain the number of actual data bytes in the data portion of the packet, with the high order byte first. From zero to 1018 actual data bytes can be inserted in the packet making a maximum packet length of 1024 bytes (six header bytes plus 1018 data bytes). In addition, the controller hardware appends two bytes to the end of every packet. These two bytes provide an error-detecting checksum, but they are completely unavailable to the host computer's software. The "sync" bit is used by the hardware to synchronize the receiver at the start of the packet transmission and is also unavailable to the host.

There are seven different types of packets that are used in this system, four of which are used in this file transfer protocol and three that are concerned with the communication initiation protocol. These seven packet types were defined specifically for this network implementation and their meanings are summarized in Table 3. The ACKFILE type

Table 3. Packet Types

Type	Code	Description
ACKFILE	1	Packet received correctly
NAKFILE	2	Requested file on HP 3000 does not exist
DATAFILE	3	Packet contains file data
ENDFILE	5	Indicates end of file
ENDREPLY	6	Acknowledges the ACKFILE of the ENDFILE
SENDFILE	7	Prepares receiver to send a file
RECFILE	8	Prepares receiver to read a file

is an acknowledgement packet used by the receiver to acknowledge the correct reception of a packet. DATAFILE is used by the transmitter to identify packets that contain the actual file data. The ENDFILE and ENDREPLY packet types are used in the end sequence described below. The SENDFILE, RECFILE, and NAKFILE packet types are used to initiate communication between the two hosts and their use is described in the next chapter.

3.2. File Transfers

As mentioned above, files are transferred from the sender to the receiver as a series of consecutively numbered packets. This protocol is based on the "Ethernet File Transfer Protocol" described in [3] with some modifications as suggested in [10]. This system only allows one file at a time to be transferred between the two hosts. Packets transmitted between other stations can be multiplexed on the ether since each station controller ignores packets intended for other destinations. That is, there can be no multiplexing of file transfers to the same destination (but the ether can be multiplexed for transfers between different destinations) and there can be no multiplexing of file transfers if two or more of these transfers involve the VAX or the HP 3000 in any way.

After the initial communication is established between the two computers, this file transfer system is invoked to perform the actual transfer. An example of this protocol is shown in Figure 3. The sender sends DATAFILE packets consecutively numbered from one (modulo 256, since the sequence number field in the header is eight bits wide).

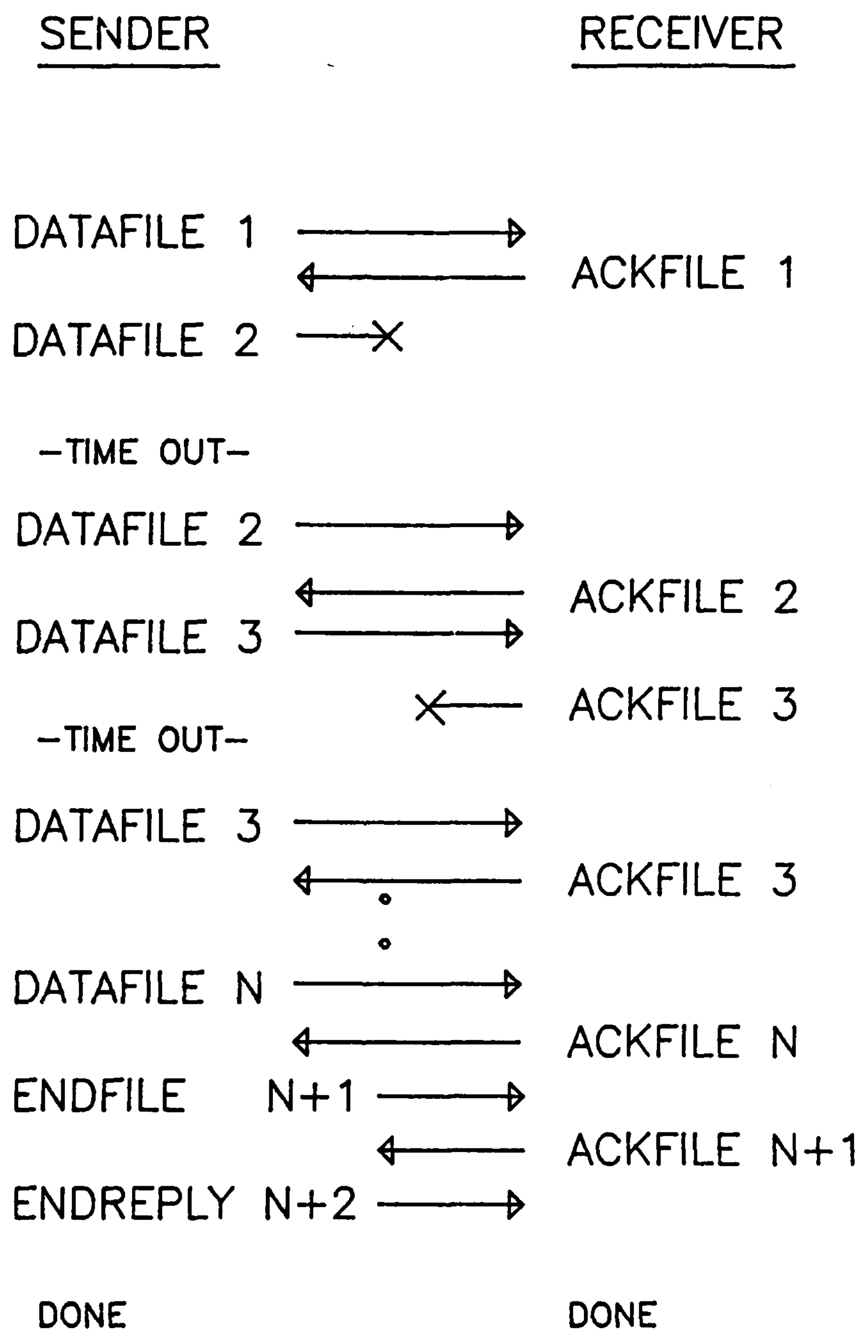


Figure 3. An example of the file transfer protocol

These DATAFILE packets have the format previously described with the "data" portion containing the actual file data. After each packet is transmitted, the sender waits for an ACKFILE packet from the receiver with a matching sequence number, thereby acknowledging the correct reception of the packet. These ACKFILE packets have an empty "data" portion and consist only of the header bytes and the controller-generated checksum. If an acknowledgement is not received within a set period of time (five seconds), the sender retransmits the same packet and again waits for the corresponding acknowledgement. The sending program will retransmit the same packet up to MAXCOUNT times, where MAXCOUNT is a parameter in the program that can be changed by recompiling the program. If there is no acknowledgement after MAXCOUNT transmission attempts, the sender simply assumes that the remote receiver is dead and aborts any further transmissions to that destination.

The receiver, meanwhile, waits for some packet to arrive on the net. Each packet is examined to make sure that it is undamaged and has the proper sequence number, that is, that the packet sequence number matches the expected sequence number. This expected number is simply one more than the sequence number of the last correctly received packet. If this is the case, the receiver accepts the packet, stores it in the file, acknowledges the packet, and increments the expected sequence number. If the packet is undamaged, but the sequence number is one less than the expected, then the packet is acknowledged and discarded. The assumption is that the previous acknowledgement for that packet was lost or damaged in transmission and needs to be retransmitted. All other packets (those that are damaged or have an improper sequence number) are simply ignored, the receiver knowing that the sender will shortly retransmit any packets that have not been acknowledged.

After all of the data has been transmitted, the sender sends an ENDFILE packet with the next consecutive sequence number to indicate to the receiver that the file transfer has been completed. This ENDFILE packet consists only of header bytes and the controller-generated checksum and has an empty "data" portion. The sender then waits for the receiver to acknowledge this ENDFILE with the standard ACKFILE packet. Upon receiving the ENDFILE, the receiver acknowledges it and then waits for a period of time. After receiving this acknowledgement, the sender transmits an ENDREPLY packet, which also has an empty "data" portion, and is now done with the entire file transfer. The waiting receiver receives this ENDREPLY and then it is also done with the file transfer.

This relatively complex end sequence makes it practically certain that both the sender and the receiver agree on whether the file has been transmitted correctly. If the ENDFILE packet is lost in transmission, the sender will simply time out and retransmit it as it would any other packet that has not been acknowledged. If the acknowledgement of this ENDFILE packet is lost, the sender will again simply time out and retransmit the ENDFILE. The packet will eventually be acknowledged by the waiting receiver. If the ENDREPLY packet from the sender is lost, the receiver will time out (five seconds). After it times out, it can quit and assume that the transfer was completed successfully since it previously received the ENDFILE packet.

4. COMMUNICATION INITIATION AND THE USER INTERFACE

The level of protocol described in this chapter is the highest level used in this local network system. This level does all of the communication with the user of the system and is concerned with opening and creating the required files by using the file handling procedures available within the UNIX operating system on the VAX and the MPE operating system on the HP 3000. It is also concerned with detecting and reporting to the user any errors that may occur due to problems in opening files, system crashes, improper usage of the file transfer commands, and other such problems. Within this protocol is a subprotocol that is concerned with initiating communication between the two computers to allow the transfer of files using the previously described file transfer protocol.

4.1. System Organization

The system organization requires the user to log in on the VAX to initiate file transfers both to and from the HP 3000. On the VAX, the user executes a command that runs a program to communicate with a similar program that is running continuously on the HP 3000. This program on the HP 3000 is called a network daemon and only responds to commands sent to it over the net. The term "daemon" is used by the developers of the UNIX operating system to refer to a program that is continuously waiting for the conditions to occur that cause it to go into action. This daemon runs in a special account called ETHERNET.SYS and uses the system's file handling procedures to store and retrieve files from any user's directory that has read, write, and save privileges specified as

ANY[11]. When the terminal port connected to the controller on the VAX is opened by running the file transfer command, it is allocated by UNIX exclusively to that user. This exclusivity allows only one VAX user at a time to transfer files.

Shown in Figure 4 is a schematic representation of this system organization. As this figure suggests, files are transferred between the user's directory on the VAX and the file system on the HP 3000. This file convention allows the user to specify the full path name on both computers and thereby place the file in any directory to which the user has access. For example, on the HP 3000 a fully specified name is of the form "FILENAME.GROUP.ACCOUNT." If the name is not fully specified, it will default to the ETHERNET.SYS account on the HP 3000 and can be transferred to the user's directory using the FCOPY command[11]. If the path name is not fully specified on the VAX, it defaults to the user's current working directory.

The file program on the VAX (see Appendix 3) must be linked to two command names after it is compiled using the "ln" command[7]. The two names to be linked to this program are "sendhp" (meaning send a file to HP 3000) and "gethp" (meaning get a file from the HP 3000). This naming is necessary because this single program uses the name with which it is called (sendhp or gethp) to determine the direction in which the files will be transferred.

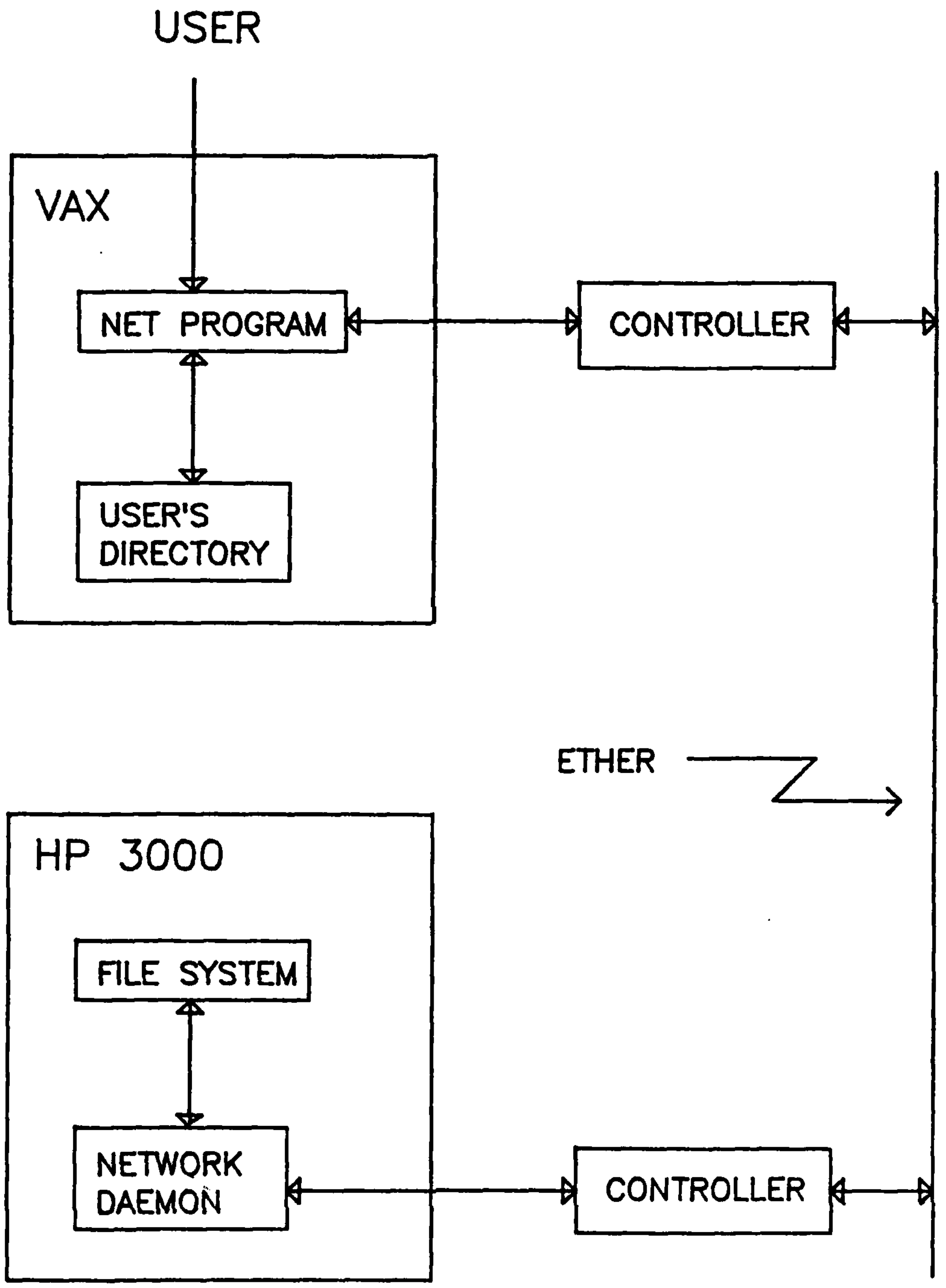


Figure 4. System organization for file transfers

4.2. File Transfers From the VAX to the HP 3000

Using the command "sendhp srcfile [destfile]" to call the VAX file program (see Appendix 3) will cause the file named in "srcfile" to be transferred to the HP 3000 over the network. The "destfile" is the name that the user wants the file to be called at the remote computer. The brackets indicate that "destfile" is an optional parameter. If it is omitted, the file name at the destination defaults to be the same as the source file name. If this command executes properly, the program simply ends and returns the system prompt. If it is unsuccessful in transferring the file after all attempts at retransmission, it will print an error message to the terminal indicating the nature of the problem.

The program executes the following sequence of events to process the "sendhp" command. It first checks the number of arguments to make sure all of the required arguments are present and to make sure that there are no additional arguments. The arguments for "sendhp" are sendhp itself and the words typed after the command on the same line, namely the source file name (srcfile) and, optionally, the destination file name (destfile). If the number of arguments is wrong, it prints an error message indicating the proper usage and terminates execution. If the number of arguments is correct, it checks the name with which it was invoked to determine the direction of the file transfer. Since in this case it was invoked with the name "sendhp," the direction of transfer is from the VAX to the HP 3000. Once the direction is known, the program assigns as the destination file name the third argument of the command, or it defaults to be the same as the source file name if no third argument is given. Next, the terminal port connected to the controller is opened and allocated to this user and the network station address of the controller is determined using the enquire (ENQ) command previously described. If the controller is presently allocated to another user, the program will terminate with a message indicating that the controller is busy. Otherwise, the program next attempts to open the source file. If it does not exist, the program terminates with an error message.

After this initialization, the program attempts to initiate communication with the network daemon running the HP 3000. This is done by sending a RECFILE packet which tells the network daemon to prepare to receive a file from the VAX. This packet has the same format as previously described with the data portion containing the destination file name, which is the name that the network daemon will use to save the file in the HP 3000 file system. Upon receiving the RECFILE packet, the network daemon opens a new file with the proper name and then sends an acknowledgement (ACKFILE) packet back to the VAX to indicate that it is ready to begin receiving the file. If for any reason the VAX does not receive the acknowledgement packet before timing out, it simply retransmits the RECFILE packet.

At this point, communication has been established between the two computers and they are ready to begin the actual file transfer. The file transfer protocol described in the previous chapter is then used to effect the transfer with the VAX as the sender and the HP 3000 as the receiver. After the end sequence is completed, the daemon on the HP 3000 attempts to close the just received file as a new permanent file and save it in the disc file system. If a file already exists in the directory with the same name, it will be unable to successfully close the file and will rename the new file so that it can be properly saved. This renaming is done automatically and the user is not notified of the change. It is done simply as a convenience so that the file will not have to be transferred again, but also will not overwrite a previously existing file. The renaming consists of simply appending a one digit number to the end of the file name to give a unique name to this new file. To get the correct (renamed) file, the user lists the contents of his directory (after logging on to the HP 3000) and looks for a file having the expected name plus the added single digit. The file with the largest added digit is the most recent version received over the net.

4.3. File Transfers From the HP 3000 to the VAX

To send files from the HP 3000 to the VAX the command "gethp srcfile [destfile]" is used to call the VAX file program, where "srcfile" and "destfile" have meanings as in the "sendhp" command. As previously mentioned, both of these commands must be linked to the same program. As when the program is called with "sendhp," "gethp" causes it to go through the same initialization sequence, first checking for the proper number of arguments, then determining the transfer direction and the destination file name, and finally opening the controller's port and determining the network station address of the controller. After this initialization, the program is ready to initiate communication with the network daemon on the HP 3000.

The first step in setting up communication with the network daemon is for the VAX to send a SENDFILE packet. This packet is formatted as previously described with the source file name in the "data" portion of the packet. After receiving this packet, the network daemon attempts to open the named file. If it cannot open the file, probably because it does not exist (perhaps the user spelled the name wrong), then it sends a NAKFILE packet back to the VAX. This NAKFILE is in the standard format with nothing in the "data" portion of the packet and consists only of header bytes and the controller-generated checksum. Upon receiving this negative acknowledgement, the VAX prints an error message to the user indicating that the file does not exist and terminates execution. If, however, the file exists and is opened successfully, the network daemon returns an ACKFILE packet to the VAX indicating that it is ready to transmit the file. Upon receiving this acknowledgement, the VAX creates a new file with the name in "destfile" (or "srcfile" if no "destfile" is specified) in which to store the incoming file from the network daemon. If a file already exists on the VAX with the same name as in "destfile," the program will append the suffix ".new" so as to avoid overwriting the old file. Once this communication is established and all of the appropriate files are opened and created, the file transfer protocol described in the previous chapter is used to transfer the files with the network daemon as the sender and the VAX as the receiver. After the file has been transferred, the program terminates and returns the system prompt to the user. Other error messages are similar to those for "sendhp."

4.4. The Network Daemon

The network daemon is the program that runs continuously on the HP 3000 listening for file transfer commands from the VAX sent over the net. The logic of this program closely follows the logic used in the program described above and a listing for the entire network daemon is included in Appendix 4. The main difference between this program and the "sendhp/gethp" program above is that this network daemon only needs to be started once when the HP 3000 is initially turned on or a cold start is executed. To start this program, the system operator needs to execute the command "STREAM STARTNET.LILJA.SYS" which will load in the network daemon and start it running in the account ETHERNET.SYS. Before this command is executed, the operator should manually reset the network controller. A listing of the file STARTNET.LILJA.SYS is shown in Appendix 5.

4.5. System Error Codes

There are several errors that can occur when using the "sendhp" and "gethp" commands that will cause the program to terminate. Besides printing a message on the user's terminal, the program also returns an error number using the "exit" system call. These error codes are summarized in Table 4. If the file was transferred successfully, the program will return 0 to indicate that no error occurred. Error code 1 means that the program was called with the incorrect number of arguments, either no source file name was given or too many file names were given. Codes 2 and 8 mean that the source file does not exist on the VAX or the HP 3000, respectively. This is usually due to the file name

Table 4. System Error Codes

Name	Code	Error Message
NOERROR	0	(No message is displayed.)
BADARGS	1	Usage is: sendhp srcfile [destfile] or Usage is: gethp srcfile [destfile].
BADOPENVAX	2	Error - can't open (filename) on VAX.
BADREMOTE	3	Remote machine does not respond.
CONTDOWN	4	Controller on VAX is dead. Try resetting it.
NOCONTOPEN	5	Couldn't open controller's port. It must be busy (or dead). Try again later.
NOCONFIG	6	Couldn't configure the controller's port.
XMITERROR	7	Some sent packets were not acknowledged. Try again.
BADOPENHP	8	Error - can't open (filename) on remote machine.

being spelled incorrectly. If the network daemon on the HP 3000 does not repond to a request from the VAX after the request has been transmitted a fixed number of times, the program will assume that the remote machine is dead and will return error code 3. If the network daemon acknowledges some, but not all of the transmitted packets, the program assumes that the daemon program must have crashed in the middle of the transfer and error code 7 is returned. In this case no (partial) file is stored in the receiver's file system if the HP 3000 is the receiver. However, a partial file could be stored in the VAX file system if the VAX is the receiver. The user may wish to try to transfer the file again. If a peculiar system state was left at the end of the first transfer, a new error message will indicate the problem.

Error codes 4 through 6 are concerned with the status of the network controller connected to the VAX. When the controller does not respond to the enquire command (ENQ), it must be dead and needs to be manually reset. This problem returns error code 4. Codes 5 and 6 are returned if the program is unable to open or configure the terminal port connected to the controller. The program will be unable to open the port (code 5) if someone else is already using the network, but the program should be able to open the port successfully after the other user is finished. If the program was able to open the port, the UNIX "stty" routine should also be able to configure the port. Error code 6 is returned if it cannot. These error codes are only intended to point the user in the proper direction and should not be taken as the last word in diagnosing the error.

4.6. Packet Size for File Transfers

The size of the packets have a significant effect on the efficiency of the file transfers. Larger packets are generally desired for faster transfers[12]. Due to a limited amount of buffer memory, the controller restricts the maximum packet size to 1024 bytes. When file transfers are attempted using this maximum size, it has been found that the file system on the HP 3000 is too slow to keep up with the packet traffic. That is, when transferring a file from the VAX to the HP 3000, the HP 3000 takes too long to store and acknowledge the packets, thereby causing the VAX to time out and retransmit many packets. The result is a degradation of the overall system performance.

Another more severe limitation is that the VAX will only buffer a maximum of 256 bytes from the controller terminal port on input. This limitation only becomes apparent when the VAX is heavily loaded. When it is lightly loaded, the terminal driver appears to have enough time to read all of the characters being sent by the controller, but characters that overflow the buffer under heavy VAX loading are simply thrown away. When these characters are discarded, the file transfer program waits indefinitely for the discarded characters to be transferred by the controller. The only way to recover from this problem is to restart the program and manually reset the controller.

The result of these limitations is that for all file transfers a packet size of no more than 256 bytes should be used. This maximum size provides for reasonably efficient transfers and still assures proper buffering and timing on both of the host computers.

5. CONCLUSION

We have described the implementation of a local computer network to allow the transfer of files from one host computer to another. The transfer is accomplished by logging in on the VAX computer and executing one of two simple commands to transfer a file to or from the HP 3000 computer. The four distinct levels of protocol used in this implementation were described. The lowest level protocol, the Ethernet protocol, controls the actual transmission of packets over the ether and is implemented in a previously built microprocessor based controller. This level of protocol delivers packets to their destinations with a high probability of success, but any error control must be done at a higher level.

The remaining three levels of protocol represent the main work of this thesis and are implemented entirely in software on the host computers. The host to controller communication protocol, which allows the host computer to control the slave Ethernet controller, was the most difficult to implement due to its dependency on the unique characteristics of the terminal drivers of the two hosts.

The next level of protocol is the file transfer protocol which implements the transfer of files as a series of sequentially numbered packets. This protocol is also tolerant of transmission errors that destroy individual packets by providing for the retransmission of packets that have not been acknowledged after a fixed interval of time.

The highest level of protocol in this local network communicates with the user and initiates the communication between the network program running on the VAX and the network daemon that runs continuously on the HP 3000.

5.1. Suggestions for Improvement

While the system works as it was intended and provides a simple mechanism to transfer files between the two computers, there are some things that could be done to improve its overall performance and usefulness. One main disadvantage of the current implementation is the relatively slow speed of large file transfers. The main bottleneck is the 9600 baud serial communication line connecting the controllers to their host computers through terminal ports. A much better strategy would employ a direct memory access (DMA) connection between each controller and its host computer. Even a byte parallel connection between the controller and the host would be preferable to the serial connection now used. One problem with a DMA or a parallel connection scheme is that it would be unique to each type of host and the controller would lose some of its portability.

Another improvement to the controller would be to simplify the host to controller communication protocol. As it is now, this protocol is complex and difficult to use. It would be nice if the controller would listen to the ether continuously and be able to notify the host whenever a packet arrived. This would eliminate the need for the daemon on the host and would make the host's network software much simpler.

There are some other improvements to the system that could be implemented totally in software on the host computer. One of these is to allow the multiplexing of several different simultaneous file transfers from several users. This multiplexing, however, would cause ambiguity in packet sequence numbers since these numbers are sequential only within a single file transfer. To resolve this ambiguity, an extra field would have to be added to the header portion of each packet for a "file number." This file number, together with the packet sequence number, would uniquely identify each packet of each file transfer in process.

Another improvement is to allow a user to log in on either host computer and transfer files either to or from the remote host. This would require a network daemon on both hosts to listen for commands on the net and some mechanism to allow a user logged in on the host to pass a command to the local daemon. The implementation of this mechanism does not appear to be straightforward and would probably be a nontrivial task.

One other useful improvement is to allow a user to log in on the remote machine through the network. For example, a user could log in on the VAX using a modem through a dial-up line and then use the network to log in on the HP 3000 as a normal user. This would require an intelligent process to be running on the remote host to simulate a terminal or it would require a substantial modification to the software in the Ethernet controller. Any modifications to the controller would probably make it no longer portable, which may be undesirable.

Anyone attempting to modify this system should look carefully at the program listings in the Appendices. Many of these routines, such as the routines to send and receive individual packets, could probably be used unchanged in any future enhancements. In any case, they should provide a guide for implementing many of the fine details such as the complex host to controller communication protocol. While this system is complete as it is, it is hoped that it will provide a starting point for a larger local network.

Appendix 1

The PACKET program for the VAX

```
/* program to send and receive packets over the Ethernet  */



/* include required libraries    */

#include <stdio.h>            /* standard i/o library */
#include <sgtty.h>            /* library for tty set-ups  */

/* define constants  */

#define TTY  "/dev/tty07" /* port to connect controller to */
#define MAXLINE  510      /* maximum length of input line */
#define PACKETSIZE 512    /* maximum size of packets */
#define MAXCOUNT 16       /* maximum number of tries before aborting
                             transmission attempts                */
#define OK  1             /* flag indicates transmission was OK   */
#define NOTOK 0           /* flag indicates transmission was not OK */


/* define command bytes     */

#define ACK       0x06     /* acknowldgement - all ok */
#define NAK       0x15     /* negative acknowledge - error has occured */
#define ENQ       0x05     /* enquiry command  */
#define SEND      0x11     /* send the following packet */
#define SOH       0x01     /* start of header - device sending SOH is
                              requesting the packet length          */
#define REC       0x12     /* receive packet from Ethernet  */
#define CAN       0x18     /* cancel last receive request  */

/* define golbal variables */

char destaddress;     /* address of destination controller */
char srcaddress;      /* address of source (this) controller */
int  controller;      /* file descriptor of controller port  */




/* main program to get and process commands  */
```

```
main ()
{
      int  command;       /* command to be executed */

      printbanner();      /* print banner, version number */
      initialize();       /* init station addresses */
      help();             /* print list of commands */

      /* execute commands until quit */

      while (((command = getcommand()) != 'q') && (command != 'Q'))

            switch (command)
            {     case 's':      /* send a packet */
                  case 'S':
                     send();
                     break;
                  case 'r':      /* receive a packet */
                  case 'R':
                     receive();
                     break;
                  case 'e':
                  case 'E':      /* enquire - print address */
                     enquire();
                     break;
                  case 'c':      /* cancel last receive */
                  case 'C':      /*   command          */
                     cancel();
                     break;
                  case 'h':      /* help - print list of */
                  case 'H':      /* these commands       */
                     help();
                     break;
                  default:      /* invalid command */
                     printf("\ninvalid command\n");
                     break;
            }
      /* print quitting and end */

      printf("\nquitting...\n");

}




/* send command  read a packet from the terminal and send it
   over the ethernet.  assumes addresses are initialized in
   srcaddress and destaddress                             */
```

```
send()
{
     int  length;      /* length of data input from terminal */
     char packet[PACKETSIZE];      /* completed packet to send.
                                      dest : src : data        */
     char data[MAXLINE];  /* data read from terminal */
     int  i,j;            /* indices in loop  */
     int  count = 1;      /* number of attempts to xmit so far */

     /* read the packet from the terminal */

     printf("Enter the packet terminated by <ret>:\n\n");
     length = getline(data, MAXLINE);

     /* create the packet, ie. add addresses to start */

     packet[0] = destaddress;
     packet[1] = srcaddress;
     for (i=2, j=0;  j < length ;  ++i, ++j)
           packet[i] = data[j];

     /* try to send packet until OK or too many tries  */

     while (  (sendpacket(packet,length+2) != OK)
           && (count < MAXCOUNT)   )
           ++count;

     /* print message about transmission */

     if (count >= MAXCOUNT)
           printf("\npacket not transmitted after %2d tries\n",count);
     else printf("\npacket transmitted OK after %2d tries\n",count);
}




/* sendpacket - send a single packet.  addresses are embedded within
   the packet.  returns OK if good transmission, NOTOK otherwise     */

sendpacket(packet,length)

 char      packet[];      /* packet to be sent */
 int      length;         /* length of packet  */
{
     char reply;      /* controller's replies to host */
     char status;     /* status byte from controller */
     int  xmitflag = NOTOK;     /* hold OK or NOTOK */
     char len_low;    /* low-order byte of length */
```

```
        char len_high;     /* high-order byte of length */
        int  i;            /* index in loop */

        /* split length into high- and low-order parts */

        len_high = length / 256;      /* truncates fraction */
        len_low = length - (len_high * 256);

        /* ready controller to xmit */

        sendcontroller(SEND);
        reply = readcontroller();
        status = readcontroller();

        if (reply == SOH)      /* controller ready */
        {
              sendcontroller(len_low);
              sendcontroller(len_high);
              reply = readcontroller();
              status = readcontroller();

              if (reply == ACK)   /* controller ready */
              {
                    for (i=0;  i<length;  ++i)   /* send packet */
                          sendcontroller(packet[i]);
                    reply = readcontroller();
                    status = readcontroller();

                    if (reply == ACK)    /* it worked */
                          xmitflag = OK;
              }
        }
        return(xmitflag);
  }




  /* receive command - receive a packet from the ethernet and print it
     on the terminal                                                 */

  receive()
  {
        int  length;       /* will contain length of packet recv'd  */
        char packet[PACKETSIZE];      /* recv'd packet  */
        int  i;            /* index in loop */
        /* print message and receive the packet  */

        printf("\nWaiting for packet on the ethernet...\n");

        if (recvpacket(packet,&length) == OK)
        {
             /* print addresses, print the packet  */
             printf("packet received OK\n\n");
             printf("Destination address:  %2x \n",packet[0]);
             printf("Source address: %2x \n\n",packet[1]);
             for (i=2;  i < length;  ++i)
                  putchar(packet[i]);
             printf("\n\n");
        }
        else printf("\npacket damaged.  receive cancelled\n");
}




/* recvpacket - receives a packet from the ethernet with this
   stations's address or the broadcast address.
   returns OK if recv'd OK, NOTOK otherwise.
   values of packet and length returned via pointers
   cancels receive command if not received OK                  */

recvpacket(packet,plength)

 char packet[];             /* contents of packet received  */
 int  *plength;             /* pointer to length.  Returns length of
                                packet to calling routine            */
{
        char reply;      /* controller's replies to host    */
        char status;     /* status byte from controller     */
        char len_low;    /* low-order byte of length        */
        char len_high;   /* high-order byte of length       */
        int  i;          /* index in loop               */
        int  rcvflag = NOTOK;      /* holds OK or NOTOK        */

        /* tell controller to receive packet  */

        sendcontroller(REC);
        reply = readcontroller();
        status = readcontroller();

        /* check if packet recv'd OK   */

        if (reply == ACK)   /* packet OK  */
        {
            /* read length */
            sendcontroller(SOH);
            len_low = readcontroller();
            len_high = readcontroller();
            sendcontroller(ACK);

            /* read the packet */
            *plength = (256 * len_high) + len_low;
            for (i=0;  i < *plength;  ++i)
                  packet[i] = readcontroller();
            sendcontroller(ACK);
            rcvflag = OK;
      }
      else cancel();              /* not recv'd OK  */

      return(rcvflag);
}
```

```
/* enquire - print ethernet station address
      also re-initializes srcaddress          */


enquire()
{
      /*  get address, print it */

      srcaddress = findaddress();
      printf("This station address is %2x H\n", srcaddress);

}
```

```
/* cancel - send cancel command to controller */

cancel()
{
        char reply;      /* controller's reply to CAN */
        char status;     /* also sends status byte    */

        /* send CAN, read reply and status */

        sendcontroller(CAN);
        reply = readcontroller();
        status = readcontroller();
        printf("\ncancelled\n");
}




/* help - print menu of available commands    */

help()
{
        printf("\n");
        printf("s - send a packet\n");
        printf("r - receive a packet\n");
        printf("e - enquire:  print this station's ethernet address\n");
        printf("c - cancel receive command\n");
        printf("h - help:  print this menu\n");
        printf("q - quit\n\n\n");
}




/* initializes source and destination addresses.
    source address is determined by enquiry command.
    destination address is prompted from user.          */

initialize()
{

        /* initialize controller's i/o port  */

        initcontroller();

        /* do ENQ command to init and print this station's address */

        enquire();


        /* read dest. address from terminal */
        printf("Enter destination address <00 - FF> ");
        scanf("%2x", &destaddress);
        printf("Destination address is %2x\n\n",destaddress);


}




/* find the ethernet station address of the controller */
/* uses the ENQ command.  returns value of address     */

findaddress()
{
        char  status;       /* status byte returned from controller */

        /* send ENQ to controller  */

        sendcontroller(ENQ);

        /* read status, read and return address  */

        status = readcontroller();
        return(readcontroller());
}






/* print heading and version number */

printbanner()
{
        printf("\n\n\n\n");
        printf("Ethernet packet transceiver test\n");
        printf("   Version 1.0\n\n\n");
}
```

```
/* getcommand - read command from terminal
    returns single character corresponding to the command */

getcommand()
{
      char inputline[MAXLINE];      /* line read from terminal */

      /* print prompt, read line, return first character */

      printf("command> ");
      getline(inputline,MAXLINE);
      printf("\n");
      return(inputline[0]);
}




/* getline - from "c" by Kernighan and Ritchie, p. 26  */

getline(s,lim)      /* returns length of line read in */

 char     s[];      /* line that is read in */
 int      lim;      /* max. size of line to be read */
{
      int c, i;

      for (i=0; i<lim-1 && (c=getchar() ) != EOF && c != '\n'; ++i)
            s[i] = c;
      if (c == '\n')
      {      s[i] = c;
            ++i;
      }
      s[i] = ' ';
      return(i);
}




/* initcontroller - open and init. tty port to talk to controller  */

initcontroller()
{
      /* structure to initialize tty port to controller   */

      static      struct      sgttyb  ttyb = {
            B9600, B9600,
            0, 0,
            RAW | ANYP
      };



      /* open controller's tty port  */

      if ( (controller = open(TTY,2)) == -1)  /* couldn't open */
            printf("Couldn't open controller's port\n");

      else  /* opened OK so configure port for RAW, 9600 baud */
        if (stty(controller,&ttyb) != 0) /* couldn't configure */
            printf("Couldn't configure controller's port\n");
}






/* sendcontroller - send a byte to controller  */

sendcontroller(byte)

 char byte;      /* byte to be sent  */
{
      /* send the byte, print error if not sent OK  */

      if (write(controller, &byte, 1) != 1)
            printf("Can't write controller\n");
}



/* readcontroller - returns byte read from controller  */

readcontroller()
{
      char byte;      /* byte read in */

      /* read byte, print error if not read OK  */

      if (read(controller, &byte, 1) != 1)
```

```
                printf("Can't read controller\n");
        else return(byte);
}
```

Appendix 2

The PACKET program for the HP 3000

```
$CONTROL USLINIT
BEGIN
<< PROGRAM TO SEND AND RECEIVE PACKETS OVER ETHERNET >>


<< GLOBAL VARIABLE DECLARATIONS >>

  INTEGER     CONTIN;           << FILE # OF CONTROLLER INPUT PORT >>
  INTEGER     CONTOUT;          << FILE # OF CONTROLLER OUTPUT PORT >>
  BYTE        DESTADDRESS;      << ADDRESS DESTINATION CONTROLLER >>
  BYTE        SRCADDRESS;       << ADDRESS SOURCE CONTROLLER >>
  BYTE ARRAY  MSGB(0:80);       << I/O BUFFER FOR TERMINAL >>
  LOGICAL ARRAY MSG(*)=MSGB(0);<< PRINT BUFFER -- "WORD" EQUATE >>
  BYTE        COMMAND;          << COMMAND TO BE EXECUTED  >>


<< DEFINE COMMAND BYTES TO/FROM CONTROLLER >>

  EQUATE ACK  = %(16)06,  << ACKNOWLEDGEMENT -- ALL OK >>
         NAK  = %(16)15,  << NEGATIVE ACK - ERROR HAS OCCURED >>
         ENQ  = %(16)05,  << ENQUIRE - RETURN STATUS & ADDRESS >>
         SND  = %(16)11,  << SEND THE FOLLOWING PACKET >>
         SOH  = %(16)01,  << START OF HEADER - DEVICE SENDING IS >>
                          << REQUESTING THE PACKET LENGTH        >>
         REC  = %(16)12,  << RECEIVE PACKET FROM ETHERNET >>
         CAN  = %(16)18;  << CANCEL LAST RECEIVE REQUEST >>


<< DEFINE CONSTANTS >>

  EQUATE PACKETSIZE = 128, << MAXIMUM SIZE OF PACKETS >>
         MAXLINE    = 80,  << MAXLINE LENGTH OF INPUT LINE >>
         MAXCOUNT   = 16,  << MAXIMUM NUMBER OF TRIES BEFORE  >>
                           << ABORTING TRANSMISSION ATTEMPT   >>
         OK         = 1,   << FLAG FOR "OK"  >>
         NOTOK      = 0;   << FLAG FOR "NOTOK"  >>




<< INTRINSIC DECLARATIONS >>

  INTRINSIC READ,PRINT,BINARY,FREAD,FWRITE,FOPEN,FSETMODE,
            FCONTROL,IOWAIT,GETPRIVMODE,GETUSERMODE,ASCII;




<<........  PROCEDURE DECLARATIONS ...........>>
```

```
<< PRINTBANNER - PRINT HEADING AND VERSION NUMBER >>

PROCEDURE PRINTBANNER;
  BEGIN
    PRINT(MSG,0,%204);    << FOUR BLANK LINES >>
    MOVE MSGB := "ETHERNET PACKET TRANSCEIVER TEST";
    PRINT(MSG,-32,%40);
    MOVE MSGB := "    VERSION 1.0";
    PRINT(MSG,-15,%40);
    PRINT(MSG,0,%203);    << THREE BLANK LINES >>
  END;   << PRINTBANNER >>




<< RWCNTLR - READ/WRITE FROM CONTROLLER >>

 COMMENT: BECAUSE OF THE LACK OF BUFFERING ON INPUT PORTS,
          MUST FIRST SET-UP READ, THEN WRITE, THEN FINISH
          READ.  THAT IS, CONTROLLER RESPONDS TOO QUICKLY
          FOR HP TO CATCH IN NORMAL I/O ;

PROCEDURE RWCNTLR(INBUFFB,INLEN,OUTBUFFB,OUTLEN);

  VALUE OUTLEN,           << PASS THESE TWO BY VALUE >>
        INLEN;

  BYTE ARRAY INBUFFB;     << INPUT BUFFER TO CONTROLLER >>
                          << RETURNS WITH DATA READ FROM CONTROLLER >>
  INTEGER INLEN;          << NUMBER OF BYTES TO READ FROM CONT >>
  BYTE ARRAY OUTBUFFB;    << OUTPUT BUFFER TO CONTROLLER >>
                          << ENTER WITH DATA TO SEND TO CONT >>
  INTEGER OUTLEN;         << NUMBER OF BYTES TO SEND TO CONT >>


  BEGIN
    LOGICAL ARRAY INBUFF(*)=INBUFFB(0); << EQUATE TO ELIMINATE >>
    LOGICAL ARRAY OUTBUFF(*)=OUTBUFFB(0);  << WARNING MESGS >>
    INTEGER DUMMY;  << HOLDS RETURNED VALUE FROM IOWAIT, FREAD >>

    << SET-UP READ, NO WAIT FOR FINISH >>
    DUMMY := FREAD(CONTIN,INBUFF,-INLEN);

    << WRITE BUFFER TO CONTROLLER >>
    FWRITE(CONTOUT,OUTBUFF,-OUTLEN,%320);

    << FINISH READ, RETURN BUFFER AND LENGTH >>
    DUMMY := IOWAIT(CONTIN,INBUFF);
  END;  << RWCNTLR >>




<< FIND ETHERNET STATION ADDRESS OF THE CONTROLLER USING ENQ   >>
<< COMMAND.  RETURNS ADDRESS IN 'ADDRESS' PARAMETER.          >>

PROCEDURE FINDADDRESS(ADDRESS);

  BYTE ADDRESS;    << RETURNS ETHERNET STATION ADDRESS >>

  BEGIN
    BYTE ARRAY BUFF(0:1);  << I/O BUFFER FOR CONTROLLER >>

    << LOAD 'ENQ' INTO BUFFER >>
    BUFF(0) := ENQ;

    << SEND TO CONTROLLER, READ STATUS AND ADDRESS >>
    RWCNTLR(BUFF,2,BUFF,1);

    << RETURN ADDRESS >>
    ADDRESS := BUFF(1);

  END;  << FINDADDRESS >>




<< ENQUIRE - GET ADDRESS, PRINT IT ON TERMINAL >>

PROCEDURE ENQUIRE;

  BEGIN
    INTEGER LENGTH;  << NUMBER OF CHARS RETURNED BY ASCII >>

    << GET ADDRESS >>
    FINDADDRESS(SRCADDRESS);  << LEAVES VALUE IN SRCADDRESS >>

    << PRINT ADDRESS >>
    MOVE MSGB := "THIS STATION ADDRESS IS ";
    PRINT(MSG,-24,%320);
    LENGTH := ASCII(LOGICAL(SRCADDRESS),10,MSGB);  << CONVERT >>
    PRINT(MSG,-LENGTH,%40);

  END;  << ENQUIRE >>
```

```
<< INITIALIZE CONTROLLER'S PORTS, INIT STATION ADDRESSES >>

PROCEDURE INITIALIZE;

  BEGIN
    INTEGER LENGTH;  << TEMP STORAGE OF RETURNED VALUE FROM READ >>
    INTEGER CNTLCODE; << CONTROL CODE FOR FCONTROL >>
    BYTE ARRAY TTYIN(0:8);  << INPUT PORT >>
    BYTE ARRAY TTYOUT(0:8);  << OUTPUT PORT >>

    << INIT TTYIN AND TTYOUT >>

    MOVE TTYIN := "ENETIN;";
    MOVE TTYOUT := "ENETOUT;";

    << OPEN AND INIT CONTROLLER'S PORTS >>

    GETPRIVMODE;   << FOR NO-WAIT I/O >>
    CONTIN := FOPEN( ,%604,%4324,-36,TTYIN);
    IF = THEN BEGIN  << PRINT OK >>
      MOVE MSGB := "INPUT OPENED OK";
      PRINT(MSG,-15,%40);
     END;
    GETUSERMODE;   << NO MORE NEED FOR PRIV MODE >>

    CONTOUT := FOPEN( ,%604,%324,-36,TTYOUT);
    IF = THEN BEGIN
      MOVE MSGB := "OUTPUT OPENED OK";
      PRINT(MSG,-16,%40);
     END;

    CNTLCODE := %074022;   << SPEED, TERMTYPE >>
    FCONTROL(CONTIN,37,CNTLCODE);  << ALLOCATE TERMTYPE 18 AT >>
    FCONTROL(CONTOUT,37,CNTLCODE); << 9600 BAUD              >>

    FSETMODE(CONTIN,%4);   << INHIBIT AUTO CR-LF ON INPUT >>

    FCONTROL(CONTIN,13,CNTLCODE);  << ECHO OFF >>
    FCONTROL(CONTOUT,13,CNTLCODE);

    FCONTROL(CONTIN,20,CNTLCODE);  << DISABLE INPUT TIMER >>

    FCONTROL(CONTIN,28,CNTLCODE);  << DISABLE BLOCK MODE >>
    FCONTROL(CONTOUT,28,CNTLCODE);

    FCONTROL(CONTIN,27,CNTLCODE);  << ENABLE BINARY TRANSFERS >>
    FCONTROL(CONTOUT,27,CNTLCODE);

    CNTLCODE := 0;
    FCONTROL(CONTIN,36,CNTLCODE);  << NO PARITY, FULL 8 BITS >>
    FCONTROL(CONTOUT,36,CNTLCODE);


    << DO ENQ COMMAND TO INIT SRCADDRESS AND PRINT IT >>

    ENQUIRE;

    << READ DESTINATION ADDRESS FROM TERMINAL >>

    MOVE MSGB := "ENTER DESTINATION ADDRESS <0 - 255> ";
    PRINT(MSG,-36,%320);
    LENGTH := READ(MSG,-3);
    DESTADDRESS := BYTE(BINARY(MSGB,LENGTH));
    PRINT(MSG,0,%203);   << THREE BLANK LINES >>
  END;  << INITIALIZE >>




<< HELP - PRINT LIST OF COMMANDS >>

PROCEDURE HELP;

  BEGIN
    PRINT(MSG,0,%201);  << BLANK LINE >>
    MOVE MSGB := "S - SEND A PACKET";
    PRINT(MSG,-17,%40);
    MOVE MSGB := "R - RECEIVE A PACKET";
    PRINT(MSG,-20,%40);
    MOVE MSGB := "E - ENQUIRE:  PRINT THIS STATION'S ADDRESS";
    PRINT(MSG,-42,%40);
    MOVE MSGB := "C - CANCEL RECEIVE COMMAND";
    PRINT(MSG,-26,%40);
    MOVE MSGB := "H - HELP:  PRINT THIS LIST";
    PRINT(MSG,-26,%40);
    MOVE MSGB := "Q - QUIT";
    PRINT(MSG,-8,%40);
    PRINT(MSG,0,%201);    << BLANK LINE >>
  END;  << HELP >>




<< ERROR - MESSAGE FOR INVALID COMMAND >>
```

```
PROCEDURE ERROR;

  BEGIN
    MOVE MSGB := "***ERROR - NO SUCH COMMAND";
    PRINT(MSG,-26,%40);
  END;  << ERROR >>




<< SENDPACKET - SEND A SINGLE PACKET.  ADDRESSES ARE EMBEDDED  >>
<< WITHIN THE PACKET.  RETURNS XMITFLAG = OK IF GOOD           >>
<< TRANSMISSION, NOTOK OTHERWISE.  LEAVES PACKET UNCHANGED.    >>

PROCEDURE SENDPACKET(PACKET,LENGTH,XMITFLAG);

  VALUE LENGTH;          << PASS BY VALUE >>

  BYTE ARRAY PACKET;     << PACKET TO BE SENT WITH EMBEDDED ADDRESS >>
  INTEGER LENGTH;        << LENGTH OF PACKET IN BYTES >>
  INTEGER XMITFLAG;      << RETURNS 'OK' OR 'NOTOK'  >>

  BEGIN
    BYTE ARRAY BUFF(0:1);  << BUFFER FOR SENDING COMMANDS   >>
                           << TO/FROM CONTROLLER            >>

    << INIT XMITFLAG >>
    XMITFLAG := NOTOK;

    << READY CONTROLLER TO XMIT BY SENDING 'SND'.    >>
    << READ REPLY AND STATUS                         >>

    BUFF(0) := SND;
    RWCNTLR(BUFF,2,BUFF,1);   << SEND 1 BYTE, READ 2 BYTES >>

    IF BUFF(0) = SOH   << REPLY IS SOH >>
      THEN BEGIN    << SEND LENGTH, READ REPLY AND STATUS >>
        BUFF(0) := BYTE(LENGTH.(8:8));  << LOW-ORDER 8 BITS >>
        BUFF(1) := BYTE(LENGTH.(0:8));  << HIGH-ORDER 8 BITS >>
        RWCNTLR(BUFF,2,BUFF,2);  << READ/WRITE 2 BYTES >>

        IF BUFF(0) = ACK  << CONTROLLER READY FOR PACKET >>
          THEN BEGIN   << SEND PACKET, READ REPLY, STATUS >>
            RWCNTLR(BUFF,2,PACKET,LENGTH);
            IF BUFF(0) = ACK  << XMIT OK >>
              THEN XMITFLAG := OK;
          END;
      END;
  END;  << SENDPACKET >>




<< SEND - READ A PACKET FROM THE TERMINAL AND SEND IT OVER THE   >>
<< ETHERNET.  WILL TRY TO SEND PACKET A MAXIMUM OF 'MAXCOUNT'    >>
<< TIMES.  ASSUMES ADDRESSES ARE INTITIALIZED IN 'SRCADDRESS'    >>
<< AND 'DESTADDRESS'.                                            >>

PROCEDURE SEND;

  BEGIN
    BYTE ARRAY PACKET(0:PACKETSIZE); << COMPLETED PACKET TO SEND  >>
                                     << WITH DEST:SRC:DATA FORMAT >>
    INTEGER LENGTH;    << NUMBER OF BYTES READ FROM TERMINAL  >>
    INTEGER COUNT;     << NUMBER OF ATTEMPTS TO XMIT SO FAR >>
    INTEGER XMITFLAG;  << INDICATES IF PACKET XMITTED OK OR NOT >>
    INTEGER I;         << INDEX IN LOOP >>
    INTEGER TEMP;      << USED IN ASCII CONVERSION FOR PRINTING >>

    << INIT COUNT >>
    COUNT := 0;

    << READ PACKET FROM THE TERMINAL >>

    MOVE MSGB := "ENTER PACKET TERMINATED BY <RET>:  ";
    PRINT(MSG,-35,%40);
    LENGTH := READ(MSG,-MAXLINE);

    << CREATE PACKET, IE. ADD ADDRESSES TO START  >>

    PACKET(0) := DESTADDRESS;
    PACKET(1) := SRCADDRESS;
    FOR I := 0 UNTIL LENGTH - 1
      DO PACKET(I+2) := MSGB(I);

    << TRY TO SEND PACKET UNTIL OK OR TOO MANY TIMES >>

    DO BEGIN
      SENDPACKET(PACKET,LENGTH+2,XMITFLAG);
      COUNT := COUNT + 1;
     END
    UNTIL XMITFLAG = OK   OR   COUNT >= MAXCOUNT;

    << PRINT MESSAGE ABOUT TRANSMISSION  >>

    IF XMITFLAG = OK
      THEN BEGIN
```

```
        MOVE MSGB := "PACKET TRANSMITTED OK AFTER ";
        PRINT(MSG,-28,%320);
        TEMP := ASCII(COUNT,10,MSGB);
        PRINT(MSG,-TEMP,%320);
        MOVE MSGB := " TRIES";
        PRINT(MSG,-6,%40);
      END  << THEN >>

      ELSE BEGIN
        MOVE MSGB := "***--PACKET NOT TRASMITTED OK";
        PRINT(MSG,-27,%40);
      END;  << ELSE >>

  END;  << SEND >>




<< CANCEL - SEND CANCEL COMMAND TO CONTROLLER >>

PROCEDURE CANCEL;

  BEGIN
    BYTE ARRAY BUFF(0:1);  << BUFFER FOR COMMANDS TO/FROM CONT >>

    << SEND 'CAN', READ REPLY AND STATUS >>

    BUFF(0) := CAN;
    RWCNTLR(BUFF,2,BUFF,1);   << SEND 1 BYTE, READ 2 >>
    MOVE MSGB := "CANCELLED";
    PRINT(MSG,-9,%40);
  END;  << CANCEL >>




<< RECVPACKET - RECEIVES A PACKET FROM THE ETHERNET WITH THIS   >>
<< STATION'S ADDRESS OR THE BROADCAST ADDRESS.  RETURNS WITH    >>
<< RCVFLAG = OK IF RECEIVED OK, NOTOK OTHERWISE.  CANCELS       >>
<< RECEIVE COMMAND (WITH 'CAN') IF NOT RECEIVED OK.             >>

PROCEDURE RECVPACKET(PACKET,LENGTH,RCVFLAG);

  BYTE ARRAY PACKET;  << RETURNS WITH PACKET RECEIVED >>
  INTEGER LENGTH;     << RETURNS LENGTH OF RECEIVED PACKET >>
INTEGER RCVFLAG;    << RETURNS 'OK' OR 'NOTOK'  >>

BEGIN
  BYTE ARRAY BUFF(0:1);  << BUFFER FOR SENDING COMMANDS   >>
                         << TO/FROM CONTROLLER            >>

    << INIT. FLAG >>
    RCVFLAG := NOTOK;

    << TELL CONTROLLER TO RECEIVE PACKET.  WAITS UNTIL PACKET >>
    << IS RECEIVED.                                           >>

    BUFF(0) := REC;
    RWCNTLR(BUFF,2,BUFF,1);     << SEND 1 BYTE, READ 2 BYTES  >>

    << CHECK IF PACKET RECEIVED OK >>

    IF BUFF(0) = ACK       << PACKET OK  >>
      THEN BEGIN
        << SEND 'SOH', READ LENGTH >>
        BUFF(0) := SOH;
        RWCNTLR(BUFF,2,BUFF,1);   << SEND 1 BYTE, READ 2 BYTES >>
        LENGTH := 256 * BUFF(1) + BUFF(0);

        << SEND 'ACK', READ PACKET >>
        BUFF(0) := ACK;
        RWCNTLR(PACKET,LENGTH,BUFF,1);

        << SEND CONTROLLER FINAL 'ACK'  >>
        BUFF(0) := ACK;
        RWCNTLR(BUFF,0,BUFF,1);   << SEND 1 BYTE, READ NO BYTES >>

        << RETURN FLAG 'OK' >>
        RCVFLAG := OK;
      END   << THEN >>

      ELSE CANCEL;   << NOT RECEIVED OK, SO CANCEL 'REC' >>

END;   << RECVPACKET >>




<< RECEIVE - RECEIVE A PACKET ON THE ETHERNET AND PRINT IT ON    >>
<< THE TERMINAL                                                  >>

PROCEDURE RECEIVE;

  BEGIN
```

```
    BYTE ARRAY PACKET(0:PACKETSIZE);  << WILL CONTAIN RECV'D PACKET >>
    INTEGER LENGTH;     << LENGTH OF RECEIVED PACKET >>
    INTFGER RCVFLAG;    << INDICATES IF PAKCET RECEIVED OK >>
    INTEGER TEMP;       << USED IN ASCII CONVERSION FOR PRINTING >>

    << PRINT MESSAGE AND RECEIVE THE PACKET >>

    MOVE MSGB := "WAITING FOR PACKET ON THE ETHERNET...";
    PRINT(MSG,-37,%40);
    RECVPACKET(PACKET,LENGTH,RCVFLAG);

    IF RCVFLAG = OK
      THEN BEGIN   << PRINT ADDRESSES AND PACKET >>
        MOVE MSGB := "PACKET RECEIVED OK";
        PRINT(MSG,-18,%202);
        MOVE MSGB := "DESTINATION ADDRESS: ";
        PRINT(MSG,-21,%320);
        TEMP := ASCII(LOGICAL(PACKET(0)),10,MSGB);
        PRINT(MSG,-TEMP,%40);
        MOVE MSGB := "SOURCE ADDRESS: ";
        PRINT(MSG,-16,%320);
        TEMP := ASCII(LOGICAL(PACKET(1)),10,MSGB);
        PRINT(MSG,-TEMP,%202);
        PRINT(PACKET(2),-(LENGTH - 2),%202);
      END  << THEN >>

      ELSE BEGIN
        MOVE MSGB := "PACKET DAMAGED.  RECEIVE CANCELLED";
        PRINT(MSG,-34,%202);
      END;   << ELSE >>

  END;







<< GETCOMMAND - READ COMMAND FROM KEYBOARD >>

PROCEDURE GETCOMMAND(COMMAND);

  BYTE COMMAND;  << RETURNED COMMAND CHARACTER >>

  BEGIN
    INTEGER TEMP;  << TEMP VARIABLE TO HOLD RETURNED VALUE >>

    MOVE MSGB := "ENTER COMMAND>> ";
    PRINT(MSG,-16,%320);
    TEMP := READ(MSG,-1);    << READ CHAR FROM KEYBOARD >>
    COMMAND := MSGB(0);      << RETURN CHAR >>
  END;  << GETCOMMAND >>




<<............ MAIN PROGRAM   ....................>>



<< MAIN PROGRAM TO READ AND PROCESS COMMANDS >>


  PRINTBANNER;    << PRINT HEADING, VERSION NUMBER >>
  INITIALIZE;     << OPEN CONTROLLER'S PORT, ETC.  >>
  HELP;           << PRINT LIST OF COMMANDS        >>

  << EXECUTE COMMANDS UNTIL QUIT >>

  GETCOMMAND(COMMAND);    << READ FIRST COMMAND >>

  WHILE COMMAND <> "Q" DO
    BEGIN
      IF COMMAND = "S"        << SEND A PACKET >>
        THEN SEND
      ELSE IF COMMAND = "R"   << RECEIVE A PACKET >>
        THEN RECEIVE
      ELSE IF COMMAND = "E"   << ENQUIRE >>
        THEN ENQUIRE
      ELSE IF COMMAND = "C"   << CANCEL >>
        THEN CANCEL
      ELSE IF COMMAND = "H"   << PRINT THIS LIST >>
        THEN HELP
      ELSE ERROR;             << PRINT ERROR MESSAGE >>

      GETCOMMAND(COMMAND);    << GET NEXT COMMAND >>
    END;  << WHILE >>

END.  << MAIN >>
```

Appendix 3

The file program for the VAX

```
/* program to send files over the ethernet.

   Called as:  sendhp srcfile [destfile]  to send a file to the hp3000
        or     gethp  srcfile [destfile]  to read a file from the hp3000

   The command gethp must be linked to this same program.
   That is, compile the program as
       cc ethernet.c -o sendhp'
   which puts the object code in the file sendhp.  This must then be
   linked to gethp using the command
       'ln sendhp gethp'
   which causes both of these names to point to the same file.

   The destfile is optional and defaults to be the same as the srcfile.
   If file name on the hp is not fully specified, it will
   store the file in the public area or try to retrieve the file
   from the public area.  This public area is ETHERNET.SYS.

   Returns an error code via call to exit( ) when program terminates.
   Return of zero means all went ok.
                                                               */




/* include program libraries  */

#include <stdio.h>               /* standard i/o library */
#include <sgtty.h>               /* library for tty set-ups  */
#include <signal.h>              /* library for signals (time-out) */


/* define constants  */

#define TTY  "/dev/tty02" /* port to connect controller to */
#define PACKETSIZE 126    /* maximum size of packets */
#define DATASIZE PACKETSIZE - 6  /* size of data in packet */
#define MAXCOUNT 6          /* maximum number of tries before aborting
                               transmission attempts                */
#define OK  1               /* flag indicates transmission was OK   */
#define NOTOK 0             /* flag indicates transmission was not OK */
#define TIMEOUT 5           /* time-out (in seconds) for read        */
#define T  1                /* true flag                             */
#define F  0                /* false flag                            */
#define VAX 0               /* flags to indicate which machine is   */
#define HP  1               /* the source of the file transfer      */
```

```
/* define packet types */

#define ACKFILE         1       /* file packet received ok        */
#define NAKFILE         2       /* packet not received ok, or     */
                                /* some other error               */
#define DATAFILE        3       /* packet contains file data      */
#define ENDFILE         5       /* end of file                    */
#define ENDREPLY        6       /* ack for ENDFILE                */
#define SENDFILE        7       /* prepares receiver to send file */
                                /* named in DATA part of packet   */
#define RECFILE         8       /* prepares receiver to read and  */
                                /* save file named in DATA part   */



/* define command bytes for controller    */

#define ACK       0x06    /* acknowldgement - all ok */
#define NAK       0x15    /* negative acknowledge - error has occured */
#define ENQ       0x05    /* enquiry command  */
#define SEND      0x11    /* send the following packet */
#define SOH       0x01    /* start of header - device sending SOH is
                             requesting the packet·length             */
#define REC       0x12    /* receive packet from Ethernet  */
#define CAN       0x18    /* cancel last receive request  */



/*  define error codes returned by main program                          */
/*  these indicate status of program when it terminates, returned        */
/*  via an exit(.) call.                                                 */

#define NOERROR      0   /* no error - all ok  */
#define BADARGS      1   /* incorrect number of args. in command line  */
#define BADOPENVAX   2   /* can't open srcfile on VAX      */
#define BADREMOTE    3   /* remote machine not responding  */
#define CONTDOWN     4   /* controller is not responding   */
#define NOCONTOPEN   5   /* can't open the controller's port */
#define NOCONFIG     6   /* can't configure controller's port */
#define XMITERROR    7   /* error in xmit of file.  packets not ack'ed
                            by remote host.                           */
#define BADOPENHP    8   /* can't open srcfile on HP       */



/* declare golbal variables */
```

```
char destaddress;       /* address of destination controller */
char srcaddress;        /* address of source (this) controller */
int controller;         /* file descriptor of controller port */
int timedout;           /* flag to indicate if read has timed-out */


/*********************************************************************/



/*  main program - gets arguments from command line and calls appropriate
                  routine to send/receive the file                    */

main(argc,argv)
   int  argc;           /* command line argument count    */
   char *argv[];        /* argument values - ie. program
                           name and filenames */

{
        char srcfile[30];       /* name of source file */
        char destfile[30];      /* name of destination file */
        int  srcmachine;        /* flag to indicate which machine is to
                                   be the source of the file.  Determined
                                   by the name the program was called
                                   with.                              */
        int  errorcode;         /* error number to be returned   */


        /* get names of files and the source machine    */

        if ( (argc < 2) || (argc > 3) )   /* check number of args */
                {
                  fprintf(stderr,"Usage is:  %s srcfile [destfile]\n",
                                argv[0]);
                  errorcode = BADARGS;
                }
        else
        {        /* find source machine  */
                if (strcmp(argv[0],"sendhp") == 0)    /* matches */
                        srcmachine = VAX;
                else  srcmachine = HP;

                /* get name of source file */
                strcpy(srcfile,argv[1]);

                /* get name of destfile, defaults to srcfile  */
                if (argc == 3)   /* there is a destfile given */
                        strcpy(destfile,argv[2]);
                else  strcpy(destfile,srcfile);
                /* init network addresses, open controller's port  */

                if ((errorcode = initialize()) == 0)  /* no error */
                        {
                        /* send or receive the file, as appropriate  */

                        if (srcmachine == VAX)
                                errorcode
                                    = sendfromvax(srcfile,destfile);
                        else  errorcode = getfromhp(srcfile,destfile);
                        }
        }

        exit(errorcode);

}   /* end of main program  */
```

```
/* sendfromvax - procedure to send a file from the vax to the hp using
   the sendafile routine after initiating communication with the hp
   over the net.  Checks for errors if the file does not exist, etc.
   Returns 0 if sent ok, else returns an errorcode.              */

sendfromvax(srcfile,destfile)

        char srcfile[];         /* name of file to be sent  */
        char destfile[];        /* name to call file at destination */
{
        int  filenum;           /* file number returned by open  */
        char packet[40];        /* packet used to initiate comm. */
        int  packlen;           /* length of finished packet  */
        int  errorcode;         /* returned error code  */

        /* open the file  */

        if ( (filenum = open(srcfile,0)) == -1 )  /* not opened ok */
                {
                fprintf(stderr,"ERROR - can't open %s on VAX\n",srcfile);
                errorcode = BADOPENVAX;
                }
        else
        {
                /* send RECFILE to initiate communication with the hp.
                   Includes destfile as data portion of packet.       */
                strcpy(&packet[6],destfile);
                assem(RECFILE,0,strlen(destfile),packet,&packlen);

                if (senderrorfree(packet,packlen) == NOTOK)
                        {
                        fprintf(stderr,
                             "Remote machine does not respond\n");
                        errorcode = BADREMOTE;
                        }
                else       /* hp is now ready for this file  */
                {
                        errorcode = sendafile(filenum);
                        close(filenum);
                }
        }
        return(errorcode);

}  /* end of sendfromvax  */





/* getfromhp - procedure to receive a file from the hp using
   the recvafile routine after initiating communication with
   the hp over the net.
   If a file already exists with the same name, this routine
   will append '.new' to the filename to avoid overwriting
   the already existing file.
   Returns 0 if the file is received OK,
   else returns an error code                              */

getfromhp(srcfile,destfile)

        char  srcfile[];    /* name of file to be recv'ed from hp */
        char  destfile[];   /* name to call file at this end      */
{
        int  filenum;       /* file number returned by creat       */
        char packet[40];    /* packet used to initiate comm        */
        int  packlen;       /* length of finished packet           */
        int  errorcode;     /* returned error code                 */
        int  packtype;      /* type of packet recv'ed from
                               senderrorfree                       */

        /* send a SENDFILE packet to initiate communication with the hp.
           Includes srcfile name as data portion of the packet.     */

        strcpy(&packet[6],srcfile);
        assem(SENDFILE,0,strlen(srcfile),packet,&packlen);
        packtype = senderrorfree(packet,packlen);

        if (packtype == NOTOK)                  /* not sent */
                {
                fprintf(stderr,"Remote machine does not respond\n");
                errorcode = BADREMOTE;
                }
        else            /* check if hp is ready to send  */
            if (packtype == NAKFILE)
                {
                fprintf(stderr,
                     "Error:  can't open %s on remote machine\n",
                                                            srcfile);
                errorcode = BADOPENHP;
                }
            else        /* hp is now going to start sending the file */
                {
                /* check if a file already exits with same name */
                if (open(destfile,0) != -1)      /* if it exists */
                        /* rename file to be xxxx.new */
                        strcat(destfile,".new");

                /* now create file and receive it from the net */
                filenum = creat(destfile,0755);
                errorcode = recvafile(filenum);
                close(filenum);
                }

        return(errorcode);

}        /* end of getfromhp */




/* initialize - open controller's ports and init. network addresses  */
/* returns 0 if initialized ok, else returns an error code           */

initialize()
{
        int  errorcode;         /* error code to be returned */
        int  temp;              /* temp variable for address */

        /* init controller's port */

        if ((errorcode = initcontroller()) == 0)   /* opened ok */
                /* init network addresses */
                if ((temp = findaddress()) == -1)  /* controller dead */
                        {
                        fprintf(stderr,"Controller on VAX is dead\n");
                        errorcode = CONTDOWN;
                        }
                else  /* init addresses. assumes only 2 stations */
                        if (temp == 0x1f)
                                {
                                srcaddress = 0x1f;
                                destaddress = 0x2f;
                                }
                        else
                        {
                                srcaddress = 0x2f;
                                destaddress = 0x1f;
                        }

        return(errorcode);
}




/* sendafile - sends a file to the remote machine using a stop-and-wait
   protocol with positive acknowledgement and retransmission.
   Enter this routine with the file descriptor (ie. the file must be
   already opened) and the remote machine ready and waiting to
   receive the file.                                                */

sendafile(filenum)

        int  filenum;           /* file number of opened file to send */

{
        int  errorcode;         /* returns an errorcode     */
        char packet[PACKETSIZE]; /* holds packets of file   */
        int  packlen;           /* length of assembled packet  */
        int  datalen;           /* length of data portion   */
        int  seqnum = 1;        /* sequence number of packets  */
        int  ackflag = ACKFILE; /* indicates if packets being ack'ed  */

        /* send the file as packets while not end-of-file   */

        while ( ((datalen = read(filenum,&packet[6],DATASIZE)) > 0)
                        && (ackflag == ACKFILE)  )
        {
                /* assemble and send the packets  */
                assem(DATAFILE,seqnum,datalen,packet,&packlen);
                ackflag = senderrorfree(packet,packlen);
                seqnum = incseq(seqnum);        /* increment seq. num  */
        }

        /* send "END" packet, wait for ack  */

        assem(ENDFILE,seqnum,0,packet,&packlen);
        ackflag = senderrorfree(packet,packlen);

        /* send ENDREPLY, do not wait for ack  */

        seqnum = incseq(seqnum);
        assem(ENDREPLY,seqnum,0,packet,&packlen);
        sendpacket(packet,packlen);

        /* return error code */

        if (ackflag == ACKFILE)                 /* all packets ack'ed */
                errorcode = NOERROR;
        else errorcode = XMITERROR;

        return(errorcode);

}        /* end of sendafile */




/* recvafile - receives a file from the remote machine using a
   stop-and-wait protocol.  Acks received packets and saves
   non-duplicated packets.
   Enter this routine with the file descriptor
   (ie. the file must be already created and opened)
   and the remote machine already set-up to begin
   sending data packets.
                                                               */

recvafile(filenum)

        int  filenum;   /* file number of opened file in which to
                           store the incoming file.           */

{
        char packet[PACKETSIZE]; /* received packet */
        int  packlen;           /* length of received packet */
        int  packtype = DATAFILE; /* type of recv'ed pack t  */
        int  datalen;           /* length of data portion    */
        int  seqnum;            /* sequence number of recv'ed packet */
        int  expectnum = 1;     /* expected sequence number  */
        int  doneflag = F;      /* indicates when done       */

        /* receive, ack, and store packets untile ENDFILE received */

        while (packtype == DATAFILE)
        {
                if (recvpacktime(packet,&packlen) == OK)
                {
                  disassem(packet,&packtype,&seqnum,&datalen);
                  if ((packtype == DATAFILE) && (seqnum == expectnum))
                        {
                        write(filenum,&packet[6],datalen);
                        expectnum = incseq(expectnum);
                        }
                  /* ack the packet */
                  assem(ACKFILE,seqnum,0,packet,&packlen);
                  sendpacket(packet,packlen);
                }
        }

        /* perform end-dally sequence of protocol
           Wait for ENDREPLY (to ACK of ENDFILE) or timeout,
           whichever comes first.
           Timeout or ENDREPLY both mean done.  If recv'ed packet
           is ENDFILE, sender didn't get ACK, so retransmit it.      */

        while (doneflag == F)
                if ( recvpacktime(packet,&packlen) == OK )
                {
                        disassem(packet,&packtype,&seqnum,&datalen);
                        if (packtype == ENDREPLY)  /* got ack, so done */
                                doneflag = T;
                        else        /* retransmit ACK */
                        {
                                assem(ACKFILE,seqnum,0,packet,&packlen);
                                sendpacket(packet,packlen);
                        }
                }
                else          /* timed out, so done */
                    doneflag = T;

        return(NOERROR);

}        /* end of recvafile */
```

```
/* incseq - increment the sequence number.  Returns the new sequence
   number.                                                          */

incseq(n)

        int n;                  /* number to be incremented */
{
        n = (n + 1) % 256;              /* modulo 256 */
        return(n);
}







/* senderrorfree - sends a packet and waits for an ACK
   from the receiver.
   If no ACK, times out and retransmits until a packet
   comes or it has tried MAXCOUNT times.
   Returns type of packet received, if one was received,
   or NOTOK if no packet is ever received                */

senderrorfree(packet,packlen)

        char packet[];                /* packet to be sent  */
        int  packlen;                 /* length of the packet */
{
        char rcvpack[50];       /* received ACK packet  */
        int  rlength;           /* length of ACK packet */
        int  count = 0;         /* counter for number of xmit tries */
        int  rtnflag = NOTOK;   /* flag to be returned  */

        /* send the packet */
        sendpacket(packet,packlen);

        /* wait for ACK with time out, re-xmit up to MAXCOUNT times */

        while ( ((rtnflag = recvpacktime(rcvpack,&rlength)) != OK)
                        && (++count < MAXCOUNT) )
                sendpacket(packet,packlen);     /* retransmit packet */

        /* check if a packet was received   */
        if (rtnflag == OK)                 /* packet recv'ed */
                rtnflag = rcvpack[2];         /* packet type    */

        /* return the flag */
        return(rtnflag);
}
```

```
/* assem - assembles a packet into the required form, ie. prepends
   addresses, etc. onto data already stored in "top" portion of
   packet.                                                          */

assem(packtype,seqnum,datalen,packet,packlen)

        int  packtype;          /* type of packet */
        int  seqnum;            /* sequence number  */
        int  datalen;           /* length of data portion  */
        char packet[];          /* returns assembled packet; enter
                                   with data already in packet        */
        int  *packlen;          /* returns length of assembled packet */
{
        /* assemble the packet */

        packet[0] = destaddress;
        packet[1] = srcaddress;
        packet[2] = packtype;
        packet[3] = seqnum;
        packet[4] = datalen / 256;          /* truncates fraction */
        packet[5] = datalen - (256 * packet[4]);

        /* return packet length */

        *packlen = datalen + 6;

}   /* end of assem */




/* disassem - disassembles a packet.
   Returns via pointers the packet type, the sequence number,
   and the length of the data portion.                            */

disassem(packet,packtype,seqnum,datalen)

        char packet[];     /* packet to be disassembled */
        int  *packtype;    /* returns packet type          */
        int  *seqnum;      /* returns sequence number      */
        int  *datalen;     /* returns length of data portion  */
{
        int thigh;         /* temporaries for type conversion */
        int tlow;

        /* disassemble the packet */
        *packtype = packet[2];
        *seqnum = packet[3] & 0377;   /* only want lower 8 bits */
        thigh = packet[4] & 0377;     /* no sign extension */
        tlow  = packet[5] & 0377;     /* no sign extension */
        *datalen = 256 * thigh + tlow;
}






/* sendpacket - send a single packet.  addresses are embedded within
   the packet.  returns OK if good transmission, NOTOK otherwise     */

sendpacket(packet,length)

 char   packet[];       /* packet to be sent */
 int    length;         /* length of packet  */
{
        char reply;     /* controller's replies to host */
        char status;    /* status byte from controller */
        int  xmitflag = NOTOK;  /* hold OK or NOTOK */
        char len_low;   /* low-order byte of length */
        char len_high;  /* high-order byte of length */
        int  i;         /* index in loop */

        /* split length into high- and low-order parts */

        len_high = length / 256;        /* truncates fraction */
        len_low = length - (len_high * 256);

        /* ready controller to xmit */

        sendcontroller(SEND);
        reply = readcontroller();
        status = readcontroller();

        if (reply == SOH)       /* controller ready */
        {
                sendcontroller(len_low);
                sendcontroller(len_high);
                reply = readcontroller();
                status = readcontroller();

                if (reply == ACK)   /* controller ready */
                {
                        for (i=0;  i<length;  ++i)   /* send packet */
                                sendcontroller(packet[i]);
                        reply = readcontroller();
```

```
                                status = readcontroller();

                                if (reply == ACK)    /* it worked */
                                        xmitflag = OK;
                        }
                }
                return(xmitflag);
        }




/* recvpacket - receives a packet from the ethernet with this
   stations's address or the broadcast address.
   returns OK if recv'd OK, NOTOK otherwise.
   values of packet and length returned via pointers
   cancels receive command if not received OK                    */

recvpacket(packet,plength)

 char packet[];          /* contents of packet received  */
 int  *plength;          /* pointer to length.  Returns length of
                            packet to calling routine            */
{
        char reply;     /* controller's replies to host    */
        char status;    /* status byte from controller     */
        char len_low;   /* low-order byte of length        */
        char len_high;  /* high-order byte of length       */
        int  i;         /* index in loop                   */
        int  rcvflag = NOTOK;   /* holds OK or NOTOK       */

        /* tell controller to receive packet  */

        sendcontroller(REC);
        reply = readcontroller();
        status = readcontroller();

        /* check if packet recv'd OK   */

        if (reply == ACK)   /* packet OK  */
        {
                /* read length */
                sendcontroller(SOH);
                len_low = readcontroller();
                len_high = readcontroller();
                sendcontroller(ACK);

                /* read the packet */
                *plength = (256 * len_high) + len_low;
                for (i=0;  i < *plength;  ++i)
```

```
                        packet[i] = readcontroller();
                sendcontroller(ACK);
                rcvflag = OK;
        }
        else cancel();                  /* not recv'd ok */

        return(rcvflag);
}




/* recvpacktime - receives a packet with time-out.  Returns OK
   if packet received ok, or NOTOK if packet damaged or
   time-out occurred.
   Values of packet and plength are returned via pointers.
   It cancels the receive request if no received ok or time-out.
                                                                  */

recvpacktime(packet,plength)

 char packet[];          /* contents of packet received  */
 int  *plength;          /* pointer to length.  Returns length of
                            packet to calling routine           */
{
        int  reply;     /* controller's replies to host    */
        int  status;    /* status byte from controller     */
        char len_low;   /* low-order byte of length        */
        char len_high;  /* high-order byte of length       */
        int  i;         /* index in loop                   */
        int  rcvflag = NOTOK;   /* holds OK or NOTOK           */

        /* tell controller to receive packet  */

        sendcontroller(REC);
        if ((reply = read_timeout()) == -1)   /* time-out occurred */
                reply = NAK;
        else status = readcontroller();         /* read next character */

        /* check if packet recv'd OK   */

        if (reply == ACK)   /* packet OK  */
        {
                /* read length */
                sendcontroller(SOH);
                len_low = readcontroller();
                len_high = readcontroller();
                sendcontroller(ACK);
                /* read the packet */
                *plength = (256 * len_high) + len_low;
                for (i=0;  i < *plength;  ++i)
                        packet[i] = readcontroller();
                sendcontroller(ACK);
                rcvflag = OK;
        }
        else cancel();                  /* not recv'd OK  */

        return(rcvflag);
}
```

```
/* cancel - send cancel command to controller */

cancel()
{
        char reply;     /* controller's reply to CAN */
        char status;    /* also sends status byte    */

        /* send CAN, read reply and status */

        sendcontroller(CAN);
        reply = readcontroller();
        status = readcontroller();
}
```

```
/* find the ethernet station address of the controller */
/* uses the ENQ command.  returns value of address     */
/* or -1 if controller doesn't respond, ie. times-out. */

findaddress()
{
        int  status;    /* status byte re. rned from controller */

        /* send ENQ to controller  */

        sendcontroller(ENQ);
        /* read status, read and return address  */

        if ((status = read_timeout()) != -1)  /* no time out */
                return(readcontroller());     /* address byte */
        else return(-1);        /* time out, so return -1 for error */
}


/* initcontroller - open and init. tty port to talk to controller  */

initcontroller()
{
    /* structure to initialize tty port to controller   */

    static  struct sgttyb  ttyb = {
                        B9600, B9600,
                        0, 0,
                        RAW | ANYP
        };
        int  errorcode = 0;         /* returned error code */


        /* open controller's tty port  */

        if ( (controller = open(TTY,2)) == -1)  /* couldn't open */
                {
                fprintf(stderr,"Couldn't open controller's port\n");
                fprintf(stderr,
                    "It must be busy (or dead). Try again later\n");
                errorcode = NOCONTOPEN;
                }
        else  /* opened OK so configure port for RAW, 9600 baud */
          if (stty(controller,&ttyb) != 0) /* couldn't configure */
                {
                fprintf(stderr,
                    "Couldn't configure controller's port.\n");
                errorcode = NOCONFIG;
                }
          else     /* set for exclusive access */
                ioctl(controller,TIOCEXCL,0);

        return(errorcode);
}
```

```
/* sendcontroller - send a byte to controller  */

sendcontroller(byte)

 char byte;        /* byte to be sent  */
{
        /* send the byte, print error if not sent OK  */

        if (write(controller, &byte, 1) != 1)
                fprintf(stderr,"Can't write controller\n");
}




/* readcontroller - returns byte read from controller  */

readcontroller()
{
        char byte;        /* byte read in */

        /* read byte, print error if not read OK  */

        if (read(controller, &byte, 1) != 1)
                fprintf(stderr,"Can't read controller\n");
        else return(byte);
}





/* read from controller with time-out.  returns -1 if time-out,
   else returns byte read.                                       */

read_timeout()
{
        int  byte;        /* byte read from controller */
        int  onalrm();    /* procedure when time-out occurs  */

        /* set-up signal, alarm for TIMEOUT seconds  */

        timedout = F;
        signal(SIGALRM, onalrm);
        alarm(TIMEOUT);                         /* set alarm    */
        read(controller,&byte,1);
        if (timedout == T)                    /* time-out occurred */
```

```
                byte = -1;
        alarm(0);                                /* turn off alarm  */
        return(byte);
}




/* sets time-out flag to T when time-out occurs    */

onalrm()
{
        timedout = T;
}
```

Appendix 4

The network daemon for the HP 3000

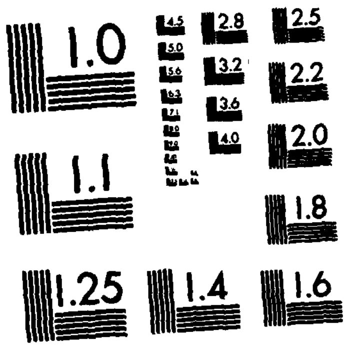
```
$CONTROL USLINIT
BEGIN
COMMENT:
  THIS PROGRAM SENDS AND RECEIVES FILES OVER THE ETHERNET
  USING THE 'ETHERNET FILE TRANSFER PROTOCOL.'
  IT IS INTENDED TO RUN CONTINUOUSLY AS A STREAMED JOB AND
  TAKES ITS COMMANDS TO SEND/RECEIVE FILES BY LISTENING TO
  THE NET.
  ERROR MESSAGES ARE DIRECTED TO THE SYSTEM CONSOLE.

     END OF COMMENT;

<<*************************************************************>>



<< GLOBAL VARIABLE DECLARATIONS >>

  INTEGER     CONTIN;           << FILE # OF CONTROLLER INPUT PORT >>
  INTEGER     CONTOUT;          << FILE # OF CONTROLLER OUTPUT PORT >>
  INTEGER     DESTADDRESS;      << ADDRESS DESTINATION CONTROLLER >>
  INTEGER     SRCADDRESS;       << ADDRESS SOURCE CONTROLLER >>
  BYTE ARRAY  MSGB(0:80);       << I/O BUFFER FOR TERMINAL >>
  LOGICAL ARRAY MSG(*)=MSGB(0);<< PRINT BUFFER -- "WORD" EQUATE >>
  INTEGER     TIMEDOUT;         << FLAG INDICATES THAT A TIME-OUT   >>
                                << OCCURRED WHEN READING CONTROLLER >>
  INTEGER     RENAMENUM;        << USED TO GIVE A UNIQUE NAME TO A >>
                                << FILE IF ONE ALREADY EXISTS WITH  >>
                                << THE SAME NAME                    >>


  << VARIABLES USED BY MAIN PROGRAM ONLY  >>

  BYTE ARRAY COMPACK(0:50);     << COMMAND PACKET FROM REMOTE MACH. >>
  INTEGER COMLENGTH;            << LENGTH OF COMPACK  >>
  INTEGER RFLAG;                << RECEIVE FLAG FOR COMPACK  >>
  INTEGER COMTYPE;              << TYPE OF COMMAND RECV'ED   >>
  INTEGER COMSEQNUM;            << SEQUENCE NUMBER OF COMPACK >>
  INTEGER NAMELEN;              << LENGTH OF NAME IN DATA PORTION   >>
                                << OF COMPACK                       >>

<< DEFINE PACKET TYPES  >>

  EQUATE ACKFILE  = 1,  << PACKET RECEIVED OK BY RECEIVER END >>
         NAKFILE  = 2,  << REQUESTED FILE DOES NOT EXIST    >>
         DATAFILE = 3,  << PACKET CONTAINS FILE DATA             >>
         ENDFILE  = 5,  << END OF FILE   >>
         ENDREPLY = 6,  << ACK FOR 'ENDFILE'   >>
         SENDFILE = 7,  << PREPARES RECEIVER TO SEND FILE NAMED   >>
                        << IN 'DATA' PORTION OF PACKET           >>
```

```
            RECFILE  = 8;   << PREPARES RECEIVER TO READ AND SAVE     >>
                           << FILE NAMED IN 'DATA' PORTION OF PACKET >>


   << DEFINE COMMAND BYTES TO/FROM CONTROLLER >>

     EQUATE ACK  = %(16)06,  << ACKNOWLEDGEMENT -- ALL OK >>
            NAK  = %(16)15,  << NEGATIVE ACK - ERROR HAS OCCURED >>
            ENQ  = %(16)05,  << ENQUIRE - RETURN STATUS & ADDRESS >>
            SND  = %(16)11,  << SEND THE FOLLOWING PACKET >>
            SOH  = %(16)01,  << START OF HEADER - DEVICE SENDING IS >>
                             << REQUESTING THE PACKET LENGTH        >>
            REC  = %(16)12,  << RECEIVE PACKET FROM ETHERNET >>
            CAN  = %(16)18;  << CANCEL LAST RECEIVE REQUEST >>


   << DEFINE CONSTANTS >>

     EQUATE PACKETSIZE = 126, << MAXIMUM SIZE OF PACKETS >>
            DATASIZE = PACKETSIZE - 6,  << SIZE OF DATA IN PACKET >>
            MAXCOUNT   = 6,    << MAXIMUM NUMBER OF TRIES BEFORE  >>
                               << ABORTING TRANSMISSION ATTEMPT   >>
            OK         = 1,    << FLAG FOR "OK"  >>
            NOTOK      = 0,    << FLAG FOR "NOTOK"  >>
            TIMEOUT    = 5,    << 'READ' TIME-OUT INTERVAL >>
            NOTIMEOUT  = 0,    << PARAM FOR NO TIME-OUT ON READ  >>
            TRU        = 1,    << TRUE FLAG  >>
            FALS       = 0;    << FALSE FLAG >>



   << INTRINSIC DECLARATIONS >>

     INTRINSIC READ,PRINT,BINARY,FREAD,FWRITE,FOPEN,FSETMODE,
               FCONTROL,IOWAIT,GETPRIVMODE,GETUSERMODE,ASCII,
               FCLOSE,FRENAME,PRINTOP;



   <<.......  PROCEDURE DECLARATIONS ...........>>



   << RWCNTLR - READ/WRITE FROM CONTROLLER >>
```

AD-A124 448 A LOCAL COMPUTER NETWORK IMPLEMENTATION USING ETHERNET 2/2
(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB
D J LILJA AUG 82 R-946 N00039-80-C-0556

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
 COMMENT: BECAUSE OF THE LACK OF BUFFERING ON INPUT PORTS,
          MUST FIRST SET-UP READ, THEN WRITE, THEN FINISH
          READ.  THAT IS, CONTROLLER RESPONDS TOO QUICKLY
          FOR HP TO CATCH IN NORMAL I/O ;

PROCEDURE RWCNTLR(INBUFFB,INLEN,OUTBUFFB,OUTLEN,TIME);

  VALUE OUTLEN,         << PASS THESE BY VALUE >>
        TIME,
        INLEN;

  BYTE ARRAY INBUFFB;   << INPUT BUFFER TO CONTROLLER >>
                        << RETURNS WITH DATA READ FROM CONTROLLER >>
  INTEGER INLEN;        << NUMBER OF BYTES TO READ FROM CONT >>
  BYTE ARRAY OUTBUFFB;  << OUTPUT BUFFER TO CONTROLLER >>
                        << ENTER WITH DATA TO SEND TO CONT >>
  INTEGER OUTLEN;       << NUMBER OF BYTES TO SEND TO CONT >>
  INTEGER TIME;         << TIME-OUT INTERVAL - 0 MEANS NO TIME-OUT >>


  BEGIN
    LOGICAL ARRAY INBUFF(*)=INBUFFB(0); << EQUATE TO ELIMINATE >>
    LOGICAL ARRAY OUTBUFF(*)=OUTBUFFB(0);  << WARNING MESGS >>
    INTEGER DUMMY;  << HOLDS RETURNED VALUE FROM IOWAIT, FREAD >>
    INTEGER PARAM;  << USED IN FCONTROL FOR TIMEOUT >>

    << INIT. TIME-OUT FLAG >>
    TIMEDOUT := FALS;

    << SET TIME-OUT INTERVAL  >>
    PARAM := TIME;
    FCONTROL(CONTIN,4,PARAM);

    << SET-UP READ, NO WAIT FOR FINISH >>
    DUMMY := FREAD(CONTIN,INBUFF,-INLEN);

    << WRITE BUFFER TO CONTROLLER >>
    FWRITE(CONTOUT,OUTBUFF,-OUTLEN,%320);

    << FINISH READ, RETURN BUFFER AND LENGTH >>
    DUMMY := IOWAIT(CONTIN,INBUFF);
    IF < THEN
      BEGIN
        TIMEDOUT := TRU;
      END;

  END;  << RWCNTLR >>
```

```
<< FIND ETHERNET STATION ADDRESS OF THE CONTROLLER USING ENQ   >>
<< COMMAND.  RETURNS ADDRESS IN 'ADDRESS' PARAMETER.          >>

PROCEDURE FINDADDRESS(ADDRESS);

  INTEGER ADDRESS;    << RETURNS ETHERNET STATION ADDRESS >>

  BEGIN
    BYTE ARRAY BUFF(0:1);  << I/O BUFFER FOR CONTROLLER >>

    << LOAD 'ENQ' INTO BUFFER >>
    BUFF(0) := ENQ;

    << SEND TO CONTROLLER, READ STATUS AND ADDRESS >>
    RWCNTLR(BUFF,2,BUFF,1,TIMEOUT);

    << RETURN ADDRESS IF NO TIME-OUT >>
    IF TIMEDOUT = FALS
      THEN ADDRESS := INTEGER(BUFF(1))
      ELSE ADDRESS := 0;

  END;  << FINDADDRESS >>




<< INITIALIZE CONTROLLER'S PORTS, INIT STATION ADDRESSES >>

PROCEDURE INITIALIZE;

  BEGIN
    INTEGER LENGTH;  << TEMP STORAGE OF RETURNED VALUE FROM READ >>
    INTEGER CNTLCODE; << CONTROL CODE FOR FCONTROL >>
    BYTE ARRAY TTYIN(0:8);  << INPUT PORT >>
    BYTE ARRAY TTYOUT(0:8);  << OUTPUT PORT >>
    INTEGER TEMP;     << TEMP VARIABLE FOR ADDRESS INITIALIZATION >>


    << INIT TTYIN AND TTYOUT >>

    MOVE TTYIN := "ENETIN;";
    MOVE TTYOUT := "ENETOUT;";

    << OPEN AND INIT CONTROLLER'S PORTS >>

    GETPRIVMODE;   << FOR NO-WAIT I/O >>
    CONTIN := FOPEN( ,%604,%4324,-36,TTYIN);
```

```
IF <>  THEN BEGIN  << CAN'T OPEN >>
  MOVE MSGB := "ETHERNET - CAN'T OPEN INPUT PORT";
  PRINTOP(MSG,-32,0);
 END;
GETUSERMODE;   << NO MORE NEED FOR PRIV MODE >>

CONTOUT := FOPEN( ,%604,%324,-36,TTYOUT);
IF <> THEN BEGIN   << CAN'T OPEN >>
  MOVE MSGB := "ETHERNET - CAN'T OPEN OUTPUT PORT";
  PRINTOP(MSG,-33,0);
 END;

CNTLCODE := %074022;   << SPEED, TERMTYPE >>
FCONTROL(CONTIN,37,CNTLCODE);  << ALLOCATE TERMTYPE 18 AT >>
FCONTROL(CONTOUT,37,CNTLCODE); << 9600 BAUD              >>

FSETMODE(CONTIN,%4);   << INHIBIT AUTO CR-LF ON INPUT >>

FCONTROL(CONTIN,13,CNTLCODE);  << ECHO OFF >>
FCONTROL(CONTOUT,13,CNTLCODE);

FCONTROL(CONTIN,20,CNTLCODE);  << DISABLE INPUT TIMER >>

FCONTROL(CONTIN,28,CNTLCODE);  << DISABLE BLOCK MODE >>
FCONTROL(CONTOUT,28,CNTLCODE);

FCONTROL(CONTIN,27,CNTLCODE);  << ENABLE BINARY TRANSFERS >>
FCONTROL(CONTOUT,27,CNTLCODE);

CNTLCODE := 0;
FCONTROL(CONTIN,36,CNTLCODE);  << NO PARITY, FULL 8 BITS >>
FCONTROL(CONTOUT,36,CNTLCODE);


<< INIT. SRC AND DEST ADDRESSES.  ASSUMES ONLY 2 STATIONS >>

FINDADDRESS(TEMP);
IF TIMEDOUT = TRU       << CONTROLLER DEAD >>
  THEN BEGIN    << PRINT ERROR MESSAGE >>
    MOVE MSGB := "ETHERNET CONTROLLER IS DEAD.  TRY RESETTING.";
    PRINTOP(MSG,-43,0);
  END  << THEN >>
  ELSE    << INIT. ADDRESSES >>
    IF TEMP = %(16)1F
      THEN BEGIN
        SRCADDRESS := %(16)1F;
        DESTADDRESS := %(16)2F;
      END
      ELSE BEGIN
        SRCADDRESS := %(16)2F;
        DESTADDRESS := %(16)1F;
```

```
                END;

        END;  << INITIALIZE >>




      << SENDPACKET - SEND A SINGLE PACKET.  ADDRESSES ARE EMBEDDED  >>
      << WITHIN THE PACKET.  RETURNS XMITFLAG = OK IF GOOD           >>
      << TRANSMISSION, NOTOK OTHERWISE.  LEAVES PACKET UNCHANGED.    >>

      PROCEDURE SENDPACKET(PACKET,LENGTH,XMITFLAG);

        VALUE LENGTH;           << PASS BY VALUE >>

        BYTE ARRAY PACKET;      << PACKET TO BE SENT WITH EMBEDDED ADDRESS >>
        INTEGER LENGTH;         << LENGTH OF PACKET IN BYTES >>
        INTEGER XMITFLAG;       << RETURNS 'OK' OR 'NOTOK'  >>

        BEGIN
          BYTE ARRAY BUFF(0:1);  << BUFFER FOR SENDING COMMANDS    >>
                                 << TO/FROM CONTROLLER             >>

          << INIT XMITFLAG >>
          XMITFLAG := NOTOK;

          << READY CONTROLLER TO XMIT BY SENDING 'SND'.    >>
          << READ REPLY AND STATUS                         >>

          BUFF(0) := SND;
          RWCNTLR(BUFF,2,BUFF,1,TIMEOUT);    << SEND 1 BYTE, READ 2 BYTES >>

          IF BUFF(0) = SOH  AND  TIMEDOUT = FALS  << REPLY IS SOH >>
            THEN BEGIN    << SEND LENGTH, READ REPLY AND STATUS >>
              BUFF(0) := BYTE(LENGTH.(8:8));  << LOW-ORDER 8 BITS >>
              BUFF(1) := BYTE(LENGTH.(0:8));  << HIGH-ORDER 8 BITS >>
              RWCNTLR(BUFF,2,BUFF,2,TIMEOUT);  << READ/WRITE 2 BYTES >>

              IF BUFF(0) = ACK  << CONTROLLER READY FOR PACKET >>
                THEN BEGIN   << SEND PACKET, READ REPLY, STATUS >>
                  RWCNTLR(BUFF,2,PACKET,LENGTH,TIMEOUT);
                  IF BUFF(0) = ACK  << XMIT OK >>
                    THEN XMITFLAG := OK;
                END;
            END;

        END;  << SENDPACKET >>
```

```
<< CANCEL - SEND CANCEL COMMAND TO CONTROLLER >>

PROCEDURE CANCEL;

  BEGIN
    BYTE ARRAY BUFF(0:1);  << BUFFER FOR COMMANDS TO/FROM CONT >>

    << SEND 'CAN', READ REPLY AND STATUS >>

    BUFF(0) := CAN;
    RWCNTLR(BUFF,2,BUFF,1,TIMEOUT);   << SEND 1 BYTE, READ 2 >>
  END;  << CANCEL >>




<< RECVPACKET - RECEIVES A PACKET FROM THE ETHERNET WITH THIS   >>
<< STATION'S ADDRESS OR THE BROADCAST ADDRESS.  RETURNS WITH    >>
<< RCVFLAG = OK IF RECEIVED OK, NOTOK OTHERWISE.  CANCELS       >>
<< RECEIVE COMMAND (WITH 'CAN') IF NOT RECEIVED OK.             >>
<< ALLOWS SETTING OF TIME-OUT INTERVAL FOR READ OF CONTROLLER   >>


PROCEDURE RECVPACKET(PACKET,LENGTH,RCVFLAG,TIME);

  VALUE TIME;           << PASS BY VALUE  >>

  BYTE ARRAY PACKET;    << RETURNS WITH PACKET RECEIVED >>
  INTEGER LENGTH;       << RETURNS LENGTH OF RECEIVED PACKET >>
  INTEGER RCVFLAG;      << RETURNS 'OK' OR 'NOTOK'  >>
  INTEGER TIME;         << INDICATES TIME-OUT INTERVAL TO WAIT FOR  >>
                        << PACKET.  0 MEANS WAIT FOREVER.           >>

  BEGIN
    BYTE ARRAY BUFF(0:1);  << BUFFER FOR SENDING COMMANDS    >>
                           << TO/FROM CONTROLLER             >>

    << INIT. FLAG >>
    RCVFLAG := NOTOK;

    << TELL CONTROLLER TO RECEIVE PACKET.  WAITS UNTIL PACKET >>
    << IS RECEIVED.                                           >>

    BUFF(0) := REC;
    RWCNTLR(BUFF,2,BUFF,1,TIME);    << SEND 1 BYTE, READ 2 BYTES  >>

    << CHECK IF PACKET RECEIVED OK >>

    IF BUFF(0) = ACK  AND  TIMEDOUT = FALS      << PACKET OK  >>
      THEN BEGIN
        << SEND 'SOH', READ LENGTH >>
        BUFF(0) := SOH;
        RWCNTLR(BUFF,2,BUFF,1,TIME);   << SEND 1 BYTE, READ 2 BYTES >>
        LENGTH := 256 * BUFF(1) + BUFF(0);

        << SEND 'ACK', READ PACKET >>
        BUFF(0) := ACK;
        RWCNTLR(PACKET,LENGTH,BUFF,1,TIME);

        << SEND CONTROLLER FINAL 'ACK'  >>
        BUFF(0) := ACK;
        RWCNTLR(BUFF,0,BUFF,1,TIME); << SEND 1 BYTE, READ NO BYTES >>

        << RETURN FLAG 'OK' >>
        RCVFLAG := OK;
      END   << THEN >>

      ELSE CANCEL;   << NOT RECEIVED OK, SO CANCEL 'REC' >>

  END;   << RECVPACKET >>




<< ASSEMBLE - ASSEMBLE A PACKET INTO THE REQUIRED FORM  >>
<< ENTER ROUTINE WITH DATA ALREADY STORED IN HIGH PART OF PACKET >>

PROCEDURE ASSEM(PACKTYPE,SEQNUM,DATALEN,PACKET,PACKLEN);

  VALUE PACKTYPE,           << PASS THESE BY VALUE >>
        SEQNUM,
        DATALEN;

  INTEGER PACKTYPE;       << TYPE OF PACKET >>
  INTEGER SEQNUM;         << SEQUENCE NUMBER  >>
  INTEGER DATALEN;        << LENGTH OF DATA PORTION  >>
  BYTE ARRAY PACKET;      << RETURNED ASSEMBLED PACKET >>
  INTEGER PACKLEN;        << RETURNED LENGTH OF ASSEMBLED PACKET >>

  BEGIN
    << ASSEMBLE THE PACKET >>

    PACKET(0) := BYTE(DESTADDRESS);
    PACKET(1) := BYTE(SRCADDRESS);
    PACKET(2) := BYTE(PACKTYPE);
    PACKET(3) := BYTE(SEQNUM);
    PACKET(4) := BYTE(DATALEN.(0:8));    << HIGH-ORDER 8 BITS >>
    PACKET(5) := BYTE(DATALEN.(8:8));    << LOW-ORDER 8 BITS  >>

    << RETURN PACKET LENGTH >>
    PACKLEN := DATALEN + 6;    << ADD HEADER BYTES >>

  END;  << ASSEM >>




<< DISASSEM - DISASSEMBLE A RECEIVED PACKET INTO COMPONENT PARTS >>

PROCEDURE DISASSEM(PACKET,PACKTYPE,SEQNUM,DATALEN);

  BYTE ARRAY PACKET;     << PACKET TO BE DISASSEMBLED >>
  INTEGER DATALEN;       << RETURNED LENGTH OF FDATA >>
  INTEGER PACKTYPE;      << RETURNED TYPE OF PACKET >>
  INTEGER SEQNUM;        << RETURNED SEQUENCE NUMBER >>

  BEGIN
    << DISASSEMABLE THE PACKET >>

    PACKTYPE := INTEGER(PACKET(2));
    SEQNUM := INTEGER(PACKET(3));
    DATALEN := INTEGER(256 * PACKET(4) + PACKET(5));

  END;  << DISASSEM >>




<< INCSEQ - INCREMENT A SEQUENCE NUMBER  >>

PROCEDURE INCSEQ(N);

  INTEGER N;

  BEGIN
    N := (N + 1) MOD 256;
  END;  << INCSEQ >>
```

```
<< RENAME - RENAMES A FILE BY ADDING OR REPLACING THE LAST
   CHARACTER IN THE FILENAME WITH THE ASCII EQUIVALENT OF
   THE GLOBAL VARIABLE RENAMENUM.
   A FILENAME IS OF THE FORM 'FILENAME.GROUP.ACCOUNT;'
   WHERE THE GROUP AND ACCOUNT ARE OPTIONAL.              >>

PROCEDURE RENAME(FILENUM,OLDNAME);

  INTEGER FILENUM;     << FILE DESCRIPTOR >>
  BYTE ARRAY OLDNAME;  << OLD NAME OF THE FILE >>

  BEGIN
    BYTE ARRAY NEWNAME(0:30);  << NEW NAME OF FILE >>
    INTEGER NEWI;              << INDEX FOR NEWNAME >>
    INTEGER OLDI;              << INDEX FOR OLDNAME >>
    INTEGER TEMP;              << TEMP. FOR USE IN ASCII CALL >>

    << MOVE OLD NAME INTO NEW NAME UNTIL FIND ; OR .    >>

    OLDI := 0;
    NEWI := 0;
    DO BEGIN
        NEWNAME(NEWI) := OLDNAME(OLDI);
        NEWI := NEWI + 1;
        OLDI := OLDI + 1;
      END
    UNTIL  OLDNAME(OLDI) = ";"  OR  OLDNAME(OLDI) = "." ;

    << IF 8 CHARS. IN FILENAME, REPLACE LAST CHAR. WITH RENAMENUM,
       ELSE APPEND RENAMENUM TO FILENAME.                     >>

    IF OLDI = 8
      THEN  << REPLACE LAST CHAR >>
        TEMP := ASCII(RENAMENUM,10,NEWNAME(NEWI - 1))
      ELSE BEGIN     << APPEND NEW CHAR >>
        TEMP := ASCII(RENAMENUM,10,NEWNAME(NEWI));
        NEWI := NEWI + 1;    << POINT TO NEXT CHAR. >>
      END;  << ELSE >>

    << MOVE REMAINING CHARS.  >>

    DO BEGIN
      NEWNAME(NEWI) := OLDNAME(OLDI);
      NEWI := NEWI + 1;
      OLDI := OLDI + 1;
     END
    UNTIL  OLDNAME(OLDI) = ";" ;
    << APPEND A ;  >>
    NEWNAME(NEWI) := ";" ;

    << RENAME THE FILE >>
    FRENAME(FILENUM,NEWNAME);

    << INCREMENT RENAMENUM >>
    RENAMENUM := (RENAMENUM + 1) MOD 10;

  END;  << RENAME >>
```

```
<< SENDERRORFREE - SENDS A PACKET AND WAITS FOR AN ACK FROM THE
   RECEIVER.  IF NO ACK, TIMES OUT AND RETRANSMITS UNTIL A
   PACKET COMES OR IT HAS TRIED MAXCOUNT TIMES.
   RETURNS THE TYPE OF PACKET RECEIVED, IF ONE WAS RECEIVED, OR
   'NOTOK' IF NO PACKET RECEIVED AT ALL.                          >>

PROCEDURE SENDERRORFREE(PACKET,LENGTH,RCVTYPE);

  BYTE ARRAY PACKET;  << PACKET TO BE SENT  >>
  INTEGER LENGTH;     << LENGTH OF PACKET IN BYTES >>
  INTEGER RCVTYPE;    << RETURNS NOTOK, OR TYPE OF PACKET RECV'ED >>

  BEGIN
    BYTE ARRAY RCVPACK(0:50);  << RECEIVED ACK PACKET >>
    INTEGER RLENGTH;           << LENGTH OF RECV'ED PACKET >>
    INTEGER RCVFLAG;           << FLAG RETURNED FROM 'RECVPACKET' >>
    INTEGER XMITFLAG;          << FLAG RETURNED FROM 'SENDPACKET' >>
    INTEGER COUNT;             << NUMBER OF XMIT ATTEMPTS         >>

    << INIT. VARIABLES >>
    COUNT := 0;
    RCVFLAG := NOTOK;

    << SEND THE PACKET, WAIT FOR ACK, RETRANSMIT UP TO MAXCOUNT >>
    WHILE RCVFLAG = NOTOK  AND  COUNT < MAXCOUNT  DO
      BEGIN
        SENDPACKET(PACKET,LENGTH,XMITFLAG);
        RECVPACKET(RCVPACK,RLENGTH,RCVFLAG,TIMEOUT);
        COUNT := COUNT + 1;
      END;
    << CHECK IF A PACKET WAS RECEIVED >>
    IF RCVFLAG = OK
      THEN RCVTYPE := INTEGER(RCVPACK(2))    << RETURN TYPE >>
      ELSE RCVTYPE := NOTOK;

END;  << SENDERRORFREE >>





<< SENDAFILE - SENDS A FILE TO THE REMOTE MACHINE USING A STOP-
   AND-WAIT PROTOCOL WITH POSITIVE ACKNOWLEDGEMENT AND RETRANS-
   MISSION.
   ENTER THIS ROUTINE WITH THE FILE DESCRIPTOR (IE. THE FILE MUST
   ALREADY BE OPENED) AND THE REMOTE MACHINE READY AND WAITING TO
   RECEIVE THE FILE.                                            >>

PROCEDURE SENDAFILE(FILENUM);

  INTEGER FILENUM;    << FILE DESCRIPTOR OF OPENED FILE TO SEND >>

  BEGIN
    BYTE ARRAY PACKET(0:PACKETSIZE);    << PACKETS TO SEND >>
    INTEGER PACKLEN;      << LENGTH OF PACKETS >>
    INTEGER DATALEN;      << LENGTH OF DATA PORTION >>
    INTEGER SEQNUM;       << SEQUENCE NUMBER OF PACKETS >>
    INTEGER ACKFLAG;      << INDICATES IF PACKETS ARE BEING ACKED >>
    INTEGER EOF;          << END OF FILE FLAG >>
    INTEGER TEMP;         << TEMPORARY >>

    << INIT VARIABLES >>
    SEQNUM := 1;
    ACKFLAG := ACKFILE;
    EOF := FALS;

    << READ FIRST DATA RECORD >>
    DATALEN := FREAD(FILENUM,PACKET(6),-DATASIZE);
    IF <>        << EOF OR ERROR >>
      THEN EOF := TRU;

    << XMIT PACKETS UNTIL END-OF-FILE >>
    WHILE EOF = FALS  AND  ACKFLAG = ACKFILE  DO
      BEGIN
        << ASSEMBLE AND SEND THE PACKET >>
        ASSEM(DATAFILE,SEQNUM,DATALEN,PACKET,PACKLEN);
        SENDERRORFREE(PACKET,PACKLEN,ACKFLAG);
      << GET NEXT DATA RECORD >>
      DATALEN := FREAD(FILENUM,PACKET(6),-DATASIZE);
      IF <>       << EOF OR ERROR >>
        THEN EOF := TRU;

      INCSEQ(SEQNUM);
    END;   << WHILE >>

    << SEND 'ENDFILE', WAIT FOR ACK >>
    ASSEM(ENDFILE,SEQNUM,0,PACKET,PACKLEN);
    SENDERRORFREE(PACKET,PACKLEN,ACKFLAG);

    << SEND 'ENDREPLY', NO WAIT FOR ACK >>
    INCSEQ(SEQNUM);
    ASSEM(ENDREPLY,SEQNUM,0,PACKET,PACKLEN);
    SENDPACKET(PACKET,PACKLEN,TEMP);


END;    << SENDAFILE >>






<< HP'IS'SOURCE - SENDS A FILE FROM THE HP TO THE REMOTE MACHINE
   USING THE 'SENDAFILE' ROUTINE AFTER CHECKING TO MAKE SURE THE
   FILE REALLY EXISTS.  IF IT DOESN'T, IT SENDS A 'NAKFILE' TO
   THE REMOTE MACHINE.
   THE FILENAME PASSED AS AN ARGUMENT SHOULD END IN A ';' TO MAKE
   THE FILE SYSTEM HAPPY.                                         >>

PROCEDURE HP'IS'SOURCE(FILENAME);

  BYTE ARRAY FILENAME;       << NAME OF FILE TO SEND >>

  BEGIN
    INTEGER FILENUM;          << FILE DESCRIPTOR RETURNED BY FOPEN >>
    BYTE ARRAY PACKET(0:20); << PACKET USED TO ACK OR NAK         >>
    INTEGER PACKLEN;          << LENGTH OF PACKET                  >>
    INTEGER XMITFLAG;         << FLAG RETURNED BY 'SENDPACKET'     >>

    << OPEN THE FILE, IF IT EXISTS >>
    FILENUM := FOPEN(FILENAME,%7,0,-DATASIZE);

    IF <     << ERROR, FILE NOT OPENED (DOESN'T EXIST) >>
      THEN BEGIN   << SEND 'NAKFILE' PACKET >>
        ASSEM(NAKFILE,0,0,PACKET,PACKLEN);
        SENDPACKET(PACKET,PACKLEN,XMITFLAG);
        END    << THEN >>
        ELSE BEGIN   << SEND 'ACKFILE' AND SEND THE FILE >>
          ASSEM(ACKFILE,0,0,PACKET,PACKLEN);
          SENDPACKET(PACKET,PACKLEN,XMITFLAG);
          SENDAFILE(FILENUM);
          FCLOSE(FILENUM,0,0);
        END;   << ELSE >>

    END;      << HP'IS'SOURCE >>





<< RECVAFILE - RECEIVES A FILE FROM THE REMOTE MACHINE USING A
   STOP-AND-WAIT PROTOCOL.  ACKS ALL RECEIVED PACKETS AND SAVES
   NON-DUPLICATED PACKETS.
   ENTER THIS ROUTINE WITH THE FILE DESCRIPTOR (IE. THE FILE MUST
   ALREADY BE OPENED) AND THE REMOTE MACHINE ALL READY TO BEGIN
   SENDING DATA PACKETS.                                          >>

PROCEDURE RECVAFILE(FILENUM);

  INTEGER FILENUM;          << FILE NUMBER OF OPENED FILE IN WHICH
                               TO STORE THE INCOMING PACKETS      >>

  BEGIN
    BYTE ARRAY PACKET(0:PACKETSIZE); << RECEIVED PACKET  >>
    INTEGER PACKLEN;        << LENGTH OF RECEIVED PACKET >>
    INTEGER RCVFLAG;        << STATUS OF RECEIVED PACKET >>
    INTEGER PACKTYPE;       << TYPE OF RECEIVED PACKET   >>
    INTEGER DATALEN;        << LENGTH OF DATA PORTION    >>
    INTEGER SEQNUM;         << SEQUENCE NUMBER OF PACKET >>
    INTEGER EXPECTNUM;      << EXPECTED SEQUENCE NUMBER  >>
    INTEGER DONEFLAG;       << INDICATES WHEN FINISHED   >>
    INTEGER XMITFLAG;       << FLAG RETURNED BY SENDPACKET >>

    << INIT. VARIABLES >>
    PACKTYPE := DATAFILE;
    EXPECTNUM := 1;

    << RECEIVE, ACK, AND STORE PACKETS UNTIL 'ENDFILE' RECEIVED >>
    WHILE PACKTYPE = DATAFILE  DO
      BEGIN
        RECVPACKET(PACKET,PACKLEN,RCVFLAG,TIMEOUT);
        IF RCVFLAG = OK     << PACKET RECEIVED OK >>
          THEN BEGIN
            DISASSEM(PACKET,PACKTYPE,SEQNUM,DATALEN);
          << IF DATA PACKET AND NOT A DUPLICATE >>
          IF PACKTYPE = DATAFILE  AND  SEQNUM = EXPECTNUM
            THEN BEGIN   << SAVE PACKET AND INCREMENT EXPECTNUM >>
              FWRITE(FILENUM,PACKET(6),-DATALEN,%320);
              INCSEQ(EXPECTNUM);
            END;  << THEN >>

          << ACK THE PACKET >>
          ASSEM(ACKFILE,SEQNUM,0,PACKET,PACKLEN);
          SENDPACKET(PACKET,PACKLEN,XMITFLAG);
        END;  << THEN >>
    END;  << WHILE >>

  << PERFORM END-DALLY SEQUENCE OF PROTOCOL:
     WAIT FOR 'ENDREPLY' (TO 'ACK' OF 'ENDFILE') OR TIMEOUT,
     WHICHEVER COMES FIRST.  TIMEOUT OR 'ENDREPLY' BOTH MEAN
     DONE.  IF RECV'ED PACKET IS 'ENDFILE', SENDER DIDN'T GET
     LAST ACK, SO RETRANSMIT IT.                             >>

  DONEFLAG := FALS;

  WHILE DONEFLAG = FALS  DO   << WAIT FOR TIMEOUT OR ENDREPLY >>
    BEGIN
      RECVPACKET(PACKET,PACKLEN,RCVFLAG,TIMEOUT);
      IF RCVFLAG = OK    << SOME PACKET RECV'ED OK >>
        THEN BEGIN    << SEE WHAT KIND OF PACKET IT IS >>
          DISASSEM(PACKET,PACKTYPE,SEQNUM,DATALEN);
          IF PACKTYPE = ENDREPLY    << GOT ACK, SO DONE >>
            THEN DONEFLAG := TRU
            ELSE BEGIN   << RETRANSMIT THE ACK >>
              ASSEM(ACKFILE,SEQNUM,0,PACKET,PACKLEN);
              SENDPACKET(PACKET,PACKLEN,XMITFLAG);
            END;  << ELSE >>
        END  << THEN >>
        ELSE  << TIMED OUT, SO DONE >>
            DONEFLAG := TRU;
    END;  << WHILE >>

END;  << RECVAFILE >>




<< HP'IS'DEST - RECEIVES A FILE FROM THE REMOTE MACHINE USING THE
   'RECVAFILE' ROUTINE AND SAVE IT ON DISK.  IF A FILE WITH THE
   SAME NAME ALREADY EXISTS, THIS ROUTINE WILL RENAME THE FILE,
     THEN SAVE IT.
     THE FILENAME GIVEN AS AN ARGUMENT SHOULD BE TERMINATED WITH
     A ';' TO MAKE THE FILE SYSTEM HAPPY.                       >>

  PROCEDURE HP'IS'DEST(FILENAME);

    BYTE ARRAY FILENAME;        << NAME TO CALL RECEIVED FILE >>

    BEGIN
      INTEGER FILENUM;          << FILE DESCRIPTOR RETURNED BY FOPEN >>
      BYTE ARRAY PACKET(0:20);  << PACKET USED TO ACK                >>
      INTEGER PACKLEN;          << LENGTH OF PACKET                  >>
      INTEGER XMITFLAG;         << FLAG RETURNED BY 'SENDPACKET'     >>

      << OPEN THE FILE AS A 'NEW' FILE >>
      FILENUM := FOPEN(FILENAME,%4,%101,-DATASIZE);
      IF <>      << COULDN'T OPEN THE FILE >>
        THEN BEGIN
          MOVE MSGB := "ETHERNET - CAN'T OPEN A NEW FILE";
          PRINTOP(MSG,-32,0);
        END << THEN >>

        ELSE BEGIN
          << ACK THE 'RECFILE' PACKET >>
          ASSEM(ACKFILE,0,0,PACKET,PACKLEN);
          SENDPACKET(PACKET,PACKLEN,XMITFLAG);

          << RECEIVE THE FILE >>
          RECVAFILE(FILENUM);

          << CLOSE THE FILE AS A PERMANENT FILE >>
          FCLOSE(FILENUM,1,0);
          IF <>      << FILE NOT CLOSED >>
            THEN BEGIN    << RENAME THE FILE >>
              RENAME(FILENUM,FILENAME);
              FCLOSE(FILENUM,1,0);
            END;  << THEN >>
        END;  << ELSE >>

    END;  << HP'IS'DEST >>
```

```
<<................ MAIN PROGRAM ..........................>>


<< MAIN PROGRAM (NETWORK DEMON) - LISTENS FOR A COMMAND PACKET ON THE
   ETHERNET.  IF TYPE OF PACKET IS 'SENDFILE', IT CONTAINS THE
   NAME OF A FILE FOR THE HP TO SEND TO THE REMOTE MACHINE.  IF THE
   TYPE OF PACKET IS 'RECFILE', IT MEANS THE HP SHOULD PREPARE TO
   RECEIVE A FILE FROM THE REMOTE MACHINE AND STORE IT WITH THE NAME IN
   THE DATA PORTION OF THE PACKET.                                 >>

  << INIT. CONTROLLER'S I/O PORTS, INIT. STATION ADDRESSES, ETC.  >>
  MOVE MSGB := "ETHERNET IS INITIALIZING";
  PRINTOP(MSG,-24,0);
  INITIALIZE;
  RENAMENUM := 0;      << INITIALIZE IT >>

  << LOOP FOREVER >>
  WHILE TRU DO
   BEGIN

    << WAIT FOREVER UNTIL A PACKET ARRIVES ON NET >>
    RECVPACKET(COMPACK,COMLENGTH,RFLAG,NOTIMEOUT);

    IF RFLAG = OK         << PACKET RECV'ED OK >>
      THEN BEGIN
        << DISASSEMBLE THE PACKET >>
        DISASSEM(COMPACK,COMTYPE,COMSEQNUM,NAMELEN);

        << TERMINATE FILE NAME WITH A ';' FOR THE FILE SYSTEM >>
        COMPACK(COMLENGTH) := ";";

        << SEND OR RECEIVE THE FILE AS APPROPRIATE, SUPPLYING THE  >>
        << FILE NAME AS AN ARGUMENT                              >>
        IF COMTYPE = RECFILE
          THEN HP'IS'DEST(COMPACK(6))
          ELSE
            IF COMTYPE = SENDFILE
              THEN HP'IS'SOURCE(COMPACK(6))
              ELSE BEGIN      << ERROR >>
                MOVE MSGB := "ETHERENT - RECEIVED UNKNOWN COMMAND";
                PRINTOP(MSG,-35,0);
              END;    << ELSE >>
      END;   << THEN >>
    END;   << WHILE >>
```

```
END.   << MAIN >>
```

Appendix 5

Listing of STARTNET.LILJA.SYS

```
!JOB ETHERNET.SYS
!PREPRUN ETHERRUN.LILJA.SYS;CAP=PM
!EOJ
```

REFERENCES

1. A. S. Tanenbaum, Computer Networks, Prentice-Hall, Englewood Cliffs, NJ (1981).

2. H. J. Saal, "Local Area Networks: Possibilities for Personal Computers," BYTE, Vol. 6, (10) p. 92 (October 1981).

3. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Communications of the ACM, Vol. 19, (7) pp. 395-404 (July 1976).

4. R. Binder, R. Abramson, and F. Kuo, "Aloha Packet Broadcasting - A Retrospect," Proceedings of the AFIPS National Computer Conference, pp. 203-215 (June 1975).

5. J. A. Ayala, Design and Construction of an Ethernet Communications Controller for a Non-homogeneous Environment, Computer Systems Group report CSG-5, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, (July 1982).

6. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Bell Laboratories, Englewood Cliffs, NJ (1978).

7. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual, Bell Laboratories (January 1979). Seventh Edition.

8. Systems Programming Language Reference Manual, Hewlett-Packard Company, Santa Clara, CA (1976). Second Edition.

9. MPE Intrinsics Reference Manual, Hewlett-Packard Company, Cupertino, CA (January 1981). Third Edition.

10. E. C. Eschenauer and V. Obozinski, "The Network Communication Manager: A Transport Station for the SGB Network," Computer Networks, Vol. 2, (4) pp. 236-249 (September 1978).

11. System Manager/System Supervisor Reference Manual, Hewlett-Packard Company, Cupertino, CA (1979). Third Edition.

12. J. F. Shoch and J. A. Hupp, "Measured Performance of an Ethernet Local Network," Local Area Communications Network Symposium, (May 1979).