1984

Reliable file transfer across a 10 megabit ethernet

Mark Van Dellon

Rochester Institute of Technology
School of Computer Science and Technology

Reliable File Transfer Across A 10 Megabit Ethernet

by
Mark Van Dellon

July 24, 1984

A thesis, submitted to The Faculty of the Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Approved by:

Roy Czernikowski — July 24, 1984
Dr. Roy Czernikowski - Committee Head — Date

John Ellis — July 24, 1984
Dr. John Ellis — Date

Tong-han Chang — July 24, 1984
Dr. Tong-han Chang — Date

Abstract: The Ethernet communications network is a broadcast, multi-access system for local computing networks. Such a network was used to connect six 68000 based Charles River Data Systems for the purpose of file transfer. Each system required hardware installation and connection to the Ethernet cable.

The software is an implementation which conforms to Xerox "PUP File Transfer Protocol Specifications". This required the writing of two programs, the FTP user and the FTP server. Each program was built upon common communication packages which also had to be written. These communication routines transferred data over the Ethernet using the PARC Universal Packets (PUP) format.

CR Categories: C.2, D.4.4.

Key Words and Phrases: Ethernet system, carrier sense multiple access with collision detection (CSMA/CD), network protocols, PARC Universal Packets (PUP), file transfer, File Transfer Protocol (FTP).

Table of Contents

Illustrations

1. Introduction

The focus of the thesis is on reliable file transfer between two computer sites over a 10 Megabit (Mbit) Ethernet. The Ethernet hardware used on this project was the commercially available 10 Mbit Interlan Ethernet board combined with software to implement the XEROX 3 Mbit file transfer protocol (FTP). The Ethernet approach provides an economical and efficient way of connecting many computer sites in a network which can be easily expanded.

The thesis project consisted of the installation of the commercially purchased hardware and the writing of FTP software. The FTP software package is based on the existence of three lower level software packages, all of which were written for the project. From the top-down, the other packages are the byte stream protocol (BSP), PARC Universal Packets (PUP), and the device driver. PARC is a XEROX facility in California, the Palo Alto Research Center, which developed the internet packet format known as PARC Universal Packets or PUP.

The rest of this introductory section will serve to acquaint the reader with FTP and the task involved to implement file transfers. Some of the material in this introduction will be discussed in detail in later sections. The redundancy is included to insure a clear understanding of the project. Section 2 will describe the history of Ethernet and packet communications; the ISO Reference Model will also be introduced there. Section 3 lists the specifications of the software and hardware that were used to implement FTP. In section 4, the breakdown of the FTP Specifications and communication layers are shown. These layers are then compared and contrasted with the ISO Reference Model. Section 5 shows the packet formats and describes the protocol exchanges of each layer. An example of the FTP protocol is shown. Section 6 describes the code written for the actual project to implement FTP while section 7 outlines the testing and verification performed. The last section points out the conclusions and possible alternate implementations.

1.1 Project Description

FTP consists of two programs communicating together in an agreed upon fashion to allow storing, retrieving, listing, or deleting of files at a remote site. The local program is the user's task and will be invoked by the user at a terminal. The remote program is the server's task which is always running as a background job listening to the Ethernet for a user's task to request a connection.

The basic framework for FTP and the associated communication packages was obtained from two documents, PUP File Transfer Protocol Specification and PUP Specifications. The first document outlines the algorithms for command/response sequences necessary to allow server and user tasks to transfer data (files, file information, etc). The second document describes the BSP and PUP level for reliable packet transfer. A third document, The Interlan NI3010A MultiBus Ethernet Communications Controller User Manual, was used to program the Ethernet hardware from the device driver code.

1.1.1 Functions Performed

The PUP File Transfer Protocol Specification defines a set of primary and secondary FTP commands. Primary commands must be implemented while secondary commands are optional. The implementation chosen for the project contains the primary commands and a portion of the secondary commands. This allows the user to open a connection with a remote host, whereby he/she may retrieve a remote file and store it on a local host or store a file from the local host onto the remote host. The files may be text or binary and have no size limitation providing there is appropriate disk space available at the storage site. The user also has commands to list or delete files on the remote file server as well. Upon completion of the FTP session, the user quits FTP which closes the connection with the remote FTP server.

1.1.2 Limitations and Restrictions

1. There is currently no function call available to allow the programmer to check disk space allocation. Therefore, storing a file on the remote server or on the local host will assume there is sufficient disk storage space available. This is not considered to be a serious implementation drawback at this time.

2. The FTP rename function will not be implemented. This is not part of the primary FTP commands and not seen as a potential problem.

1.1.3 User Inputs and Outputs

Execution of FTP is accomplished by entering a command of the following syntax at the command interpreter level:

ftp [-b -f -p] NetAddress-or-HostName

NetAddress-or-HostName is the 3 Mbit Ethernet address or the character string name associated with the remote host with which you are desiring a connection. The optional flags are meant for various levels of debug to allow programmers the ability to verify functions or trace problems. These are probably not of any interest to the normal user.

Optional Flag	Function
-b	show debug messages at BSP level
-f	show debug messages at FTP level
-p	show debug messages at PUP level

A sample FTP session to store local file **abc** as file **xyz** on the remote server, whose Ethernet address #5, and then retrieve remote file **123** and store it as the local file **789** follows:

```
ftp #5
UNOS/68000 FTP Server, Version x.x, version date
>Store local file abc as remote file xyz
>Retrleve remote file 123 as local file 789
>quit
```

Note that x.x and version date are the current version number and date of that version for the responding server and will be set with each new software release. Checking this will verify the correct version of the server is running and assure that bad versions get eliminated.

1.2 Project Organization

The project is organized into four logical divisions, each building upon primitives from the lower level. The top three levels. FTP, BSP, and PUP. are combined to form the FTP user program or the FTP server program. The bottom level is the Ethernet driver and will be linked to the operating system. The bottoms-up view of the four logical levels can be seen in Figure 1.1 and their functions are described below:

Level	Function
0	Ethernet driver mechanism for packet transport and communication with the hardware. Responsible for opening and closing of Ethernet devices, reading and writing of Ethernet frames, initializing of Ethernet hardware, and interrupt handling.
1	PUP datagram level. Responsible for opening and closing PUP channels, reading and writing of PUP packets, and routing.
2	BSP flow-control level. Responsible for opening and closing of byte streams, reading and writing of byte stream.
3	FTP server or FTP user level. Responsible for command/response sequencing between server and user process.

A simplified diagram representing the flow of data for reading and writing is shown in Figures 1.2 and 1.3 respectively. Outgoing data passes from layer to layer until it is packetized and transmitted on the 10 Mbit Ethernet cable, while incoming data traverses a similar route but in reverse order.

The lowest level of process-to-process communication is the PUP packet. It is the packetized data that is transmitted over the Ethernet. The following highlights the information required to construct a PUP.

Field	Byte Length	Function
PUP Length	2	Number of 8-bit bytes in the PUP, including header, contents, and checksum.
Transport Control	1	For use by gateways and should be zero at PUP's source.
Pup Type	1	Assigned by the source process for interpretation by the destination process and defines the format of the Pup contents.
Pup Identifier	4	Assigned by the source process and is used by most protocols to hold a sequence number for purposes of duplicate suppression and ordering.
Source Port	6	The originator of the PUP packet. Contains an 8-bit network number, an 8-bit host number, and a 32-bit socket number.
Destination Port	6	The final destination of the PUP packet. Contains an 8-bit network number, an 8-bit host number, and a 32-bit socket number.
Data Field	0 - 532	Contents for transfer.
Checksum	2	Optional software checksum computed over header and data field.

1.2.1 System Files

1. Each FTP connection will require its own Ethernet device driver connection through /dev/enet#, where # is between 0 and the maximum number of device drivers supported. The connection with the driver is handled automatically from within the software and the user need not worry about this detail.

2. A difference in length of 10 Mbit and 3 Mbit Ethernet host addresses requires a mapping to take place internally. This is handled by looking up preassigned mappings in the file /etc/enetmappings.

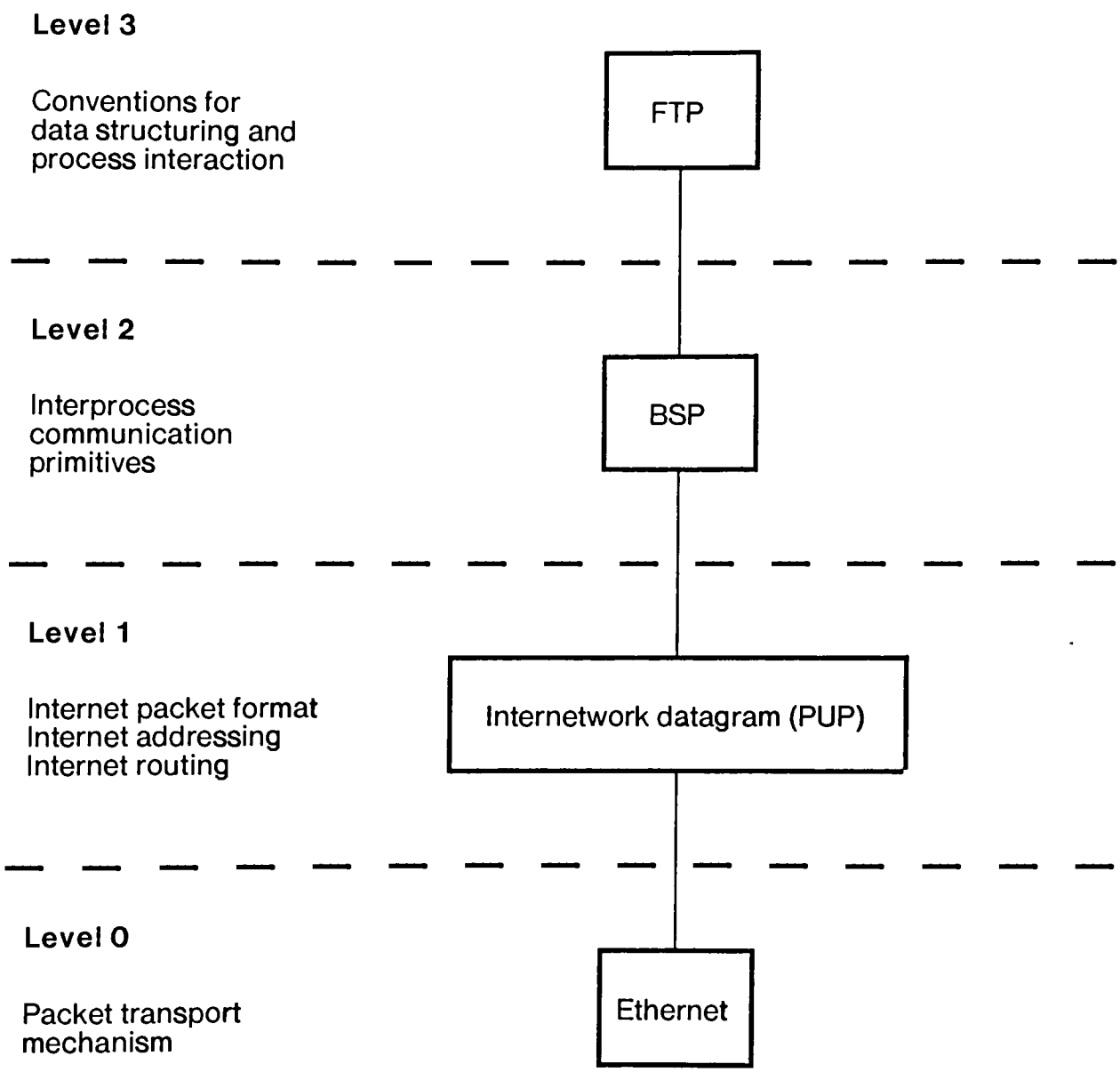


Figure 1.1: System Organizational Chart

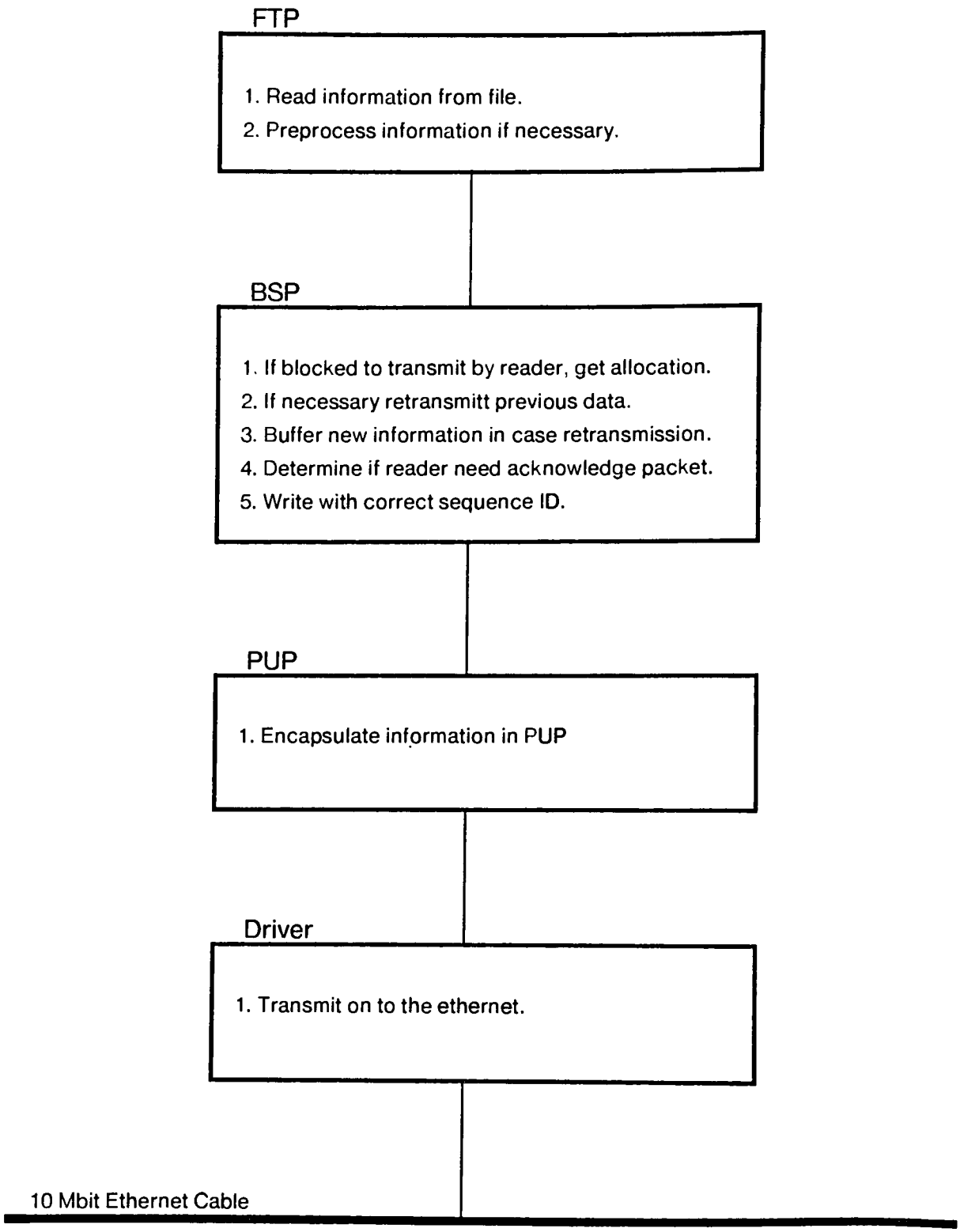


Figure 1.2: Writing Interface

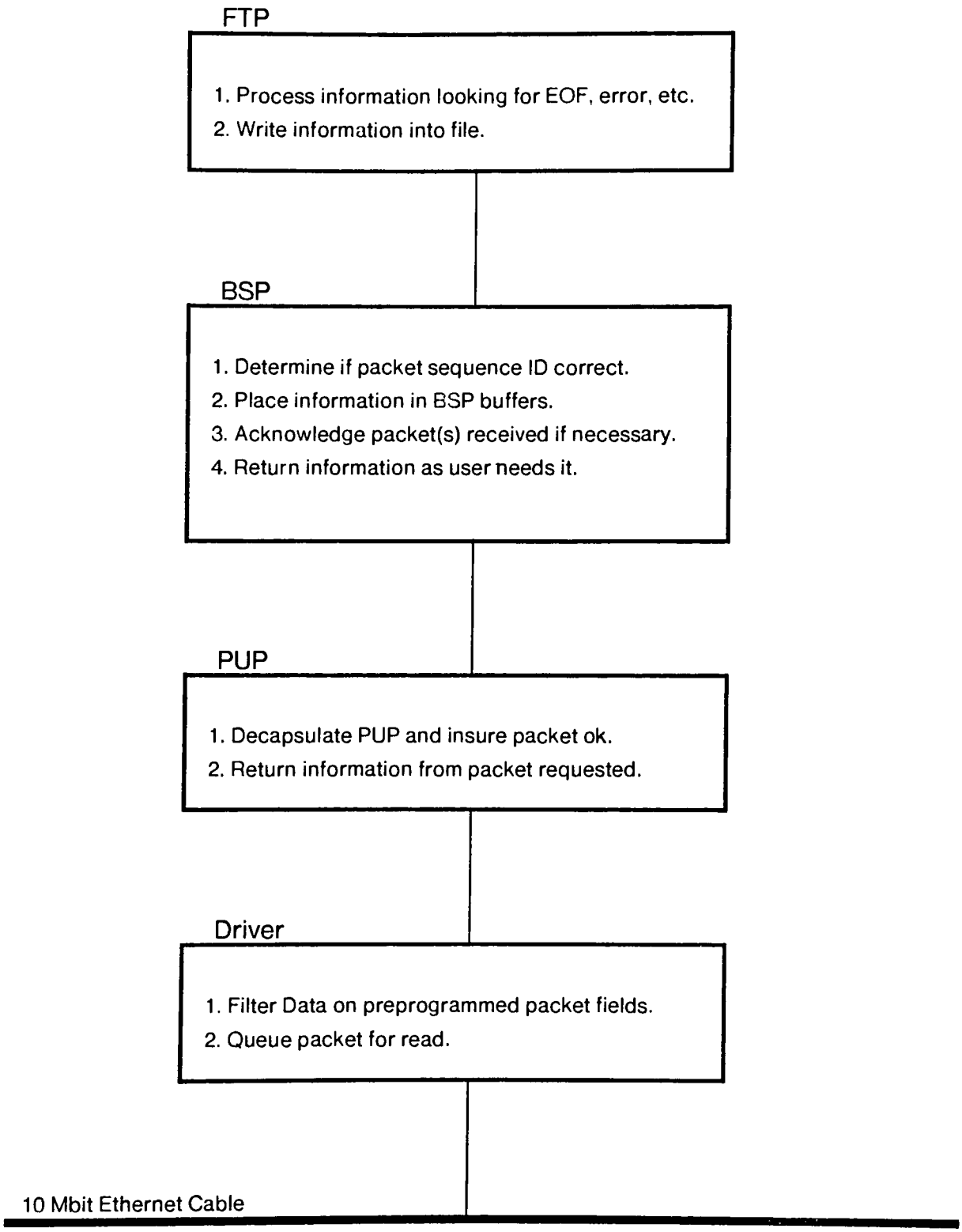


Figure 1.3: Reading Interface

2. Background

2.1 The Ethernet

In general terms, Ethernet is a multi-access, variable length, packet-switched communications system for carrying digital data among locally distributed computing systems. Such computing systems include, for example, personal computers, printing facilities, large file storage devices and long-haul communication equipment. The shared communications channel in an Ethernet is a passive broadcast medium, known as an Ether, with no central control. Ethernet is named for the historical *luminiferous ether* through which electromagnetic radiations were once alleged to propagate. The Ether transport medium can be one of many different broadcast media but is typically a coaxial cable. Packet address recognition in each station is used to take packets from the channel. Access to the channel by the stations wishing to transmit is coordinated in a distributed fashion by the stations themselves.

2.1.1 The Experimental Ethernet

The Experimental Ethernet system was developed at Xerox Palo Alto Research Center starting in 1972 and is still in use today. Ethernet design started with the basic idea of packet collision and retransmission developed in the Aloha Network by the University of Hawaii. The Aloha Network applied packet radio techniques for communication between a central computer and its terminals scattered among the Hawaiian Islands. Like an Aloha radio transmitter, an Ethernet transmitter broadcasts completely-addressed transmitter synchronous bit sequences called packets onto the Ether and hopes that they are heard by the intended receivers.

The Experimental Ethernet provided four mechanisms for reducing the probability and cost of losing a packet. These are (1) carrier sense, multiple access/collision detection (CSMA/CD), (2) collision consensus enforcement, (3) truncated packet filtering, and (4) packet error detection.

2.1.1.1 CSMA/CD

Carrier is generated by using Manchester encoding as a packet's bits are placed on the Ether. This encoding technique guarantees that there is at least one transition on the Ether during each bit time. The passing of a packet on the Ether can therefore be detected by listening for bit transitions. With carrier detection, a station is able to implement *deference* which avoids transmitting while hearing carrier. Once a packet transmission has been in progress for an Ether end-to-end propagation time, all stations are hearing carrier and are deferring; the Ether has been *acquired* and the transmission will complete without interfering collisions. Collisions should occur only when two or more stations find the Ether silent and begin transmitting simultaneously within the Ether end-to-end propagation time. This will almost always happen immediately after a packet transmission during which two or more stations were deferring. A collision is indicated when the transceiver notices a difference between the value of the bit it is receiving from the ethernet and the value of the bit it is attempting to transmit.

2.1.1.2 Collision Consensus Enforcement

After a station detects a collision, it momentarily jams the Ether to insure that all participants in the collision will detect interference and will be forced to abort. Without this collision consensus enforcement mechanism, it would be possible that the transmitting station which would otherwise be the last to detect a collision might not do so as the other interfering transmissions successively abort and stop interfering. Although the packets may look good to that last transmitter, collisions on the Ether will cause the packet to arrive damaged at the intended receiver.

2.1.1.3 Truncated Packet Filtering

Interference detection and deference causes most collisions to result in a truncated packet of only a few bits. Truncated packets are filtered out in hardware thereby reducing the processor load.

2.1.1.4 Packet Error Detection

As a packet is placed on the Ether, a cyclic redundancy check (CRC) is computed and appended. As the packet is read from the Ether, the CRC checksum is recomputed. Packets which do not carry consistent CRC checksums are discarded.

However, with all these precautions an Ethernet is probabilistic. Packets may be lost due to interference with other packets, impulse noise on the Ether, an inactive receiver at a packet's intended destination, or purposeful discard. Protocols used to communicate through an Ethernet must assume that packets will be received correctly at intended destinations only with a high probability.

As mentioned above, the Experimental Ethernet is still at use in Xerox today. About 300 million bytes traverse the network daily at a data rate of 2.94 Mbits per second (more commonly referred to as the 3 Mbit Ethernet). Under normal load, latency and error rates are extremely low and there are few collisions. Studies have shown even under artificially generated heavy load, the system shows stable behavior.

2.1.2 Ethernet Specification

More recently, a cooperative effort in the late 1970's involving Digital Equipment Corporation, Intel, and Xerox has produced an updated version of the Ethernet design, generally known as the Ethernet Specification. The two systems, Experimental Ethernet and Ethernet Specification, are very similar; they both use coaxial cable, Manchester signal encoding and CSMA/CD. Some changes were made based on the experience with the experimental system or in an effort to enhance the characteristics of the network. The most noticeable changes are the increase in data rate from 2.94 Mbits to 10 Mbits and the expanded address fields from 8 bits to 48 bits.

2.1.3 Ethernet Summary

The Ethernet provides a low cost, reliable, high speed local network. Each station has equal access to the network with minimum delay time. No single station operating in accordance with the protocol is able to prevent the progress of other stations. The Ethernet does not employ any data security or protection from the hostile user. These functions are left to the application programs.

The Ethernet Specification is designed as a layered architecture. This allows separation of the logical aspects of the software protocol from the physical details of the communication medium.

2.2 Network Architecture

From the beginning, many networks were designed hierarchically, as a series of layers, each one building on the one below. At first, each network design team started out by choosing its own set of layers. However, in the past few years, a consensus has begun to develop among network designers, a consensus embodied in the 1979 International Organization for Standardization's Reference Model of Open Systems Interconnection (ISO Reference Model). The Ethernet Specification is designed after such a model.

2.2.1 The ISO Reference Model

The ISO Reference Model has seven layers, each layer performing a small set of closely related functions. The seven layers are shown in Figure 2.1 and presented here with a brief overview of the functions of each layer.

(1) The *physical layer* is the lowest layer of the seven and its protocol is concerned with the transmission of a raw bit stream. The protocol designers must decide how to represent 1's and 0's. how many microseconds each bit will last, how many pins the network connector has, what each pin is used for, and other electrical and mechanical details.

(2) The *data link layer* converts an unreliable transmission channel into a reliable one for use by the network layer. The technique for doing so is to break up the raw bit stream into frames.

(3) The *network layer* in a point-to-point network is primarily concerned with routing of packets and the effects of poor routing, namely, congestion. In a broadcast network, routing is not an issue, since only one channel exists.

(4) The *transport layer* is to provide reliable host-to-host communication for use by the session layer. It presents an ordered sequence of messages to the session layer and hides the fact that there may be lost packets that require retransmission.

(5) The *session layer* is responsible for setting up, managing, and tearing down process-to-process connections, using the host-to-host service provided by the transport layer.

(6) The *presentation layer* performs generally useful transformations on the data to be sent, such as text compression. It also performs the conversion required to and from local standards.

(7) The *application layer* is the highest layer of the seven. Its contents are the functions to be performed, for example, electronic mail distribution.

Each layer should be thought of as a program, process, microcode or hardware device, that communicates with the corresponding layer or *peer* on another machine. The rules governing the layer k conversation are called the *layer k protocol*. Thus, the ISO Reference Model contains seven protocols, one for each layer.

Data communications is depicted in the ISO Reference Model as passing vertically down the layers of the sending machine and up the layers of the receiving machine. Only layer 1, the physical layer, does actual intermachine communication or horizontal transfer. As a message passes from layer $k + 1$ to layer k of the sending machine, a layer k header is added containing control information for use by layer k on the receiving machine. No layer is aware of the header formats or protocols used by other layers. Layer k on the sending machine regards its job as getting the bits that come in from layer $k + 1$ over to the receiving machine using the services of layer k 1. It neither knows nor cares what the bits mean. The boundary between layers is called an *interface*.

When referring to the unit of information being transferred at each layer of the ISO Reference Model, it must be unambiguous as to avoid confusion with other layers. For the sake of clarity, we refer to the entities exchanged from the application layer through the transport layer as *messages*, to distinguish them from the *packets* of the network layer and the *frames* of the data link layer. The physical layer passes information in the form of *bits*.

2.2.2 Network Architecture Summary

The layers, interfaces and protocols described above in the ISO Reference Model form the network architecture. The layered concept approach of the network architecture allows easy change of a *layer k protocol* with no effect on the *layer k 1* or the *layer k + 1* protocols, since *layer k* neither knows nor cares what those protocols are. Once the layers have been established with an existing application program, different application programs may be written using the same layers. This takes advantage of the layered architecture and is a fine example of modular programming.

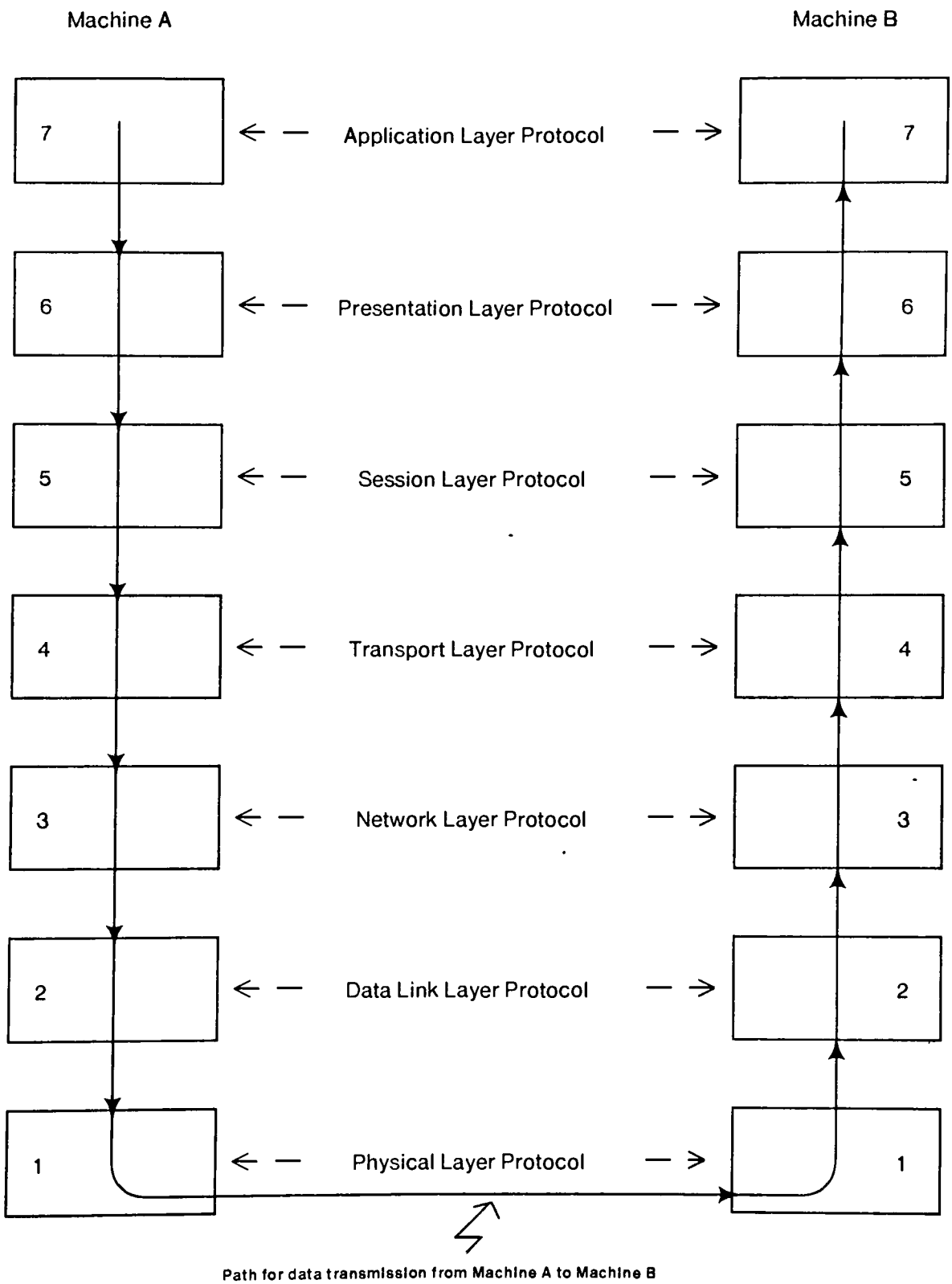


Figure 2.1: The Seven Layer ISO Reference Model

3. Specifications

3.1 Functional Specification

The application layer of this project allows a user to transfer data, to or from a remote file server, with reliable, error free, communication protocols over an Ethernet transport medium. This function is performed through the use of a file transfer program (FTP) containing a keyboard interface for human interactions. A user opens a connection with a remote file server, issues commands causing transfer of data, and then closes the remote file server connection. FTP permits only one remote file server connection to be open at a time. While file transfer is the primary function of FTP, it also provides the user with the capability to list directories or delete files on the remote file server.

FTP takes appropriate security measures by disallowing invalid users access to the remote file server. A login command is provided by the local FTP user's interface to accept the user's name and password. The parameters are not immediately checked for legality, but rather are sent to the remote file server for checking when the next file transfer command is issued. File access privileges are also adhered to by the remote file server. Deletion of a file not owned, retrieval of a file with inappropriate read access or storing of a file with inappropriate write access is prohibited. The remote file server will allow the superuser or administrator file access regardless of the filemodes.

FTP does not perform any data encryption to protect the contents of a file as it is being transferred over the Ethernet. Thus, a user with hostile intentions could connect to the Ethernet, receive the broadcast transmission, and be able to understand its content without the need of a data decryption expert.

3.2 Equipment Configuration

The host machines that are connected through the 10 Mbit Ethernet are various Motorola MC68000 microprocessor based systems obtained from Charles River Data Systems (CRDS). There are currently a total of six systems, all running UNOS version 4.0. UNOS is a "UNIX-like" operating system with real time enhancements to the scheduler. Each system will contain a HalVersa SYNERGIST multibus-to-Versabus adapter card supporting the Interlan NI3010A multibus Ethernet controller card. The systems connected are: (1) One Motorola EXORMacs (Upgraded with CRDS CP-32 processor card), (2) Two Charles River Data Systems UV 68/37, and (3) Three Charles River Data Systems UV 68/27.

3.3 Implementation Tools

The language chosen for implementation of the project was C. There were two major reasons for the choise: (1) C is the only licensed language for the inhouse CRDS systems and (2) C is the source language of the kernel and it will be necessary to create a new operating system with the Ethernet device driver and therefore abide by the procedural interconnects of C.

4. Architectural Design

This project is an implementation of the Xerox "Pup File Transfer Protocol Specifications" (FTP Specifications) originally used to give Nova and Alto minicomputers the capability of file transfer. The initial design began circa 1974 and since then has seen minor additions, corrections and revisions. This section will discuss the FTP Specification and communication layers; comparing and contrasting each with the ISO Reference Model.

As stated previously, FTP was designed in 1974; prior to the ISO Reference Model of 1979. The FTP design team started out by choosing its own set of four hierarchical network divisions. These divisions differ from the seven layers of the ISO Reference Model, however the basic functions described in the ISO Reference Model exist in the FTP architecture. While comparing and contrasting the FTP architecture against the ISO Reference Model, a person must remember that the ISO Reference Model is just that, a reference or guide. It does not specify that an implementation must have all layers or protocols, however most networks will have a majority of the layers or protocols and typically all the functions. Some network architecture implementations may combine two or more layers into one layer or not implement a particular layer all together.

One last point to mention is that there could be different interpretations as to how a pre-1979 ISO Reference Model network architecture implementation maps to the ISO Reference Model. The criteria used to develop the mapping of the FTP Specifications and communications to the ISO Reference Model for this presentation is based upon the fact that layer k is built upon the services of layer $k \quad 1$ and only $k \quad 1$. Differing interpretations may be achieved based upon different criteria.

The FTP network architecture implementation for this project contains four divisions, each relating to one or more layers of the ISO Reference Model. These are (1) hardware, (2) PARC Universal Packets (PUP), (3) Byte Stream Protocol (BSP), and (4) FTP. The relationships can be seen in Figure 4.1 and are discussed below.

4.1 Hardware Division

The hardware used for this project was the commercially available, Interlan NI3010A Multibus Ethernet Communications Controller, chosen for its DMA (direct memory access) ability. Note that this board has a Multibus interface and the host machine, Charles River Data Systems (CRDS), has a Versabus interface. At the time the hardware was ordered, no Versabus DMA Ethernet Controller cards were commercially available. However, there are commercially available adapter cards. The Hal Versa Synergist board does adapt the Multibus to the Versabus and visa-versa. Four pairs of these boards were ordered, required small address and interrupt modifications, and were installed for this project.

The Interlan NI3010A Multibus Ethernet Communications Controller maps to two layers of the ISO Reference Model, the physical layer and the data link layer. These two layers are defined in the Ethernet Specification by Digital Equipment Corporation, Intel, and Xerox. The physical layer is responsible for Manchester bit encoding/decoding and channel access (CSMA/CD). It provides a 10 Mbit physical channel through a coaxial cable medium. Compatibility is an important issue at this level since any vendors with correct implementation of the physical layer will be capable of exchanging data over the coaxial cable. The data link layer performs the functions of data encapsulation/decapsulation (framing) and detection of physical channel transmission errors (bad CRC's).

4.2 PUP Division

The PUP division tends to correspond to the network layer of the ISO Reference Model. The primary concern of the network layer is routing, however, in a broadcast network like the Ethernet, the notion of routing is not an issue since only one channel exists. This tends to bring up the question of whether broadcast systems have a network layer? To answer this, the entire picture must be examined and in particular the Ethernet network.

The Ethernet Specification states a maximum coaxial cable length and a maximum number of transceivers on the coaxial cable (2500 meters and 100 transceivers for the 10 Mbit Ethernet). Increase in the cable length would degrade the input signal to the transceivers below the threshold levels. While repeaters are allowed in the Ethernet Specification to amplify the signal, too many repeaters would cause the end-to-end propagation time to exceed the specification. Therefore, a maximum number of repeaters are allowed (2 for the 10 Mbit Ethernet).

The connection of many hosts across the country must be done on multiple networks. Communication between the networks is done through a *gateway*. Gateways have the ability to convert packets from one protocol to another if the incoming and outgoing networks do not match protocols. Each gateway may be connected to many networks. Whenever a packet must be passed from one network to another, the source network delivers it to the interconnection network, where it is routed internally among the gateways until it is delivered to the destination network.

The packets being delivered on the Ethernet are datagrams and arrive at their destination with only a high probability of success. Each packet is unrelated to any other packet, past or future, and must therefore carry a full destination address. A destination address contains the network, the host machine on that network, and the process or *socket* on that host machine. As a packet passes through a gateway, the destination address can be examined and shortest path routing can be performed to minimize the number of network hops.

The PUP division must perform minimal routing inorder to broadcast the packet with the correct first hop address. A packet destined for a host on the local network will have the first hop address the same as the destination address. A packet destined for a host on a different network will have the first hop address of the local network gateway. The gateways will route this packet, hop by hop, until the destination network gateway is reached. An example of gateway routing can be seen in Figure 4.2.

The PUP software for this project performs two functions; routing and address mapping. Routing was minor since the network that connected the host machines was an isolated Ethernet (no gateway). With the lack of a gateway, the first hop address will always be the same as the destination address. The address mapping performs the conversion from the 10 Mbit Ethernet Specification to the 3 Mbit FTP/PUP requirement and visa-versa.

4.3 BSP Division

This is a relatively sophisticated protocol for supporting reliable, sequenced streams of data. It provides for multiple outstanding packets from the source, and uses a moving window flow control procedure. Error and flow control between the two programs is accomplished by using this bi-directional stream. BSP is equivalent to the transport layer of the ISO Reference Model.

Much of the design time of the project was spent organizing (and reorganizing) the procedures and algorithms necessary to implement the BSP protocol. This was partially due to the lack of documentation available on the BSP protocol.

4.4 FTP Division

Transferring a file from one machine (or host) to another over a network requires the active cooperation of programs on both machines. The two parties to a file transfer are named the *user* and the *server*: the user initiates actions and the server responds. The user and the server programs are the highest level or the application layer in the FTP network architecture. These are two distinct programs; the server program is started by the administrator and is always waiting for a user attempting a connection over the Ether.

FTP also encompasses the presentation layer. It has the responsibility of reformatting data to conform to local host conventions. One such necessary conversion is the end-of-line convention for a text file. UNOS and UNIX use linefeed as there end-of-line convention, while other systems use carriage return, etc.

The available FTP documentation was well written and allowed easy transformation from algorithm to code. The FTP modules written for this project composed a bulk of the generated code. There were many common routines between the server and user involving command transmission and file transfer thereby helping to reduce the overall coding effort.

4.5 ISO and FTP Contrasts

The four divisions discussed above mapped into six of the seven ISO Reference Model layers. The layer not found in the FTP network architecture was the session layer. It did not meet the criteria specified above that the session layer is to be built upon the services of the transport layer and only the transport layer. This does not imply that the function of process connection between the two hosts, performed by the session layer, does not exist. The connection function does exist, however it is built upon the network layer or k 2 layer not k - 1 layer as the mapping criteria specified.

One last difference between the FTP architecture and the ISO Reference Model was that a header was not added to the message as it passed from layer $k + 1$ to layer k. Instead, the PUP Internet Datagram Packet, seen in Figure 4.3, contains reserved fields for use by the upper levels and in particular the BSP level. The PUP level is responsible for inserting the correct data in the PUP Length, Transport Control, Destination Port, Source Port, and PUP Software Checksum fields. The PUP Type, PUP ID, and Data fields are for use by the BSP level. The PUP format will be discussed in full at a later time.

4.6 Architecture Summary

The approach to the FTP network architecture was divided into four layers; (1) hardware, (2) PUP, (3) BSP, and (4) FTP. Each layer mapped to one or more of the seven layers in the ISO Reference Model based on the specified criteria. The thesis project involved writing the three software layers, FTP, BSP, and PUP. One other coding section was necessary for the FTP project, an Ethernet driver. The driver is an interface between the Ethernet hardware and the kernel file system; it is not an architectural layer.

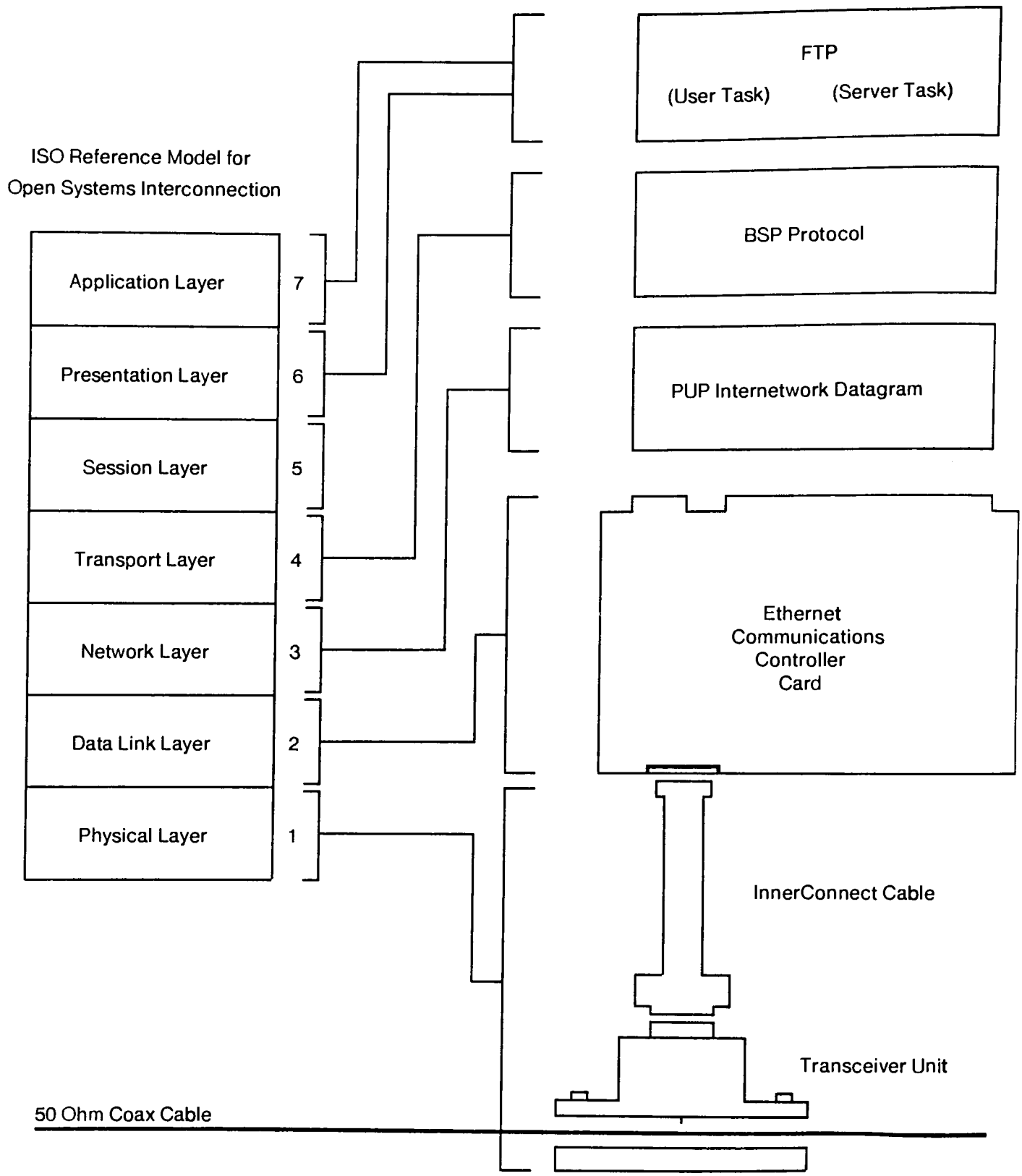


Figure 4.1: FTP and the ISO Reference Model

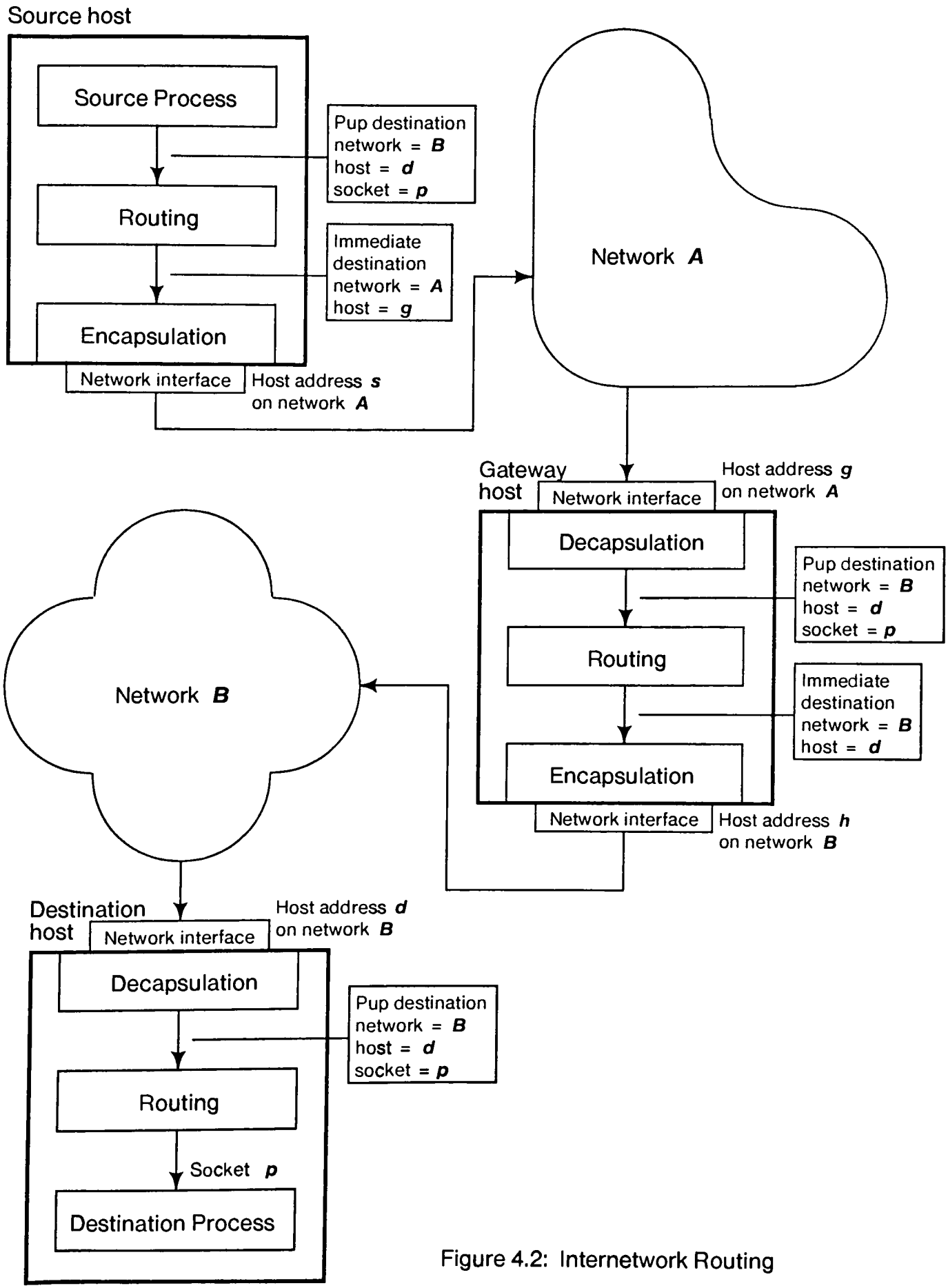


Figure 4.2: Internetwork Routing

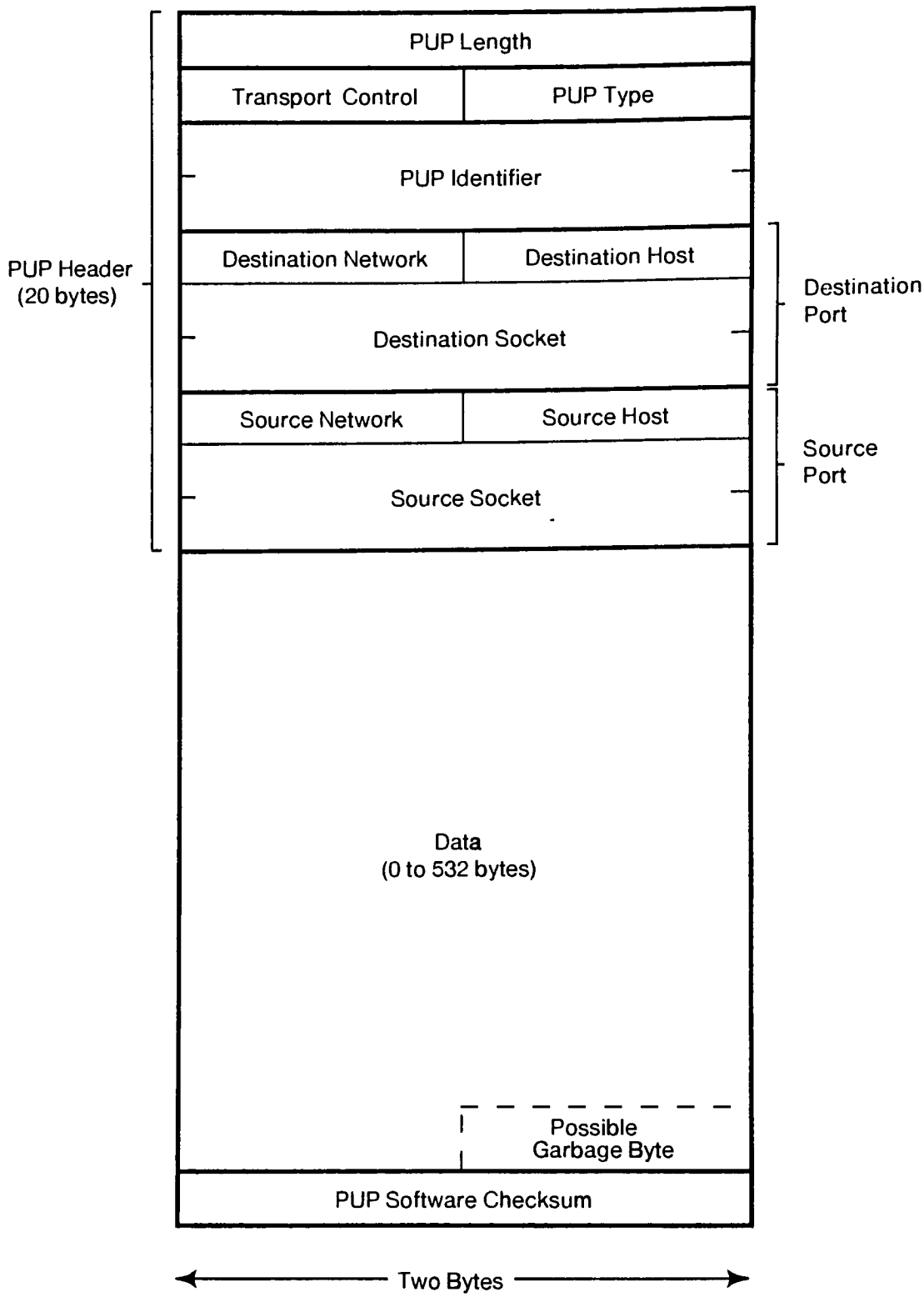


Figure 4.3: PUP Internet Datagram

5. Protocols and Packet Formats

The hub of the communication protocol is the PUP packet. This section is organized for discussion of the PUP packet design. how the packet is transmitted through encapsulation, and how the PUP is used by upper layers of the FTP architecture. The protocols of the PUP, BSP, and FTP layers will also be explained along with connection initiation and termination.

5.1 PUP Format

The standard format for a PUP is shown in Figure 4.3. The following paragraphs highlight the sorts of information required at the internet datagram level.

The *PUP length* is the number of 8-bit bytes in the PUP, including header (20 bytes), contents, and checksum (2 bytes). There may be from 0 to 532 content bytes in a PUP so that the length will range between 22 and 554 bytes. A PUP is always carried in an integral number of 16-bit words. When there are an odd number of content bytes in a PUP, the extra garbage byte required to fill the PUP out to an integral number of 16-bit words precedes the checksum word.

The *transport control* field is used for two purposes: as a scratch area for use by gateways, and as a way for source processes to tell the internet how to handle the packet. (Other networks call this the "facilities" or "options" field.) The *hop count* subfield is incremented each time the packet is forwarded by a gateway. If this ever overflows, the packet is presumed to be traveling in a loop and is discarded. A *trace bit* is specified, for potential use in monitoring the path taken by a packet.

The *PUP type* is assigned by the source process for interpretation by the destination process and defines the format of the PUP contents. The 256 possible types are divided into two groups. Some types are *registered* and have a single meaning across all protocols; PUPs generated or interpreted within the internet (e.g., by gateways) have types assigned in this space. Interpretation of the remaining *unregistered* types is strictly a matter of agreement between the source and destination processes.

The *PUP identifier* (PUP ID) is used by most protocols to hold a sequence number. PUP IDs identify PUPs and their contents to distinguish them from others, for purposes of duplicate suppression and ordering. The specific interpretation of the PUP ID is not defined at the PUP level but is rather a matter of convention established at higher levels of protocol, for example BSP.

PUPs contain two addresses: a *source port* and a *destination port*. These hierarchical addresses include an 8-bit network number, an 8-bit host number, and a 32-bit socket number. Hosts are expected to know their own host addresses, to discover their network numbers by locating a gateway and asking for this information, and to assign socket numbers in some systematic way not legislated by the internet protocol.

A source network of zero indicates that the source process does not know the identity of the network to which its host is connected. A destination network of zero indicates the the PUP is addressed to a host in the current network. This convention is for use by processes which know that they want to communicate with a known host on their own network, but can't find out which network they are on. This avoids the situation in which processes on the same network can't communicate with one another because there is not an operating gateway

If the destination host is zero, the process intends the PUP to be a *broadcast* in the destination network. The PUP is received by active ports at the given destination socket in all hosts on the specified network. By convention broadcast PUPs never propagate through gateways.

The *data* field contains up to 532 data bytes. The selection of a standard maximum packet length must reflect many considerations: error rates, buffer requirements, and needs of specific applications. A reasonable value might range anywhere from 100 to 4000 bytes. In practice, much of the internet traffic consists of packets containing individual "pages" of 512 bytes each, reflecting the quantization of memory in most of our computers. But just carrying the data is not enough, since the packet should accommodate higher-level protocol overhead and some identifying information as well. Allowing 20 additional bytes for such purposes, the value of 532 bytes becomes the maximum size of the data field (a somewhat unconventional value in that it is not a power of two). Thus, there may be between 0 and 532 content bytes in a PUP, so its total length will range from 22 to 554 bytes. PUPs longer than 554 bytes are not prohibited and may be carried by some networks, but no internetwork gateway is required to handle larger ones.

The optional *software checksum* is used for complete end-to-end coverage. It is computed as close to the source of the data and checked as close to the ultimate destination as is possible. This checksum protects a PUP when it isn't covered by some network-specific technique, such as when it is sitting in a gateway's memory or passing through a parallel I/O path. Most networks employ some sort of error checking on the serial parts of the channel, but parallel data paths in the interface and the I/O system often are not checked.

The checksum algorithm is intended to be straight forward to implement in software. It also allows incremental updating so that intermediate agents which modify a packet (gateways updating the hop count field, for example) can quickly update the checksum rather than recomputing it. The checksum may (but need not) be checked anywhere along a PUP's route in order to monitor the internet's integrity. Note that the checksum includes the garbage data byte if there is one. The algorithm for computing the checksum is included in Appendix A.

5.2 Encapsulated PUPs

PUPs are to be *encapsulated*, creating a frame, to conform to the conventions and formats of the transporting network. Encapsulation for the 10 Mbit Etherent is performed by adding a Destination Address, Source Address, Frame Type and a 32-bit CRC. The format for this is shown in Figure 5.1.

When a PUP is received at its final destination port, it is *decapsulated* by applying the inverse of the encapsulation transformation before being passed to the destination process. When a PUP is received by a gateway, it is (1) decapsulated, (2) routed to another network, and (3) re-encapsulated according to the conventions of this new network.

5.3 RTP Protocol

The Rendezvous/Terminate Protocol (RTP) is a reliable convention by means of which a connection between two ports may be established and later broken.

5.3.1 Rendezvous Protocol

A rendezvous is accomplished with an exchange of packets, each called a *Request for Connection* or RFC. A (user) task initiates a connection by transmitting an RFC to a listening (server) rendezvous port. A server's rendezvous port usually contains a well known socket. The listener confirms by returning an RFC with matching PUP ID. In addition to the

rendezvous ports carried in the PUP header, each RFC carries the address of a *connection port* through which the RFC's intend to maintain the newly established connection.

The PUP ID of the initiating RFC, whose format is shown in Figure 5.2, also defines the *Connection ID* for the resulting connection. It should be chosen in such a way as to reduce the probability of confusion among connections established near in time. If connection IDs are generated from an appropriate real-time clock, for example, the probability of PUPs from an extinct connection being mistaken for PUPs in a new connection between the same pair of ports may be made vanishingly small.

If the initiator's RFC is lost, it should be retransmitted by the initiator after enough time has passed for a normal answer. Duplicate RFC's can, at best, be discarded after retransmission of the appropriate answering RFC. At worst, multiple servers will be generated to which no packets are ever sent; these servers should eventually time out and destroy themselves.

5.3.2 Normal Termination Protocol

A connection is normally terminated by a three-way handshake consisting of an *End* PUP and two *EndReply* PUPs, the structure of which is shown in Figure 5.3. The end of a connection may be initiated from either of its ports by transmission of an End PUP whose ID matches the Connection ID. The End PUP must be transmitted until a matching EndReply PUP is received. Upon receiving an EndReply, the initiator of the End should then send an EndReply in response and promptly self destruct.

The receiver of the End PUP responds by returning an EndReply PUP with matching ID and then dallying up to some reasonably long timeout interval in order to respond to a retransmitted End PUP should the initial EndReply be lost. If the dallying end of the stream connection receives its EndReply, it may immediately self destruct.

It may happen that both processes choose to send End PUPs simultaneously. Upon receiving an End PUP in seeming answer to an End PUP of its own, a port should at once send an EndReply and begin dallying in the normal fashion and abandon sending Ends.

5.3.3 Abnormal Termination Protocol

The *Abort* PUP should be used to terminate a connection (or a connection attempt) in the event of a detected catastrophe. An Abort can be sent to reject an RFC or to terminate a connection in the event of a catastrophe, such as storage overflow or continuing checksum errors. A listener wishing to reject a rendezvous should try to send an Abort PUP with an explanation (e.g., "disk full", etc.). Either end of a connection in progress, with its back against the wall, should try to send an Abort before self destructing, though its demise will eventually be detected by a timeout.

The Abort PUP, diagrammed in Figure 5.4, carries a program interpretable code and a human readable explanation of some abnormal condition. An Abort must carry as its PUP ID, the ID of the connection being aborted; the ID of the connection's initiating RFC. Abort PUPs need not be acknowledged because it is presumed that there would be nobody to receive the acknowledgment.

5.4 BSP Protocol

Bytes in a stream are numbered consecutively by a 32-bit number referred to as the Byte ID, which is initialized to the Connection ID (i.e., the PUP ID used for the rendezvous) when the connection is created. A byte stream is carried from one port to another by data PUPs, each containing zero to 532 consecutive bytes starting with the one identified by the PUP's ID. In return for these are transmitted acknowledgement packets with matching identifiers (though

not necessarily on a one-to-one basis) which verify the correct receipt and control flow. The streams of data flowing in each direction, while starting with the same initial Byte ID, are independent.

Data packets should not be sent unless space has been allocated for them at the receiver. The sender is informed about receiver allocations in the acknowledgment packets returned by the stream receiver. These allocations are not additive; each one reflects the current state of the receiver's space allocation at the time of its departure. Acks and therefore allocations travel in both directions, independently for each direction of data flow.

5.4.1 Data PUPs

There are two kinds of data PUP under the BSP, one which demands an immediate acknowledgement, called the *AData* PUP, and one which doesn't, called the *Data* PUP. The format of Data (and AData) PUPs are shown in Figure 5.5. All data must be positively acknowledged, but not on a strict packet-for-packet basis. It is intended that data will be transmitted in a number of Data PUPs followed by an AData PUP asking for acknowledgement of receipt of all.

Data which have been transmitted but not acknowledged must be retransmitted after some timeout. If there are too many retransmissions, a stream connection may be aborted.

Null AData PUPs (containing no data bytes) will be used to probe a receiver for an update of the allocation block and Receiver Byte ID. This will probably happen when the byte stream is first established and the sender has no allocation information or when the sender has been held up for some time with zero allocation and wants to verify that the receiver is still alive.

5.4.2 Acknowledgement PUPs

The Acknowledgement (Ack) PUP, pictured in Figure 5.6, supplies two pieces of information to the sender. First, it indicates to the sender that all bytes previous to that, identified by its PUP ID, have been received correctly. A received Ack whose PUP ID is less than the previous one should be considered a delayed duplicate and discarded. Second, an Ack carries a 3-word allocation block indicating the receiver's state as of the time the Ack was sent. The stream sender should update its state by discarding acknowledged data packets being held for retransmission before considering the allocation block.

Each word in the allocation block indicates a portion of the receiver's buffering status. The allocation block contains (1) the maximum number of bytes per PUP that the receiver is willing to accept, (2) the number of PUPs that can be buffered (Number of PUPs), and (3) the number of bytes total (Number of Bytes) which can safely be sent. The maximum number of bytes per PUP and the number of bytes total are both expressed in terms of *Data* bytes, exclusive of the fixed-length PUP headers involved.

The "Number of PUPs" allocation indicated the maximum number of additional Data PUPs the stream receiver is prepared to handle over and above any Data PUPs it has already received and may be holding in its buffers. In making use of this allocation, the stream sender should assume that any Data PUPs it has transmitted but which have not been acknowledged have in fact not yet reached the receiver. Hence, the "Number of PUPs" allocation should be compared to the number of unacknowledged Data PUPs in order to determine whether or not it is OK to transmit additional Data PUPs.

The "Number of Bytes" allocation should be interpreted relative to the Ack's ID, i.e., the Byte ID of the first byte yet to be acknowledged. Adding the allocation to the PUP ID of the Ack in which it came yields the ID of the last byte in the stream which the receiver is prepared to

accept. This Byte ID implied in Acks should be monotonically increasing so that stream senders need not hold back existing data PUPs they have previously committed to transmit. Similarly, the maximum number of bytes per PUP should not be decreased during the life of a BSP connection, so that the stream sender need not take existing data PUPs and break them apart.

Optionally, an Ack PUP may carry *specific acknowledgments*, described as *Pos/NegAck Blocks*. Each is a 3-word item indicating a specified interval of bytes in the unacknowledged part of the byte stream that is known by the receiver to be either received or lost. The purpose of these indications is to hasten the retransmission of bytes known to be missing and to avoid the retransmission of bytes already received. Once a receiver indicates with a PosAck Block that an interval of bytes has been received, the sender may discard the packets which contain them knowing that they will not require retransmission.

The transfer of bytes from stream sender to stream receiver should not depend on these Pos/NegAck Blocks being used by either end, except that bytes might flow with less efficiency without them. A receiver may choose not to include Pos/NegAck Blocks in its Ack PUPs and a sender may choose to ignore them if present.

Experience has shown that when communicating over high-bandwidth, reliable networks such as an Etherent, the software overhead required to generate and interpret specific acknowledgments is not rewarded by any noticeable improvement in performance. No known software presently implements specific acknowledgments and Figure 5.6 does not show them.

5.4.3 Mark PUPs

The *Mark* is a distinguished byte in the byte stream. It is analogous to the file mark found on magnetic tapes. The Mark PUP, pictured in Figure 5.7, carries exactly one content byte which indicates which of a possible 256 types of mark is being signalled.

While reading the data from a stream, a process reads up to a Mark and is then signalled in much the same way as when reading end-of-file. The type of Mark should then be accessible. After clearing mark status, the user should be able to read on in the stream.

For purposes of transmission and flow control, Marks are treated exactly the same as Data PUPs. They occupy one position in the Byte ID sequence, and are acknowledged in the same manner as any other byte in the stream. An AMark PUP is simply a Mark that demands an immediate acknowledgment (in the same manner as an AData).

5.4.4 Interrupt PUPs

An *Interrupt* PUP, diagrammed in Figure 5.8, is used to signal some asynchronous event requiring immediate action by the other end of the stream. The Interrupt PUP is not subject to BSP flow control allocations and jumps over any and all buffered data.

An Interrupt PUP should not be sent until the previous Interrupt PUP has been acknowledged with an *InterruptReply* PUP. Interrupt PUP ID are generated from the stream's send Interrupt ID, which is initially the Connection ID. Successive interrupts advance the Interrupt ID. An Interrupt should be retransmitted until acknowledged.

Upon receipt of an Interrupt PUP, it should be acknowledged with an InterruptReply PUP only if its ID is equal to or one less than the current Interrupt ID. If its ID matches the current Interrupt ID, the using process should be signalled and the Interrupt ID advanced. If its ID is one less than the current Interrupt ID, it is a duplicate and should therefore be acknowledged without giving rise to a new signal to the process.

5.5 FTP Protocol

The complete document for the "PUP File Transfer Protocol Specifications 7th edition" is included in Appendix B. It is a precise, well written paper describing the FTP protocol in great detail. Due to the amount of material covered in that document, only a summary will be given in this section. For full details refer to Appendix B.

There are three major elements defined in the FTP protocol: syntax of a property list, typical property/value pairs, and command sequences. A general description of each is provided here.

5.5.1 Syntax of a File Property List

The notion of a file property list is central to the file transfer protocol. The properties of a file are, in general, contained either explicitly or implicitly within an operating system on a particular machine. An actual property list may contain information provided by the user, from the file system, or from specialized information about the environment. There is no guarantee that a receiving process will be able to properly encode all properties.

The property list might also be used to merely provide information about one or more files (where they are, when they were last written, etc.) In addition, a file property list can provide a partial description for a file which you are trying to find, or which you would like to create.

A file property list consists of a string of ASCII characters, beginning with a left parenthesis and ending with a matching right parenthesis. Within that list, each property is represented similarly as a parenthesized list. For example:

((Server-Filename TESTFILE.7)(Byte-Size 8))

The first item in each property (delimited by a left parenthesis and a space) is the property name, taken from a fixed but extensible set. Upper- and lower-case letters are considered equivalent in the property name. The text between the first space and the right parenthesis is the property value. All characters in the property value are taken literally.

All spaces are significant, and multiple spaces may not be arbitrarily included. There should be no space between the two leading parentheses, for example, and a single space separates a property name from the property value. Other spaces in a property value will become part of that value, so that the following example will work properly:

((Server-Filename xxxxx)(Read-Date 23-Jan-76 11:30:22))

FTP implementations are required to ignore properties whose names they don't know. This enables extensions to be introduced to the set of defined properties without invalidating existing programs. Also, FTP implementations are expected to send as complete a set of properties as is possible in every message.

5.5.2 Typical Properties and Their Values

The FTP property lists are divided into two areas; mandatory properties and optional properties. Listed below is the set of properties, and typical values associated with them, used in the thesis implementation. Not all properties may be required in each property list. The full detail of each property is discussed in Appendix B.

Server-Filename	ftp.c, /etc/passwd, cc
Directory	/usr/mark/ether, /etc, /bin
Name-Body	ftp.c, passwd, cc
Type	Text, Binary
End-of-Line-Convention	CR, CRLF, Transparent
Byte-Size	*decimal number, i.e.,* 7, 8, 36
Size	*decimal number, i.e.,* 14597
Author	bill, dave, admin
Date-Written	31-May-84 11:30:22
User-Name	*user-name, i.e.,* root
User-Password	*password, i.e.,* secret

5.5.3 FTP Commands

In every FTP connection, one party is the User and the other party is the Server: the User initiates commands, while the Server provides replies. In general, there will be some form of reply generated in answer to every command.

Every transaction in the FTP consists of a BSP Mark Byte, followed by a piece of data, and terminated at the next BSP Mark Byte encountered. The data thus enclosed might be one or more file property lists, a text string, or a file. A command might simply be terminated by the arrival of the next Mark Byte command, or by an End-of-Command Mark Byte (essentially a no-op).

For example, a simple retrieval of a file would include:

1. User sends the Server a command requesting the retrieval, with a file property list describing the desired file.
2. Server responds with an appropriate message.
3. If the Server said Yes, then the file is placed into the byte stream.

If a process replies with a No, the first byte of the following data is a numeric code for possible machine processing, and the remaining bytes are a comment for the user. (The correct functioning of the protocol, however, does not depend upon the many possible numeric codes; the simple Yes and No messages are sufficient for using the protocol.)

Once again, particular implementations of the FTP may not support all possible commands. The protocol provides a standard response (the No message with a specific code) by which a server may indicate that a requested command is not implemented. This must be the server's sole response to an unimplemented command; it must not take more drastic action such as breaking the connection.

Listed below are the Mark Bytes used in the thesis implementation. The full detail of each Mark Byte is discussed in Appendix B.

[Retrieve] *property-list*

[Store] *property-list*

[New-Store] *property-list*

[Yes] *code human-readable-string*

[No] *code human-readable-string*

[Here-is-Property-List] *property-list*

[Here-is-File] *file-data*

[Version] *code human-readable-string*

[Comment] *human-readable-string*

[End-of-Command]

[Enumerate] *property-list*

[Delete] *property-list*

5.5.4 Command/Response Sequences

The complete user-server protocols for each command are documented in flowchart form, both here in the summary, Figures 5.9 5.13, and in Appendix B. Commands from user to server are prefixed by "U:"; from server to user by "S:". Commentary is included in *italics*.

These flowcharts describe only the protocols themselves, and do not show any local actions that the two parties must take. Such actions are necessarily file system or application dependent.

Each sequence begins with the FTP user in control, and ends with the user again in control. A party sends messages only when it is in control. Sending [EOC] switches control to the other party. Where a branch occurs in the flowchart, the party then in control may follow *either* branch and send the message shown at the end of that branch. The normal flow of control is straight down from the top (and looping back up at the left in those protocols that contain loops); errors and other exceptions branch off to the right.

At any point in the protocol, the party then in control may insert a [Comment] message immediately preceding its next command; such a message does not alter the flow of control. Apart from Comments, an occurrence of a command-response sequence other than those shown in the flowcharts is a violation of the protocol and is grounds for aborting the connection.

The command at the beginning of every sequence includes a property list designating some file (or potential file) on the server. The file may be described by Server-Filename, Directory, and Name-Body properties in any combination permitted by the server. The property list must include not only filename information but also any credentials required to gain access to the server (User-Name, User-Password, etc.).

5.5.5 Example

The sample output shown below is taken from the thesis project by invoking FTP with the command line "ftp -ftpdebug r68k". The -ftpdebug enables the display of the FTP protocol. This example demonstrates list, retrieve, store, and delete capabilities along with showing some access and error violations.

```
[Connected to: r68k]
U: [Version]
U: <1> 68000/UNOS User FTP 1.00
U: [End-of-Command]
S: [Version]
S: <1> r68k - 68000/UNOS FTP Server 1.00 of Mon May 14 1984 10:10
S: [End-of-Command]
< r68k - 68000/UNOS FTP Server 1.00 of Mon May 14 1984 10:10
FTP-> ? - One of the following:
cd (local directory to)            delete remote file
directory (remote default)         EOL convention
list remote files matching         login as remote user
quit                               retrieve remote file
store local file                   show (progress of transfer)
type (default for transfer)        verbose (mode)
! (shell command)
FTP-> directory (remote default) /tmp
FTP-> list remote files matching a
U: [Directory]
U: ((Server-Filename a)(Directory /tmp)(User-Name mark)(User-Password ))
U: [End-of-Command]
S: [No]
S: <17> Incorrect user-password
S: [End-of-Command]
< Incorrect user-password
FTP-> login as remote user mark password
FTP-> list remote files matching a
U: [Directory]
U: ((Server-Filename a)(Directory /tmp)(User-Name mark)(User-Password secret))
U: [End-of-Command]
S: [Here-is-PList]

 Name                    Author              Write-date       Size     Type
S: ((Server-Filename /tmp/a)(Name-body a)(Directory /tmp)(Author mark)(Size 21\
55)(Write-Date 14-May-84 16:31:49)(Type Text))
 a                       mark           14-May-84 16:31:49    2155     Text
S: [End-of-Command]
FTP-> cd (local directory to) /tmp
FTP-> retrieve remote file a
U: [Retrieve]
U: ((Server-Filename a)(Directory /tmp)(User-Name mark)(User-Password secret))
U: [End-of-Command]
S: [Here-is-PList]
S: ((Server-Filename /tmp/a)(Name-body a)(Directory /tmp)(Author mark)(Size 21\
55)(Write-Date 14-May-84 16:31:49)(Type Text))
S: [End-of-Command]
Text file /tmp/a to local file ar
U: [Yes]
U: <0> File open, ready for data
U: [End-of-Command]
S: [Here-is-File]
!!
S: [Yes]
2155 bytes, 1 seconds, 17240 bits/sec
S: <0> Transfer complete
S: [End-of-Command]
FTP-> store local file a as remote file at
```

```
U: [Directory]
U: ((Server-Filename at)(Directory /tmp)(User-Name mark)(User-Password secret))
U: [End-of-Command]
S: [No]
S: <64> File(s) do not exist.
S: [End-of-Command]
a, Type Text
U: [NewStore]
U: ((Server-Filename at)(Directory /tmp)(User-Name mark)(User-Password secret)\
(Type Text)(End-of-Line-Convention CR))
U: [End-of-Command]
S: [Here-is-PList]
S: ((Server-Filename /tmp/at)(Name-body at)(Directory /tmp)(Type Text))
S: [End-of-Command]
U: [Here-is-File]
!!
S: [Yes]
2155 bytes, 1 seconds, 17240 bits/sec
U: [Yes]
U: <0> Transfer complete
U: [End-of-Command]
S: [Yes]
S: <0> Store complete
S: [End-of-Command]
FTP-> store local file a as remote file at
U: [Directory]
U: ((Server-Filename at)(Directory /tmp)(User-Name mark)(User-Password secret))
U: [End-of-Command]
S: [Here-is-PList]
S: ((Server-Filename /tmp/at)(Name-body at)(Directory /tmp)(Author mark)(Size \
2155)(Write-Date 14-May-84 16:34:07)(Type Text))
S: [End-of-Command]
Remote file at already exists - overwrite? Yes
a, Type Text
U: [NewStore]
U: ((Server-Filename at)(Directory /tmp)(User-Name mark)(User-Password secret)\
(Type Text)(End-of-Line-Convention CR))
U: [End-of-Command]
S: [Here-is-PList]
S: ((Server-Filename /tmp/at)(Name-body at)(Directory /tmp)(Author mark)(Size \
2155)(Write-Date 14-May-84 16:34:07)(Type Text))
S: [End-of-Command]
U: [Here-is-File]
!!
2155 bytes, 1 seconds, 17240 bits/sec
U: [Yes]
U: <0> Transfer complete
U: [End-of-Command]
S: [Yes]
S: <0> Store complete
S: [End-of-Command]
FTP-> delete remote file at
U: [Delete]
U: ((Server-Filename a)(Directory /tmp)(User-Name mark)(User-Password secret))
U: [End-of-Command]
S: [Here-is-PList]
S: ((Server-Filename /tmp/a)(Name-body a)(Directory /tmp)(Author mark)(Size 21\
```

```
55)(Write-Date 14-May-84 16:31:49)(Type Text))
S: [End-of-Command]
Delete /tmp/a of 14-May-84 16:31:49 ? Yes
U: [Yes]
U: <0> Delete that file
U: [End-of-Command]
S: [Yes]
S: <0> File deleted
S: [End-of-Command]
FTP-> list remote files matching logins
U: [Directory]
U: ((Server-Filename logins)(Directory /tmp)(User-Name mark)(User-Password se\
cret))
U: [End-of-Command]
S: [Here-is-PList]

 Name                    Author              Write-date       Size     Type
S: ((Server-Filename /tmp/logins)(Name-body logins)(Directory /tmp)(Author ad\
min)(Size 1408)(Write-Date 14-May-84 15:36:30)(Type Text))
 logins                  admin          14-May-84 15:36:30    1408     Text
S: [End-of-Command]
FTP-> delete remote file login*
U: [Delete]
U: ((Server-Filename login*)(Directory /tmp)(User-Name mark)(User-Password se\
cret))
U: [End-of-Command]
S: [Here-is-PList]
S: ((Server-Filename /tmp/logins)(Name-body logins)(Directory /tmp)(Author ad\
min)(Size 1408)(Write-Date 14-May-84 15:36:30)(Type Text))
S: [End-of-Command]
Delete /tmp/logins of 14-May-84 15:36:30 ? Yes
U: [Yes]
U: <0> Delete that file
U: [End-of-Command]
S: [No]
S: <65> File is protected - access denied
< File is protected   access denied
S: [End-of-Command]
FTP-> quit ... connection closed
```

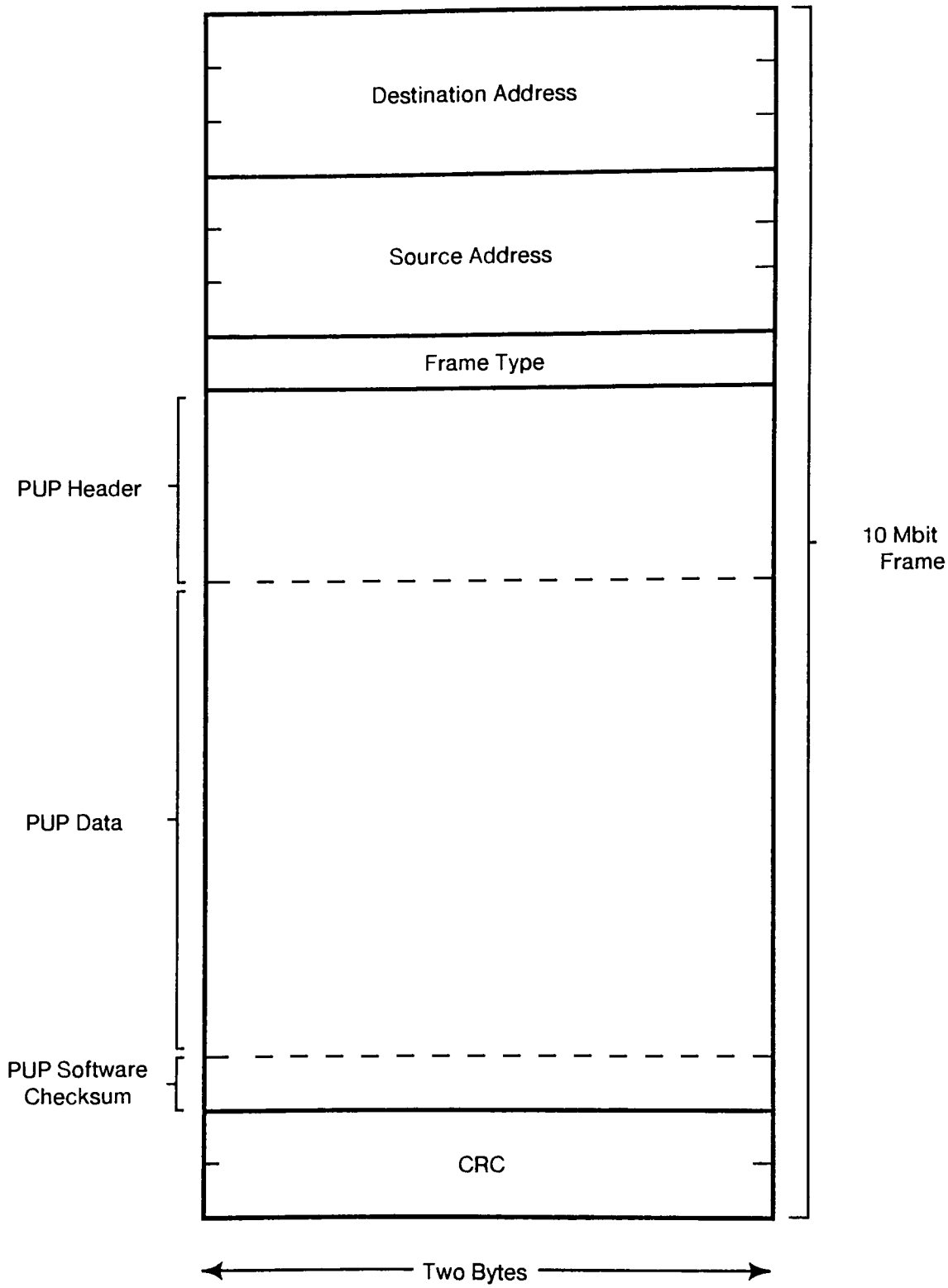


Figure 5.1: Encapsulated PUP Format

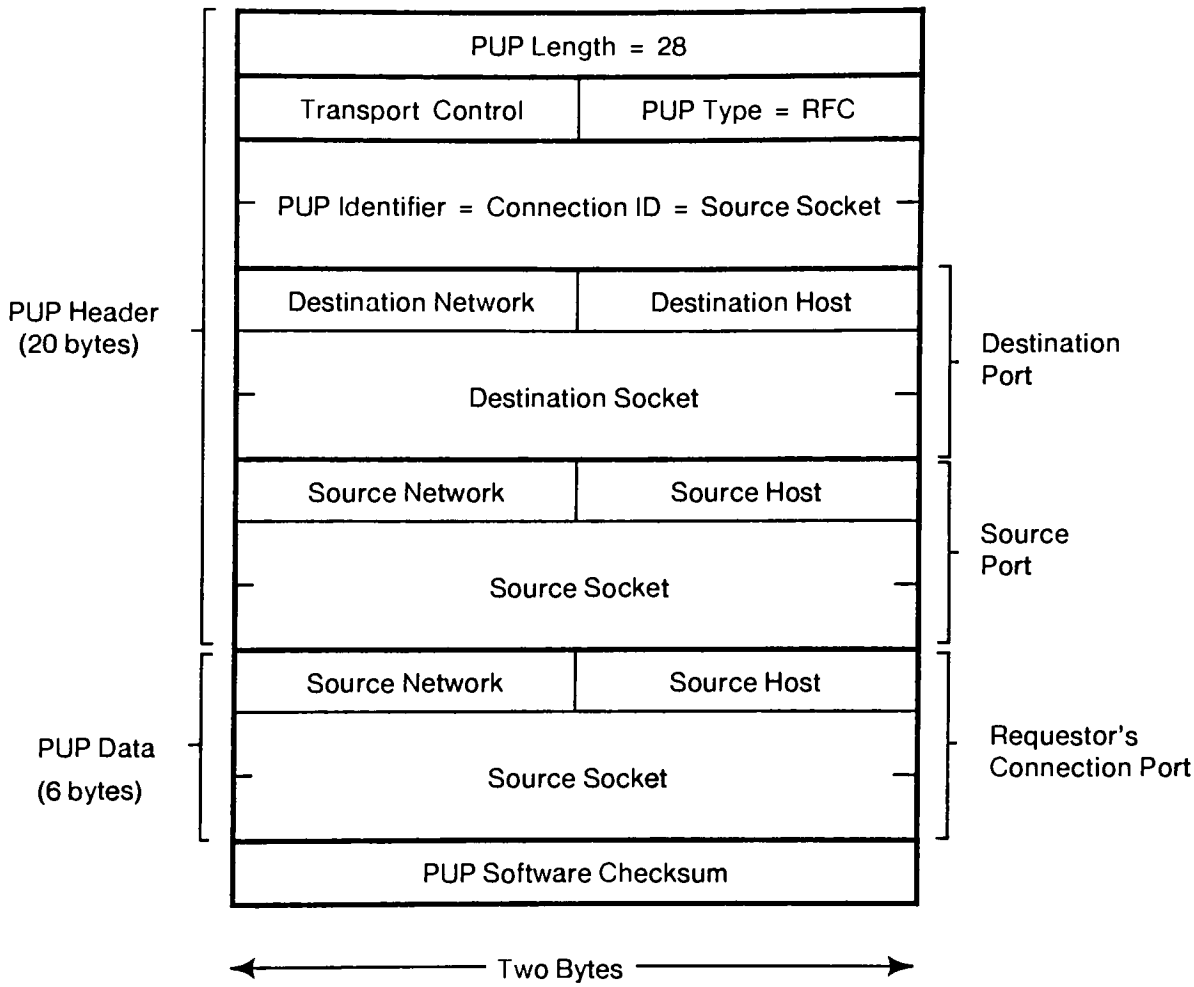


Figure 5.2: Rendezvous PUP Format

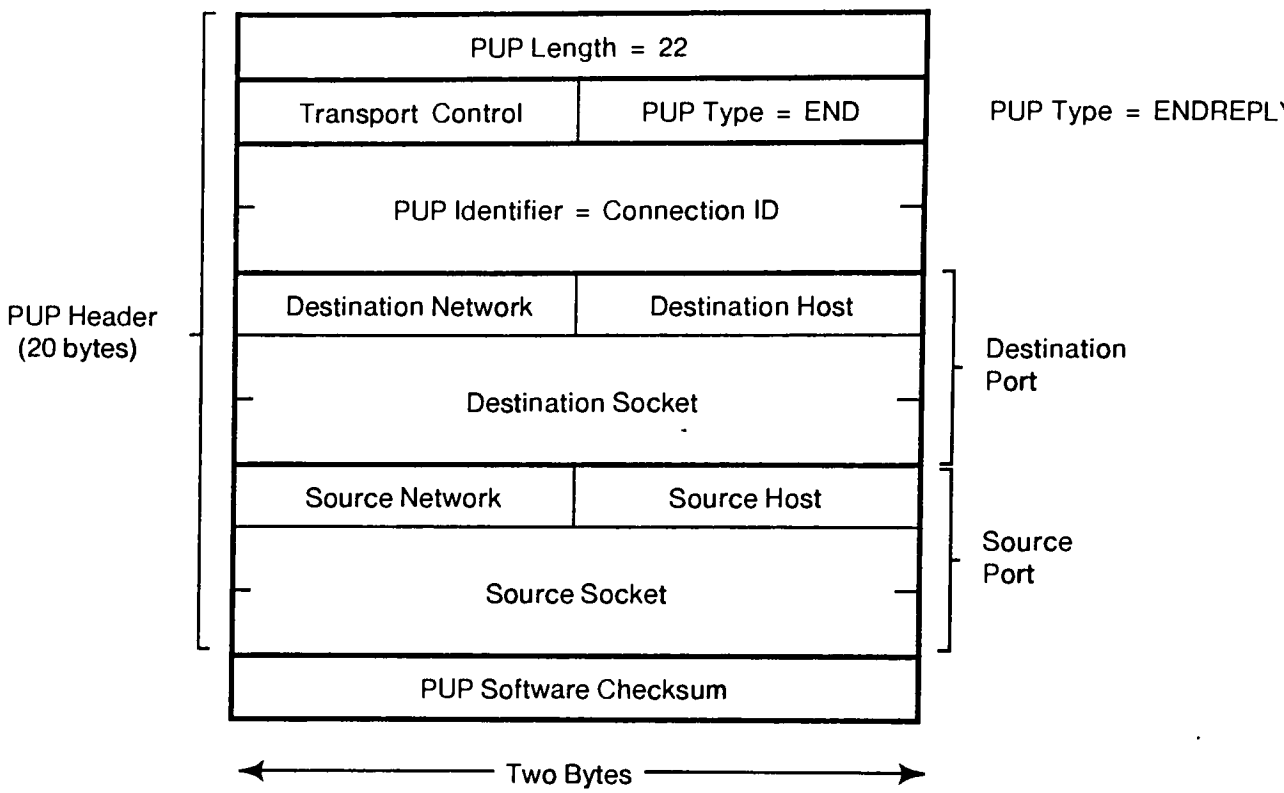


Figure 5.3: Terminate PUP Formats

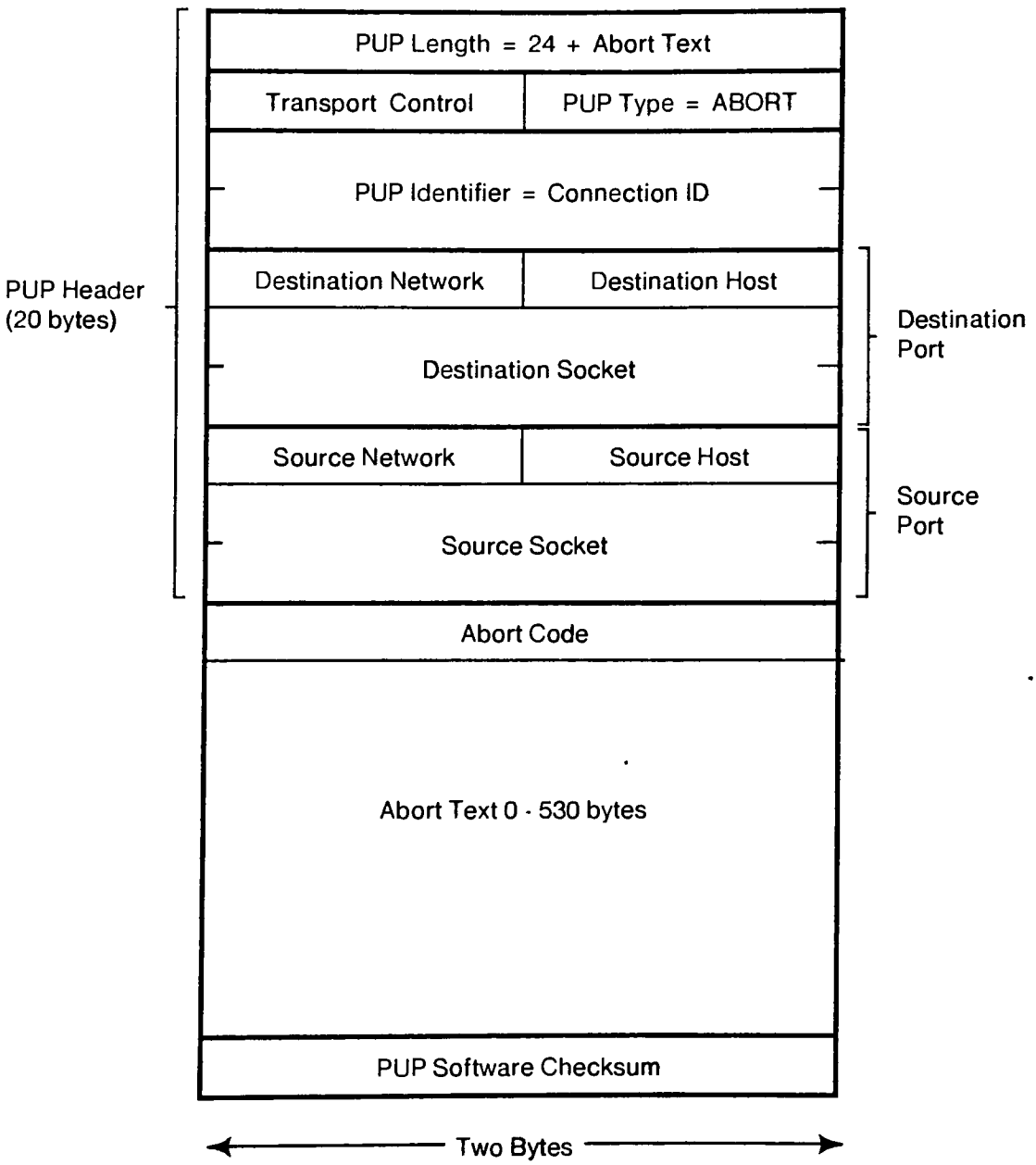


Figure 5.4: Abort PUP Format

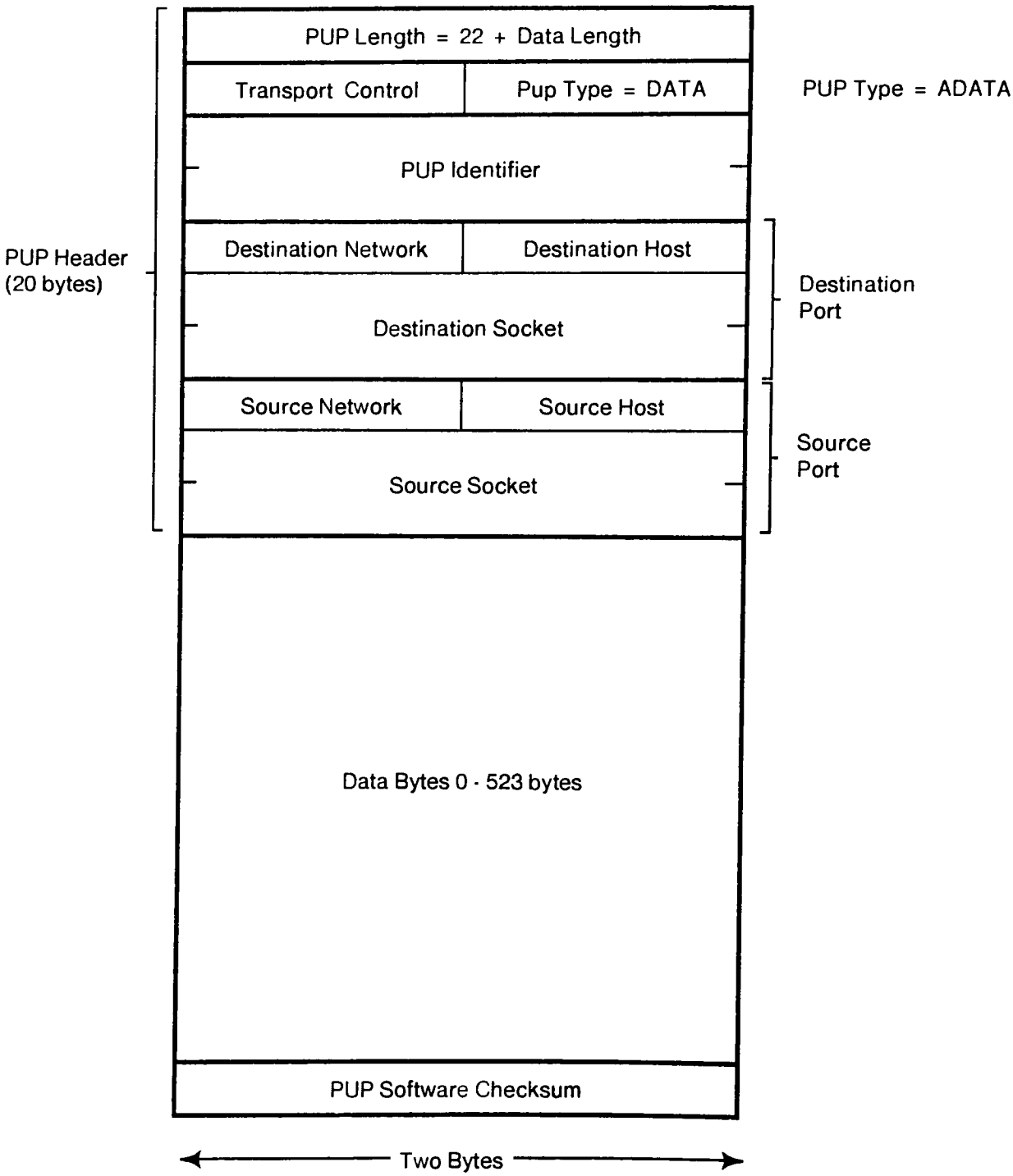


Figure 5.5: Data PUP Formats

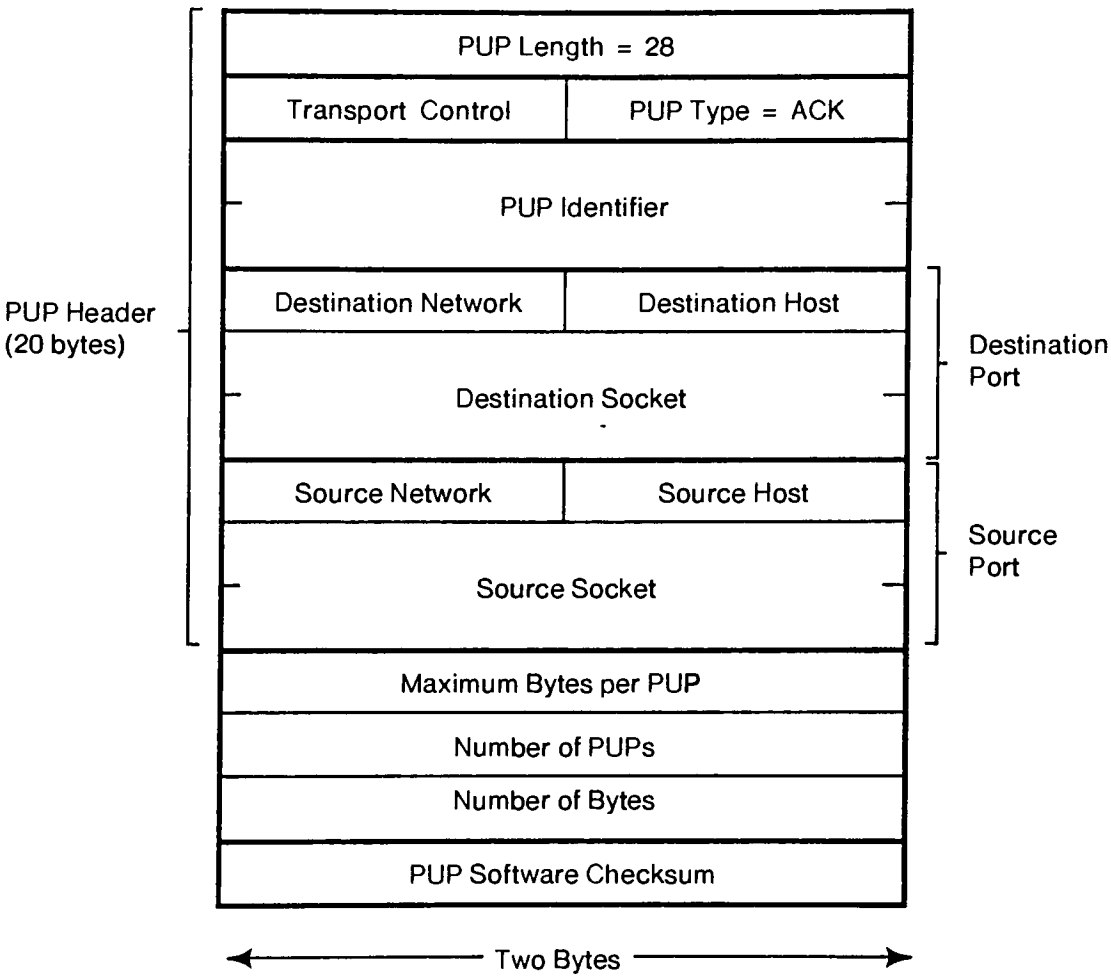


Figure 5.6: Acknowledgment PUP Format

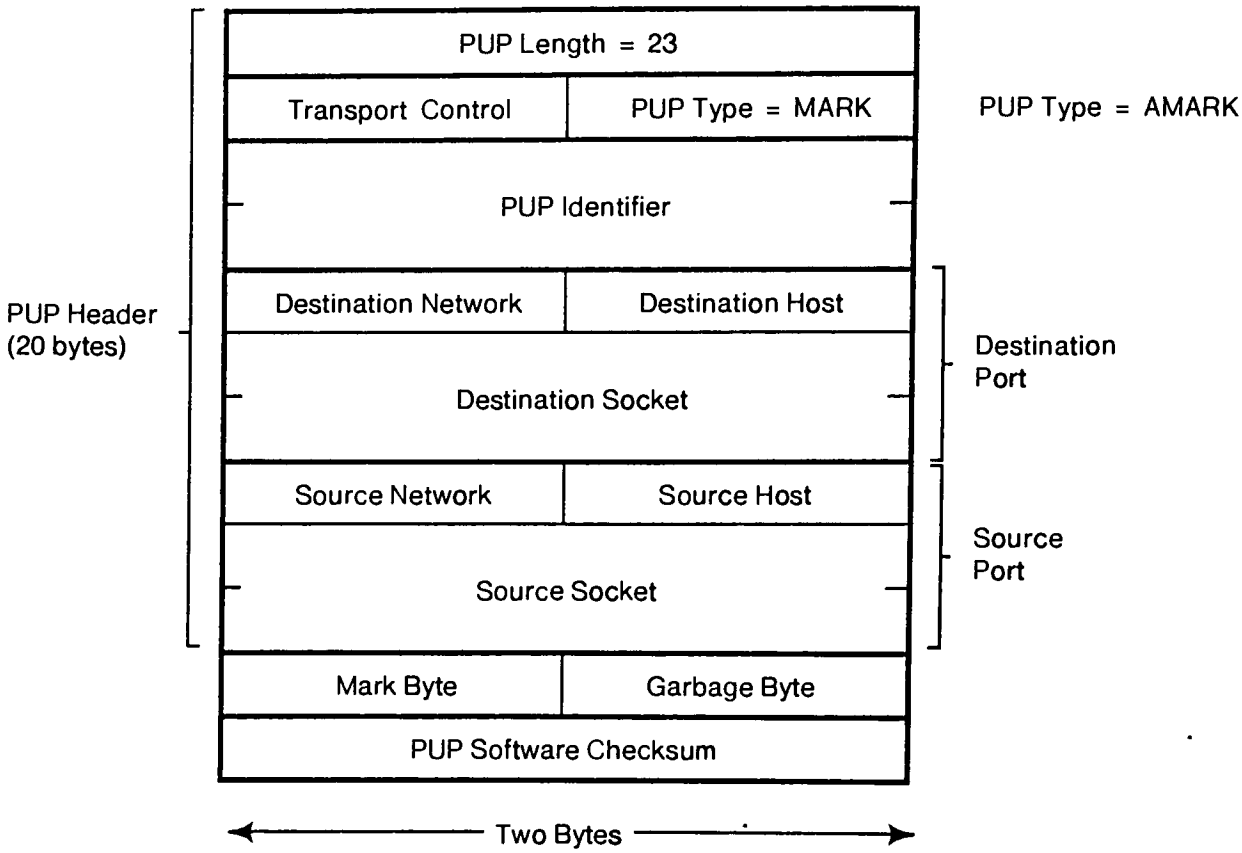


Figure 5.7: Mark PUP Formats

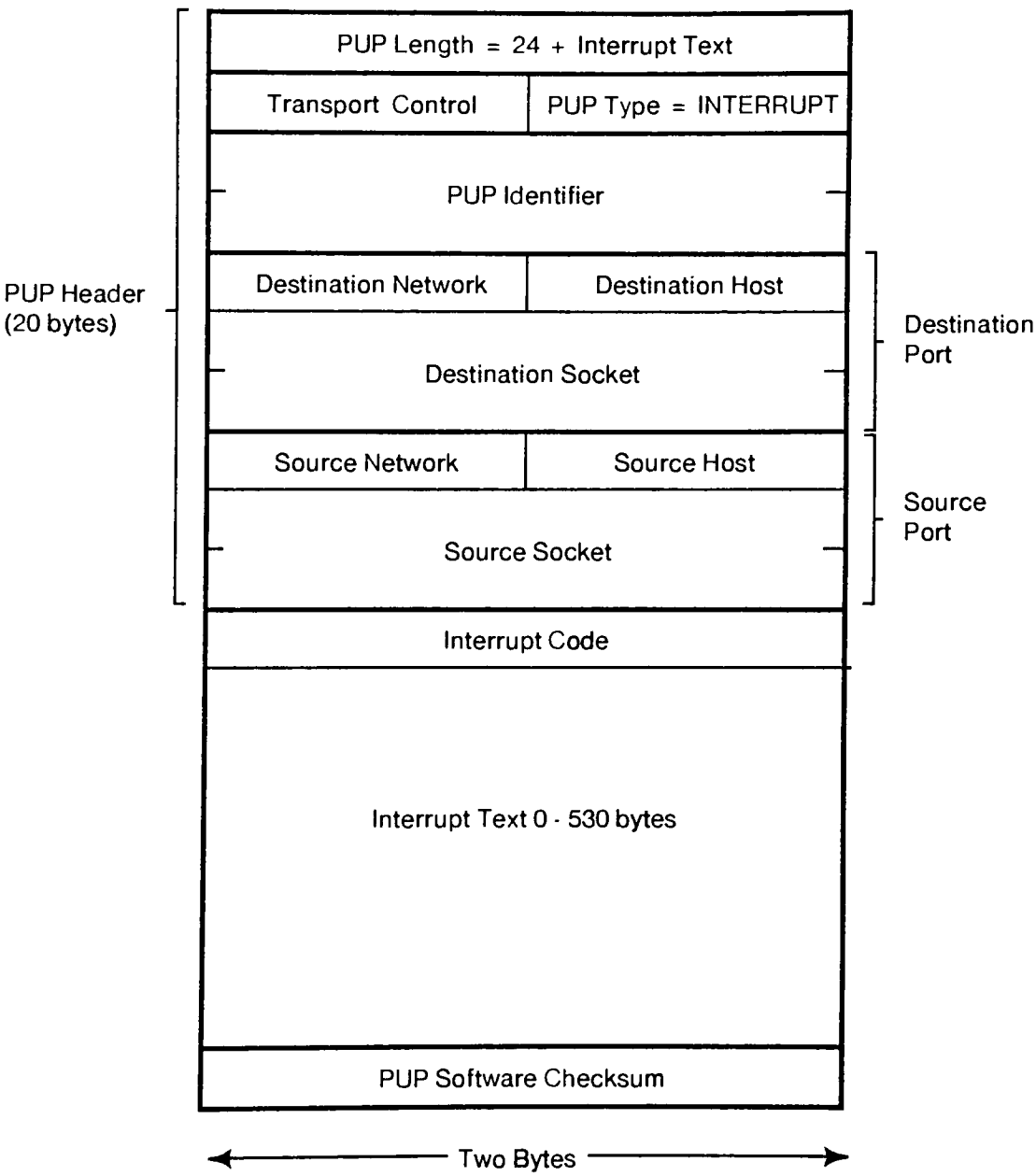
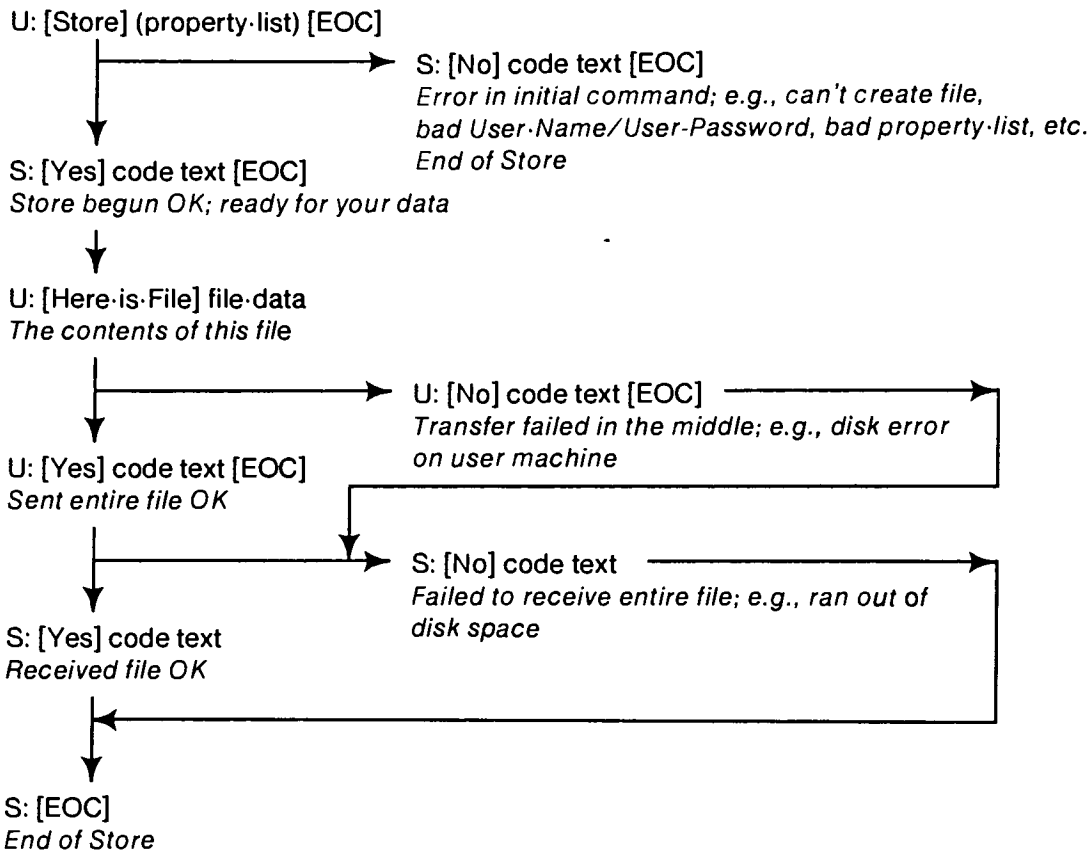
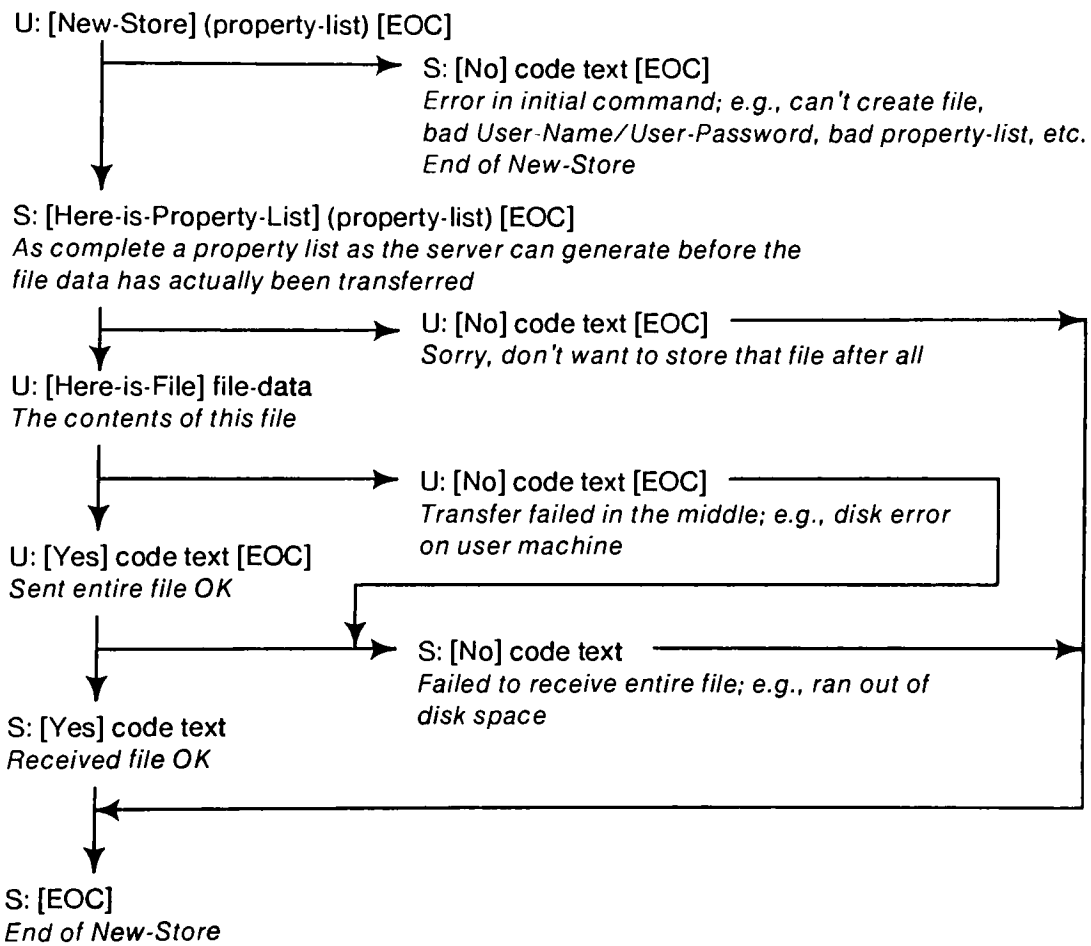


Figure 5.8: Interrupt PUP Format



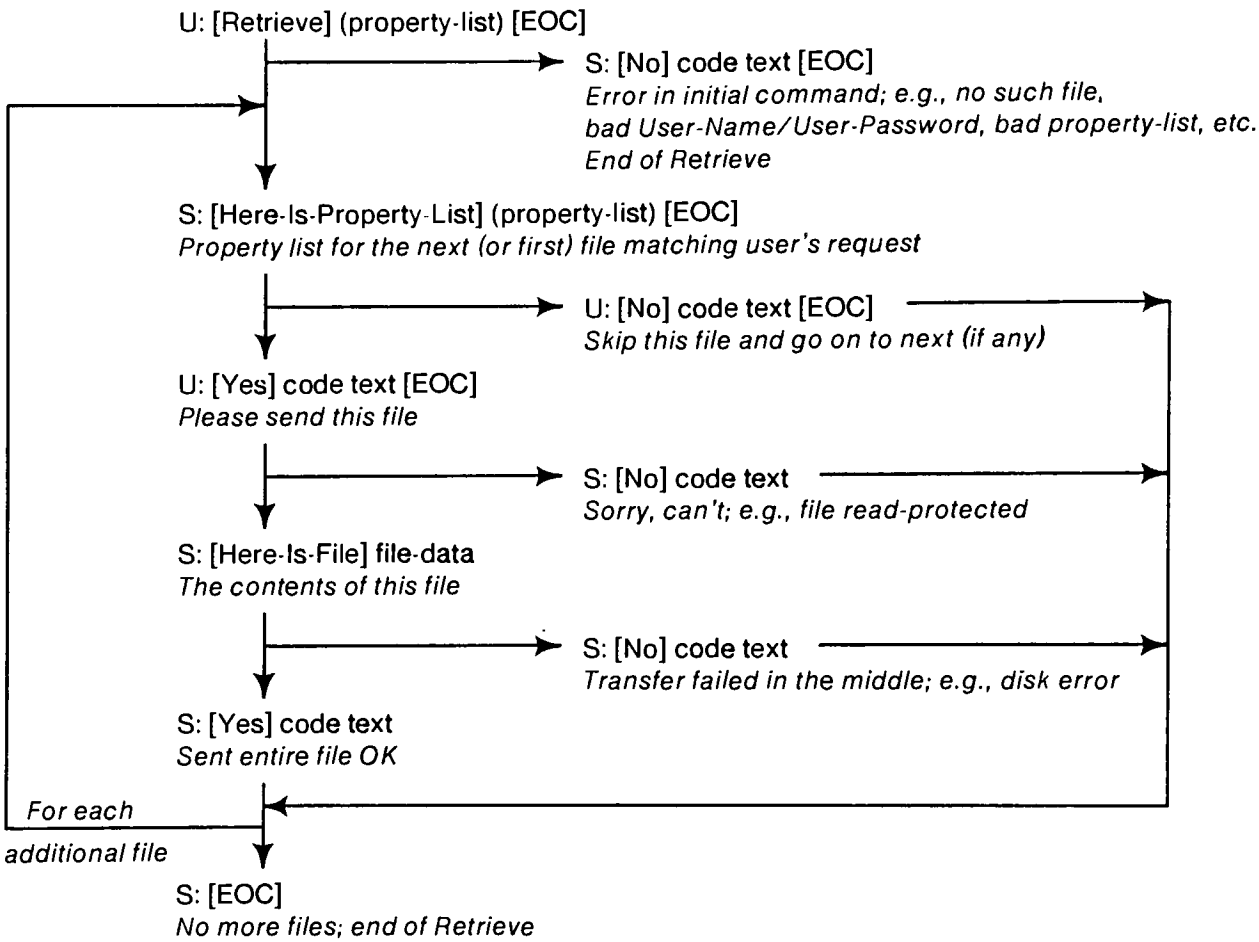
FTP [Store] Protocol

Figure 5.9



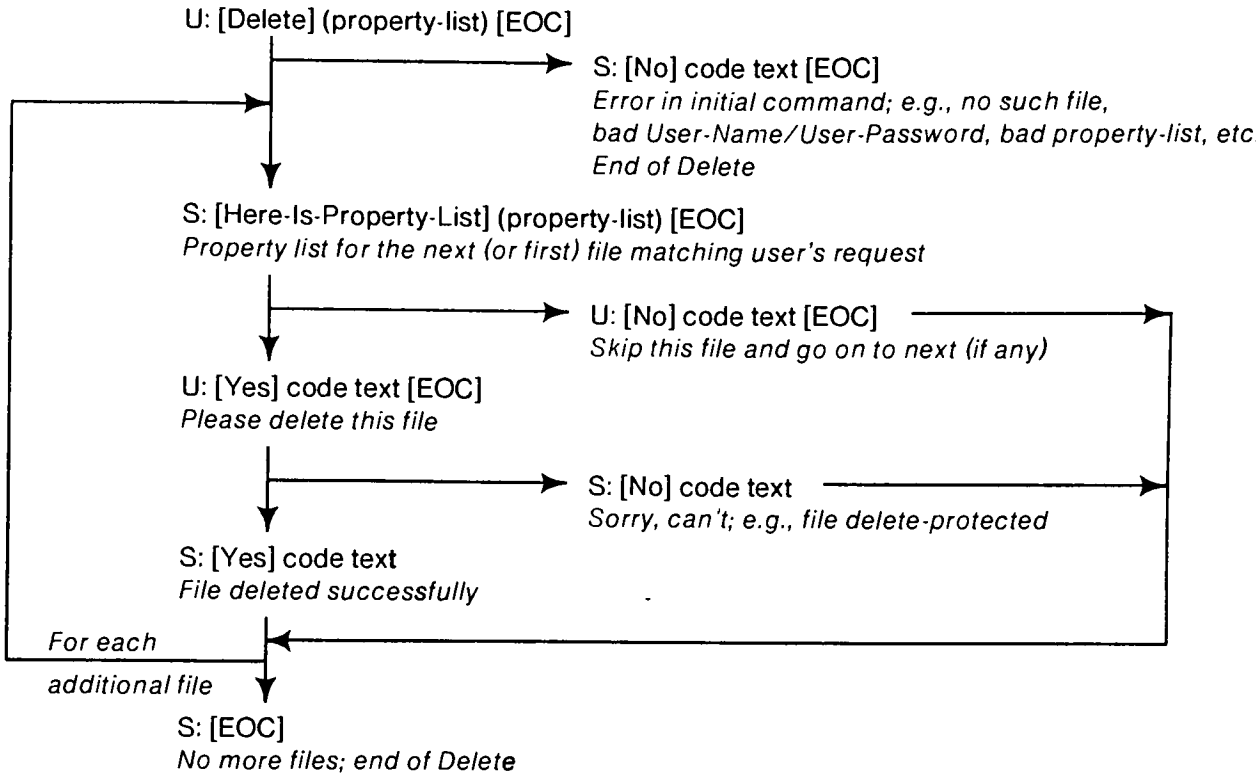
FTP [New-Store] Protocol

Figure 5.10

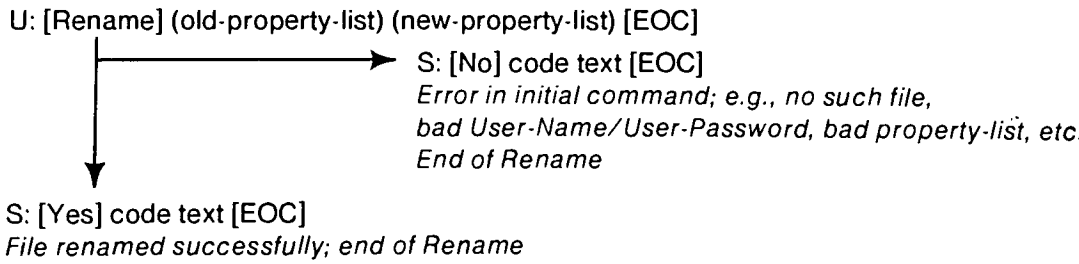


FTP [Retrieve] Protocol

Figure 5.11

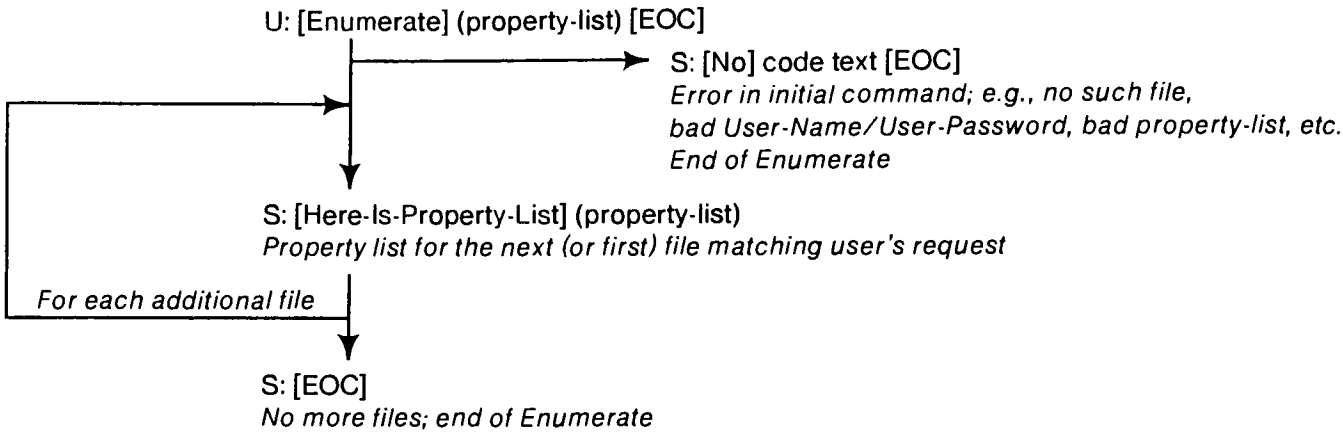


FTP [Delete] Protocol

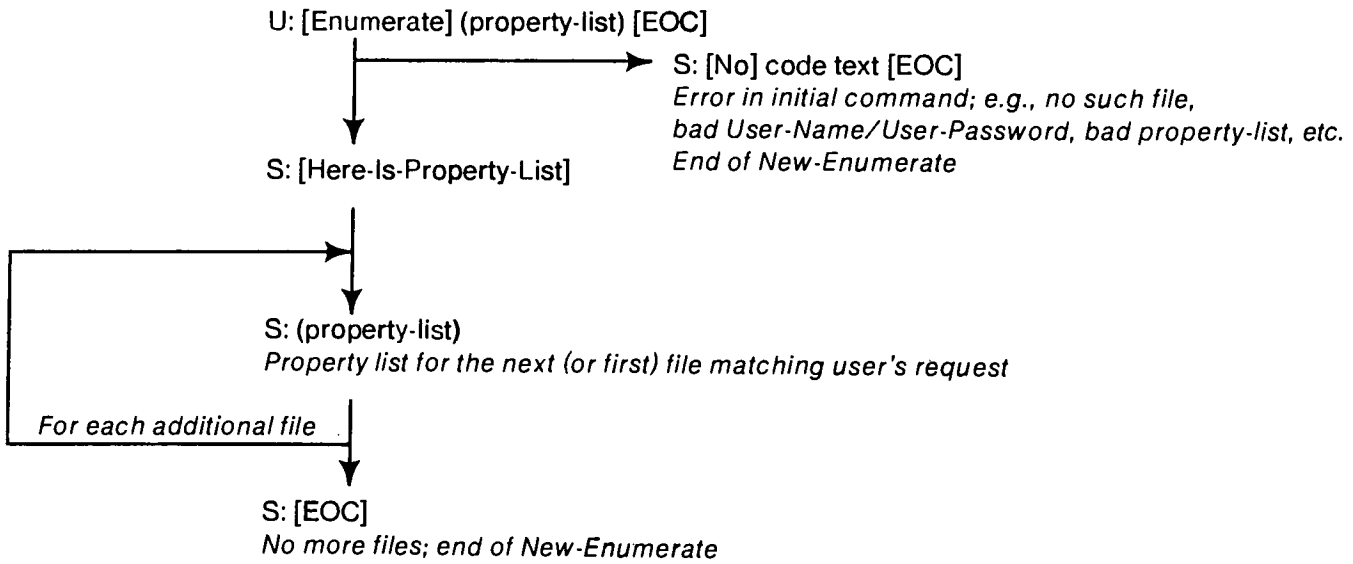


FTP [Rename] Protocol

Figure 5.12



FTP [Enumerate] Protocol



FTP [New-Enumerate] Protocol

Figure 5.13

6. Module Designs

The purpose of this section will be used to describe the written code and data structures used to implement file transfer across a 10 Megabit Ethernet on the CRDS minicomputer. There were four modules of code written for the project; (1) the Ethernet driver, (2) the PUP package, (3) the BSP package, and (4) the FTP package. An overall picture will be used to describe the location and interconnection of each module followed by detailed explanations of each module.

Figure 6.1 shows the relationship between software and hardware for the CRDS implementation. The FTP, BSP and PUP levels reside in the user space portion of the memory map. The PUP level does a file write (or read) which will trap into the kernel space of the memory map. At this point, the Ethernet driver gets control allowing transfer of data to the Ethernet hardware. Upon completion of the operation, the status is returned to the user space file system and eventually back to the PUP level.

6.1 Ethernet Driver

6.1.1 Introduction to UNOS Drivers

The UNOS operating system communicates with devices via special routines called device drivers. Device access is divided into two classes; buffered and unbuffered. Buffered I/O passes data between a system cache and the device, in a block size meaningful to that device (typically 512 bytes). Once in the system cache, a smaller number of bytes may be passed to or from the user's routines, as their requests warrant. Hence in buffered I/O, the burden of blocking is handled by the system. Conversely, in unbuffered I/O, the user process must handle the blocking. Unbuffered I/O moves data between the user process and the device, avoiding the system cache.

UNOS uses a single device switch table to access all its drivers. This table has an entry for each driver on the system. The order of the entries in the device switch table determines the device driver's major number. Devices are divided into major and minor device numbers. The major number accesses the particular device driver and the minor number is passed to the device driver for use as the routine sees fit. The first entry in the device switch table has major number 0, while the second has major number 1, etc. Each entry in the device switch table points to an individual driver's dispatch table. Each dispatch table has eight entry points that are accessible to the rest of the kernel. Each entry point corresponds to a specific kernel action or request; i.e., initializing, reading, or writing. Note that a driver need not support all functions but must fill the dispatch table with appropriate dummy routines for those not supported.

When a process makes an I/O request, the kernel uses the major device number it finds in the file descriptor as an index into the device switch table in order to select the correct device driver. Based on the I/O request, the kernel finds the specific function to perform from the driver's dispatch table. The eight specific functions a device driver may have are:

initialization	Only called when UNOS is first booted.
queue entry	Queues a request for a buffered device.
read	Performs an unbuffered read.
write	Performs an unbuffered write.
special function	Allows commands to be given to the device that were not covered in the other entry points.
open	Ready device for access.

close	Closes device.
map eventcount	Maps driver eventcount address for use in user space.

6.1.2 Ethernet Driver Specifics

The UNOS Ethernet device driver operates in unbuffered access mode and supports eight minor devices. The entry point to the minor devices are located in the "/dev" device directory of UNOS. The eight entries have the same major device number, seventeen, and consecutive minor device numbers starting at zero. The entries in "/dev" are:

Major	Minor	Device Name
17	0	/dev/enet0
17	1	/dev/enet1
17	2	/dev/enet2
17	3	/dev/enet3
17	4	/dev/enet4
17	5	/dev/enet5
17	6	/dev/enet6
17	7	/dev/enet7

The Ethernet driver supports seven of the eight functions in the dispatch table. Because the driver operates in unbuffered access mode, there is no need for the "queue entry" routine. The software is broken into the seven modules for the dispatch table plus the "interrupt handler" and the "hardware start" routines. The functions performed by each module are described below.

6.1.2.1 Initialization

The initialization routine, "enetinit", is called once when the UNOS operating system is booted. It must check if the Ethernet board exists and mark the software accordingly. Existence is determined by reading or writing a hardware register on the Ethernet board. A positive response from that address indicates that the Ethernet hardware exists. A negative response would be flagged in the software and any other access to the Ethernet driver would return an error.

After insuring that the hardware is available, the software is initialized. The Ethernet interrupt vector is setup to point to the Ethernet interrupt handler, "enetintr". Next the eight minor devices are shown closed and their respective receive queue empty. Last, the write and read data is setup. The write buffer structure is initiated to be available. The memory is obtained from the system for the receive queue buffers and placed on the free buffer list.

Finally, the Ethernet board is issued a reset to insure a known state, put on line to logically connect it to the Ethernet cable, and enabled to interrupt the processor upon receiving a frame.

6.1.2.2 Read

The Ethernet unbuffered read routine, "enetread", need only copy a previously received frame from kernel space to user space. Before the frame is copied, the Ethernet read routine checks that a valid minor device has been specified and examines that minor's receive queue for pending frames. If the queue is not empty, the next pending frame is dequeued and copied to the user's buffer. The empty frame is returned to the free buffer list and the minor's "FrameOUT" eventcount is advanced. The number of bytes transferred to the user's buffer is returned to the calling procedure. The "FrameOUT" eventcount will be discussed at a later time.

6.1.2.3 Write

The Ethernet unbuffered write routine, "enetwrite", transfers data from the user supplied buffer to the Ethernet board. Before data is transferred, the Ethernet write routine checks that a valid minor device has been specified. Next, the write buffer structure is acquired through mutual exclusion and filled with the necessary information. The structure is passed to the "enetstart" routine to perform the actual transfer. At this point the process becomes blocked until the I/O completes. Once active again, the status of the completed I/O is checked from data placed in the write buffer structure by the interrupt handler. Finally the write buffer structure is released and the result of the I/O returned.

6.1.2.4 Special Function

The special function entry point to the Ethernet driver, "enetspfun", allows valid minor devices to give additional commands to the ethernet driver, i.e. run diagnostics, dump statistics, reset. The Ethernet driver status can also be obtained and examined for debug purposes.

6.1.2.5 Open

The Ethernet open routine is responsible for allowing or denying access to the Ethernet hardware. Before any transfers can be performed, a minor device must be opened through the "enetopen" routine. Only valid, non-busy, minor devices allow access to the Ethernet hardware. Each minor device has an associated data structure which is initialized at this time and marked with a device open flag.

6.1.2.6 Close

The "enetclose" routine closes a valid minor device by marking the associated data structure with a device closed flag. Any pending frames in the receiver's queue are dequeued and placed back on the free buffer list.

6.1.2.7 Map Eventcount

To allow access to the "FrameIN" and "FrameOUT" eventcounts, the Ethernet driver supplies the map eventcount routine "enetmapec". This will return the address of the specified eventcount for a valid minor device. The eventcount and their usage will be explained at a later time.

6.1.2.8 Interrupt Handler

The Ethernet interrupt handler, "enetintr", is responsible processing the four interrupts generated by the Interlan Ethernet Controller card. The four interrupts are briefly discussed here.

(1) The *receive frame available* indicates that the Ethernet hardware has received a frame with this hardware's destination address. The interrupt routine will start a DMA transfer to obtain the frame.

(2) The *DMA receive done* indicates that the DMA transfer to obtain a frame from the Ethernet hardware has completed. The frame is queued for the appropriate socket and the "FrameIN" eventcount advanced. Any pending Ethernet I/O requests are now started. The "FrameIN" eventcount will be discussed at a later time.

(3) The *DMA transmit done* indicates that the DMA transfer to send data to the Ethernet hardware has completed. The command to transmit this frame onto the Ethernet is now given to the Ethernet hardware.

(4) The *status register full* indicates that the previous Ethernet command has completed and the status results are available. Any pending Ethernet I/O requests are now started.

6.1.2.9 Hardware Start

This routine, "enetstart", checks if a new Ethernet transfer can be started. If the Ethernet hardware is busy with another I/O request, the routine does nothing. It then examines the hardware to see if a frame has been received or checks the software to see if the write buffer structure has been filled. At this point it either starts the appropriate I/O request or, if no requests are pending, enables the board to interrupt upon receiving a frame.

6.1.3 Driver Usage

While the description above explains the actions performed by each routine, it does not indicate the path taken by a received or transmitted frame. Also, eventcounts were mentioned above however their usage was not described. This information is outlined below.

6.1.3.1 Data Transfer for Write

The data and control flow for the write process in the Ethernet driver are shown in Figure 6.2. The first step involved in writing, be it data from "enetwrite" or a command from "enetspfun", is acquisition of the one write buffer structure. The kernel supplies a mutual exclusion routine for this function since it is crucial that only one process own the write buffer structure. The necessary data is placed in the write buffer structure and "enetstart" called to start the DMA process. Upon return from "enetstart", the process will become blocked while waiting for its I/O request to complete.

While blocked the Ethernet hardware will interrupt twice. The first time to indicate the DMA has completed; the second time to indicate the frame has been transmitted. The status from the I/O operation is placed in the write buffer structure and the kernel signalled that this I/O operation is complete. The kernel will then schedule the blocked process to run at the process's next deserving timeslice.

When the process wakes, it will get the result of the I/O from the write buffer structure and then release the write buffer structure for others to use.

6.1.3.2 Data Transfer for Read

The data and control flow for the read process in the Ethernet driver are shown in Figure 6.3. Unlike the write process, the read process does not get blocked. Instead, frames from the Ethernet board are preprocessed and queued for the "enetread" routine by the interrupt routine.

In order to read, there must first be frames waiting in the queue associated with the minor device desiring a frame. This happens through two interrupts. The first interrupt happens when the Ethernet board receives a frame. This causes the interrupt handler to obtain an empty receive buffer from the free buffer list and initiate a DMA transfer.

Upon completion of the DMA, the second interrupt occurs. The frame is now filtered to determine its rightful owner (or minor device). Filters are setup through a call to the special function routine, "enetspfun". If the owner's queue is not full, the frame is queued for read. Any frame not used, due to no owner or owner's queue full, is returned to the free buffer list. If a frame is queued, the "FrameIN" eventcount is advanced.

The "FrameIN" eventcount can be used to notify a program of the arrival of a frame destined for that process's socket. UNOS also allows a program to wait in a 'logical or' fashion on multiple eventcounts. Whichever eventcount reaches the desired value first will cause the 'or' to be satisfied and execution to continue. Waiting on the "FrameIN" eventcount and on the (kernel supplied) clock eventcount allows detection of a timeout while waiting for data to be received. If the "FrameIN" eventcount is incremented, by queuing a received frame, data is available and a timeout did not occur. If the clock eventcount reached its desired value first, data did not arrive and a timeout has occurred.

Now that it is apparent that a process can be notified of waiting data in the queue through the use of eventcounts, it must be able to obtain that data. This is done with the read routine, "enetread". The read routine need only dequeue the next frame and copy the data to the user's program buffer. Once done the buffer is returned to the free buffer list and the "FrameOUT" eventcount advanced. The "FrameIN" and "FrameOUT" eventcounts are initialized to the same value. Hence, the difference of the two values shows the number of pending frames in the minor device's queue.

6.2 PUP Level

The PUP software package, implemented for this project, performs three basic functions: (1) open/close the files necessary for access to the driver, (2) handle PUP format for read/write, and (3) perform necessary routing. The PUP package also contains some miscellaneous support routines.

6.2.1 PUP Open and Close

Before any communications may be performed, a minor device must be opened to access the driver. Since the Ethernet device names have a sequential number as a suffix, the PUP open routine executes a linear search to find the first free minor device. The algorithm is show below:

1. Open the device "/dev/enet0". It returns success, device busy, or file does not exist.
2. If success, then a minor device has been obtained and algorithm is completed.
3. If device busy, increment suffix number and try again.
4. If file does not exist, then all minor devices are busy, no minor device is available at this time, and algorithm is completed. This is the error termination point of the algorithm.

Notice the algorithm does not know the number of minor devices supported by the driver. This allows the number of supported minor devices to be increased and not necessitate recompiling of existing code. The PUP open routine simply continues incrementing the suffix and trying to open that device until a "file does not exist" error message is returned. At that point, open has tried all the Ethernet devices and found all the devices to be busy.

After a minor device has been opened successfully, the PUP open routine opens the two Ethernet device related eventcounts, "FrameIN" and "FrameOUT", plus the system supplied clock eventcount. At this point, the necessary defaults are chosen for the local net, local host, and read timeout.

Upon completion of Ethernet I/O, the PUP close routine will be called to close the previously opened device and eventcounts. This will free the Ethernet minor device for use by another process.

6.2.2 PUP Read and Write

The PUP read and write routines perform the necessary packetizing and depacketizing of data to and from the PUP packet format. The read routine must recompute the checksum and verify that it matches the incoming packet's checksum. The write routine must compute the packet checksum for the outgoing PUP packet.

To perform a read, the PUP read routine is supplied with a timeout value. A value of zero is defined to mean wait forever for a packet or essentially no timeout. With the use of the "FrameIN" eventcount and the system device eventcount, the PUP read routine can perform a timed Ethernet read. If a packet does not arrive within the specified period of time, a timeout will be declared and the PUP read aborted. The usage of the eventcounts was described earlier in the device driver.

After receipt of a packet and verification of the checksum, the requested data is returned to the calling routine. Returned data can consist of any part of the PUP header or the contents of the packet. The PUP format was described earlier in Section 5.1.

The PUP write procedure places the necessary data in the PUP packet and writes the information to the file system. The operations to perform a PUP write are extremely simple.

6.2.3 PUP Routing

The PUP routing for this project was minimized due to the isolated Ethernet that connected the hosts. The first hop address would always be on the local Ethernet and not that of a gateway, since no gateway exists. However, mapping was necessary to convert between eight bit 3 Mbit Ethernet address and forty-eight bit 10 Mbit Ethernet addresses. The mapping function was simplified by placing the Interlan assigned 10 Mbit address, a unique 3 Mbit address, and a symbolic name identifying the host in a file called "/etc/enetmappings". When a 3 Mbit address is specified, the translation to the corresponding 10 Mbit address is performed by using the "/etc/enetmappings" file. The symbolic name is used to access a host by that name. The name lookup of the symbolic name and conversion to the 3 Mbit address is performed in the BSP level. Name lookup is not part of the routing function.

Creating a file for the address mapping forces every host to need that file in order to establish any connections. Updating for the addition of new hosts or changing of addresses becomes extremely painful. Forgetting to update "/etc/enetmappings" on a particular host will not allow that host to establish the proper connections. Address mapping with a file is recognized as a disadvantage of the implementation.

6.2.4 Support Routines

The PUP package provides some miscellaneous support routines for the higher level protocols and for debug purposes. Upper level support routines include setting the Ethernet driver receiver's filter, the creation of unique sockets, and the checking of packets pending in Ethernet receiver's queue. Debug support includes routines to print PUP packets and print PUP ports.

6.3 BSP Level

The BSP package was the most complex code written for the thesis. The code performs packet sequencing, packet duplication suppression, retransmission, flow control and timeout detection. The buffer size chosen for this implementation was five. This indicates that there can be up to five unacknowledged packets and that the reader can buffer five packets. The writer must not overflow the reader's packet buffer size.

The BSP package was separated into nine modules, each module performing a specific function. Some modules have sections of code that will only get executed if the process is a server task and conversely a user task. The modules are: (1) open, (2) close, (3) name lookup, (4) abort, (5) read, (6) write, (7), acknowledgement, (8) watchdog timer, and (9) queue package. Each module is discussed below, referencing appropriate drawings where necessary.

6.3.1 BSP Open

The BSP open routine is responsible for creating a unique socket used to establish a connection with another socket. This socket can reside on the same host or another host, miles and gateways apart. The open routine has three modes of operation in which it can be called. Two modes are necessary for the server task and the remaining mode is necessary for the user task.

A user task uses the *initiate* mode to establish a connection, through an exchange of RFC's, with a server task. RFC packets were discussed and shown in Section 5.3.1. The initiate mode of the BSP open routine calls PUP open to get an Ethernet minor device, sets up a local port with a unique socket, and establishes the PUP read timeout and driver receive filter. At this point, there is a fully initialized BSP channel ready for use. The channel will be used to attempt a connection with the destination port on the remote host. The destination port has been previously setup by the BSP name lookup routine. Finally, the matching RFC should be received from the remote host after which the remaining user task software initialization for the BSP package is completed. The software initialization includes setup of the read, write and free queue buffers, clearing all counts, setting the read and write IDs to that of the connection ID, indicating that the writer is blocked and starting the BSP watchdog timer. The function of the timer will be shown later.

The server task, started prior to the user task, uses the *listen* mode and the *server* mode of the BSP open routine. The listen modes sets up a BSP channel with a port containing a well known socket. This socket will be used as the destination socket of the user's RFC packet. While in the listen mode, the BSP open routine will "sit and wait forever · no timeout" until a packet is received for the process. Once verified that the packet does not contain a duplicate connection ID, the BSP open routine returns to the calling procedure.

The BSP open routine is then recalled with the *server* mode. This mode will open its own BSP channel and send the matching RFC back to the user's task. Software initialization, similar to that described above under the *initiate* mode, is then performed to finish setup of the BSP package for the server host.

6.3.2 BSP Close

Like the BSP open, the BSP close has three modes by which it can be called. The first mode is used to start the normal termination protocol, the second mode to reply back to the normal termination protocol, and the last mode to close BSP channels after an abnormal termination from an abort packet. The first mode is usually performed by the user task and the second mode by the server task. Either task may use the abnormal termination mode.

The first mode is used to start the three·way termination handshake for closing two communication ports. Before initiating the normal RTP termination protocol, BSP close processes any remaining data left in the read buffers. Close especially looks for an RTP packet or abort packet from the other host. If neither of these are found, BSP close will try three times to write an RTP end and read an RTP end reply. If three attempts have been made to send an end or the end reply is received, the corresponding end reply is sent and the BSP channel closed.

The second mode in the BSP close routine is used by a task receiving an RTP end packet. This mode will send the corresponding RTP end reply and dally in order to respond to a retransmitted RTP end, should the initial end reply be lost. After timing out or receiving the third packet, an end reply PUP, of the three·way termination handshake, the BSP channel will be closed.

The final mode available in the BSP close routine is used after an abort packet has been received or sent. It simply closes the BSP channel.

6.3.3 Name Lookup

The BSP lookup routine converts a symbolic remote host name or a 3 Mbit remote host address to a port. As mentioned previously, a port contains a net, host, and socket number. Symbolic names are mapped to the corresponding address through the use of "/etc/enetmappings". The name is located and the related address extracted. The "/etc/enetmappings" file was mentioned previously under PUP routing. An unknown symbolic name or unknown host address will cause a message to be printed indicating the error.

6.3.4 BSP Abort

The function of the BSP abort routine is to format the supplied abort code and abort text into an abort packet. The abort packet format was discussed in Section 5.3.3. The packet is then transmitted to the other host after which BSP close is called to close the BSP channel.

6.3.5 BSP Read

The function of BSP read is to copy data from the BSP buffers to the user's buffer and return the PUP packet type just copied into the buffer. This function is not as trivial as it first may appear. The read routine must get the data, handle different packet types, detect timeouts and attempt to keep the Ethernet communications flowing as smoothly as possible.

In this implementation, the BSP read routine has the ability to buffer five PUP packets. While excepting data from the Ethernet, the greatest throughput can be obtained by attempting to fill the five BSP buffers. When coping the BSP buffers to the user buffer, the greatest throughput can be obtained by attempting to fill the user's buffer, thereby emptying the BSP buffers to be filled during the next Ethernet read. To explain the implementation chosen for this project, the paragraphs below will refer to Figure 6.4.

Figure 6.4 shows that the read function is split into two routines, "BSPread" and "BSPreceive". This allows the watchdog timer and the write routine the ability to receive data with "BSPreceive", a function that they must perform. The use of "BSPreceive" will become apparent during the explanation of the watchdog time and the write routines.

When an upper level routine calls BSP read, the BSP read routine examines the BSP buffers. If no data is available, it does a timeout read because data is needed now. Once data begins to flow, it will arrive in groups of packets known as burst mode transmission. Therefore, to improve efficiency, the read routine will accept all pending packets or until the BSP buffers are full. Data for the BSP buffers is read from the PUP level, processed, and placed in the BSP buffers by the "BSPreceive" routine. Incoming packets may also require acknowledgment. This is also performed by "BSPreceive" with a call to the "BSPwtACK" routine. Acknowledgement processing is described later.

After all data is read by "BSPreceive", it must be copied back to the user's buffer. PUP Data and AData packets may be returned at the same time, while Mark, AMark, RTP, and Abort PUPs must be returned one at a time.

The last step in the BSP read routine is to send a gratuity acknowledgment packet only if the last allocation packet sent contained zero and the act of copying data from the BSP buffers to the user buffer made available a BSP buffer. A zero allocation will block the writer and will cause a delay before the writer attempts to get a new allocation. The sending of a

gratuity acknowledgement packet is not part of the BSP protocol; it is just an attempt to avoid unnecessary delays.

In summary, whenever BSP read is called, it will pick up as many pending packets as possible through the use of "BSPreceive". This strategy of picking up pending packets is also seen in the watchdog timer and the write routine.

6.3.6 BSP Write

The BSP write routine sends data from the local host to the remote host. While sending, the write routine must obey the allocation restrictions imposed by the remote host. The allocation refers to the PUP packet size and the number of PUP packets that the reader on the remote host is capable of buffering. Figure 6.5 shows the layout of the BSP write routine. The description below discusses the BSP write code and the diagram.

After the initial connection is established, the writer does not know the reader's buffering capability. Therefore, the writer is considered to be blocked for write. When the writer is blocked, it will use "BSPreceive" to process pending packets while waiting for an allocation packet from the reader. Also, when blocked, the watchdog timer will retransmit a request for allocation if the other host has previously missed the request. The watchdog timer will be further explained at a later time.

Once allocation has been obtained, data pending transmission is broken into the PUP packet size that the reader allocation indicated. This data packet is also saved on the write buffer queue for possible retransmission should the information not reach the reader.

Before transmitting, it is necessary to determine if an acknowledgment is needed after this transmission by comparing the number of PUP packets sent against the number of PUP packets from the current allocation. If this transmission will fill the current allocation quota, then a Data PUP is changed to an AData PUP and a Mark PUP is changed to an AMark PUP. An AData or AMark PUP will cause the reader to transmit a new allocation block.

At this point, the PUP write routine is called with the data to transmit. The necessary pointers and counts, for the packet sent, are updated and the watchdog timer is synchronized. This resynchronization stops the watchdog timer from interrupting the BSP package when there are no operations for the watchdog to perform. The watchdog timer function is discussed in Section 6.3.8.

Last, the BSP write routine determines if it is blocked for transmission by examining if the allocation quota has been met. If blocked, then before the next loop to transmit data, the write routine will need to receive an allocation from the reader.

6.3.7 Acknowledgement Processing

The BSP package must be able to transmit or receive acknowledgements. These two functions are divided into two routines contained in the same module.

The "BSPwtACK" transmits to the writer the current position of the receiver byte ID along with an allocation block indicating the state of the reader buffers. It also flags if a zero allocation is being sent. This flag is used in the BSP read routine to determine whether a gratuity acknowledgement packet should be sent. As an option, the "BSPwtACK" flushes received data packets that are not being acknowledged prior to transmission of the allocation block. The flushed packets have been received out of sequence and since they are not being acknowledged, will be retransmitted. The flushing assures the BSP read buffers will not overflow, however, the sequencing algorithm will discard any duplicated packets.

The other acknowledgement process, "BSPrdACK", is not as simplistic as the "BSPwtACK". The read acknowledgment process, shown in Figure 6.6, must interpret the acknowledgement packet and interface with the BSP write function. Upon receipt of an acknowledgement packet, the packet ID is checked against the previous acknowledgement packet to detect a delayed duplicate. Once determined good, any acknowledged packets being held for retransmission are freed. This is done by comparing the acknowledgment packet's ID against the queued for possible retransmission packet ID. Now, the allocation block is examined for the reader's allotment. Next, any data packets pending retransmission are sent. There will always be enough room in the receiver, otherwise these packets would not have been sent previously. The last step is to determine if the writer is blocked. This can happen if the writer process is faster than the reader process, hence the reader's BSP buffers may get full and the acknowledgement packet could contain zero allocation.

6.3.8 Watchdog Timer

The watchdog timer has been mentioned a few times previously while its function has not totally been explained. The watchdog's purpose is to keep the BSP package and the Ethernet running smoothly. It has the task of breaking deadlocks, i.e., both server and user waiting to receive data.

In this implementation, the watchdog is setup to run when a three second timer expires. When the watchdog code executes, it reads any pending packets by calling "BSPreceive". If the BSP write is blocked, the watchdog will transmit an AData packet containing no data. This will cause the remote host reader to transmit a new allocation for the writer and free the blocked transmission.

6.3.9 Queue Package

The queue package is responsible for maintaining the BSP buffers. Routines are available for placing data on either end of the queue or after a particular node of the queue. These are necessary for sequencing packets as they are being read. The remaining queue routine removes data from the front (the oldest data) of the queue.

The BSP package maintains three queues: (1) a free buffer list, (2) a read buffer list for sequencing incoming packets, and (3) a write buffer list for possible retransmissions.

6.4 FTP Level

The FTP code, written for this project, consists of two separate programs; a user program called "ftp" and a server program called "ftpserver". The "ftpserver" is started by the system and waits for an RFC from the "ftp" program initiated by a person desiring to transfer data. The code is best viewed as being split into three sections; (1) the code specific to the user program, (2) the code specific to the server program, and (3) the code common to both the user and server programs. The paragraphs below describe the three sections of the FTP code.

6.4.1 Common Routines

The FTP code consists of some common support routines, which are divisible into three major areas. The first set of common routines read or write property lists, extract information from property lists, or create property lists. The second set of common routines support the FTP command/response language by reading or writing the appropriate information. Finally, the actual transfer of a file is handled by two routines in the last set of common support routines. The pertinent common routines from each set are mentioned below along with of brief description of their function.

The property list function routines consist of:

"DetermineEOL"	Scans a property list to find the end of line convention for a text file. This routine is needed to determine if the host must perform conversion on the file about to be transferred or received.
"DetermineType"	Scans a property list to find the file type. Valid types are binary and text.
"ScanPL"	Scans a property list to find the value of a specified property. Property values are needed for filename, end of line convention, file type, etc.

The FTP routines for handling the command/response consist of:

"readData"	Retrieve the next packet which should be Data. Any other type of packet is a protocol error.
"readMark"	Read the next Mark byte from the BSP connection. This is used when expecting a known command or response sequence.
"writeData"	Write the Data packet to the Ethernet. This is primarily used to write property list and command responses.
"writeMark	- Write the Mark packet to the Ethernet. Mark bytes are written to begin and end FTP commands.

The file transfer routines consist of:

"FixupEOL"	Perform the necessary end of line conversion for text files between local machine and Ethernet or Ethernet and local machine.
"DiskToNet"	- Transfer data from a file on the local disk to the remote host via the established Ethernet connection.
"NetToDisk"	- Transfer data from the remote host to a file on the local disk via the established Ethernet connection.

6.4.2 Server Program

The "ftpserver" is best viewed as a diagram such as that shown in Figure 6.7. The main routine is responsible for initialization of the software, opening the Ethernet connection, waiting for an RFC, and "forking" a server task for each unique connection. Fork, is a UNIX concept that allows creation of a new image process, identical to that of the calling process. The original calling process is known as the parent of the new image process, and that new image process is known as a child of that parent. Once a server task has been forked, the parent "ftpserver" resumes its task of waiting for RFC packets and the child "ftpserver" becomes the remote server for the established connection. Note that the parent "ftpserver" never exits.

Once the connection is open, the remote server waits in the "Connection" routine of the "ftpserver" for an FTP command from the user. Failure to receive a command within five minutes will be considered an inactivity timeout. Once a timeout has been detected, the connection is aborted by sending an abort packet to the FTP user and the child "ftpserver" exits.

When a command is received, it is processed by that appropriate FTP command routine. Each FTP command has its own routine; they are: "DoComment", "DoDelete", "DoDirectory", "DoStore", "DoRetrieve", and "DoVersion". These routines use the common command/response sequence handling routines, described above, and execute the FTP command/response sequence mentioned previously and shown in Figures 5.9 through 5.13.

The "ftpserver" is divided into two modules; that just described above and utilities that are used only by the "ftpserver". The utilities include functions that check the access filemodes, prepend a directory to make an absolute pathname, check the login of the user, and expand wild cards for the delete, list and retrieve commands.

6.4.3 User Program

Like the "ftpserver" program, the "ftp" program is split into two modules; the main module and the support routines. "Ftp" has similar structure to that of "ftpserver". The main module of "ftp" is pictured in Figure 6.8. The main routine initializes the software, opens an Ethernet port, and establishes a connection with the remote server. Once a connection is open, the keyboard command interpreter is invoked. The command interpreter routines are contained in there own module, called "command".

Once a command is entered, it is processed by the appropriate routine. As in the "ftpserver", the commands in "ftp" are handled by separate routines. The "ftp" program commands are of two types; those that effect the remote host and those that do not effect the remote host. The routines that process the commands are shown as two separate columns in Figure 6.8. The column at the left, process commands that communicate with the remote server over the Ethernet. These routines are "DoDelete", DoList", "DoRetrieve", "DoStore", and "DoQuit". Like in the "ftpserver", these routines use the common command/response sequence handling routines, described in Section 6.4.1, and execute the FTP command/response sequence mentioned previously and shown in Figures 5.9 through 5.13. The column at the right of Figure 6.8, process commands that effect local variables and does not require the Ethernet. These routines are "DoLogin", "DoDirectory", "DoEOL", "DoType", "DoChange", and "DoShell". Most of these routines set variables that will be sent to the remote server with the next property list.

As mentioned previously, the "ftp" routine also has its own set of utility functions. These functions include routines to determine the local file type, obtain and open a local filename, and determine if a filename exists on the remote server.

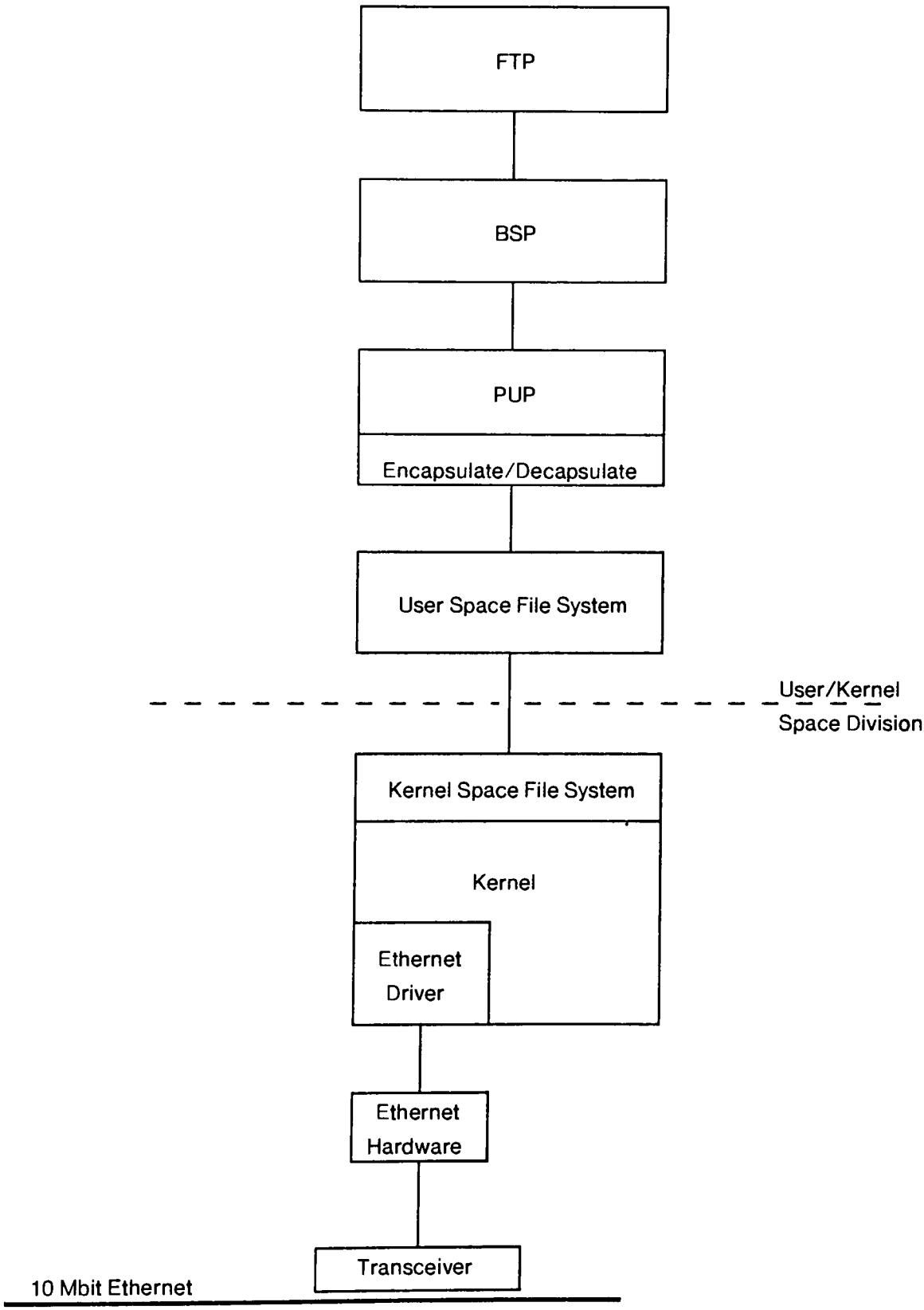


Figure 6.1: Kernel and User Space Relationships

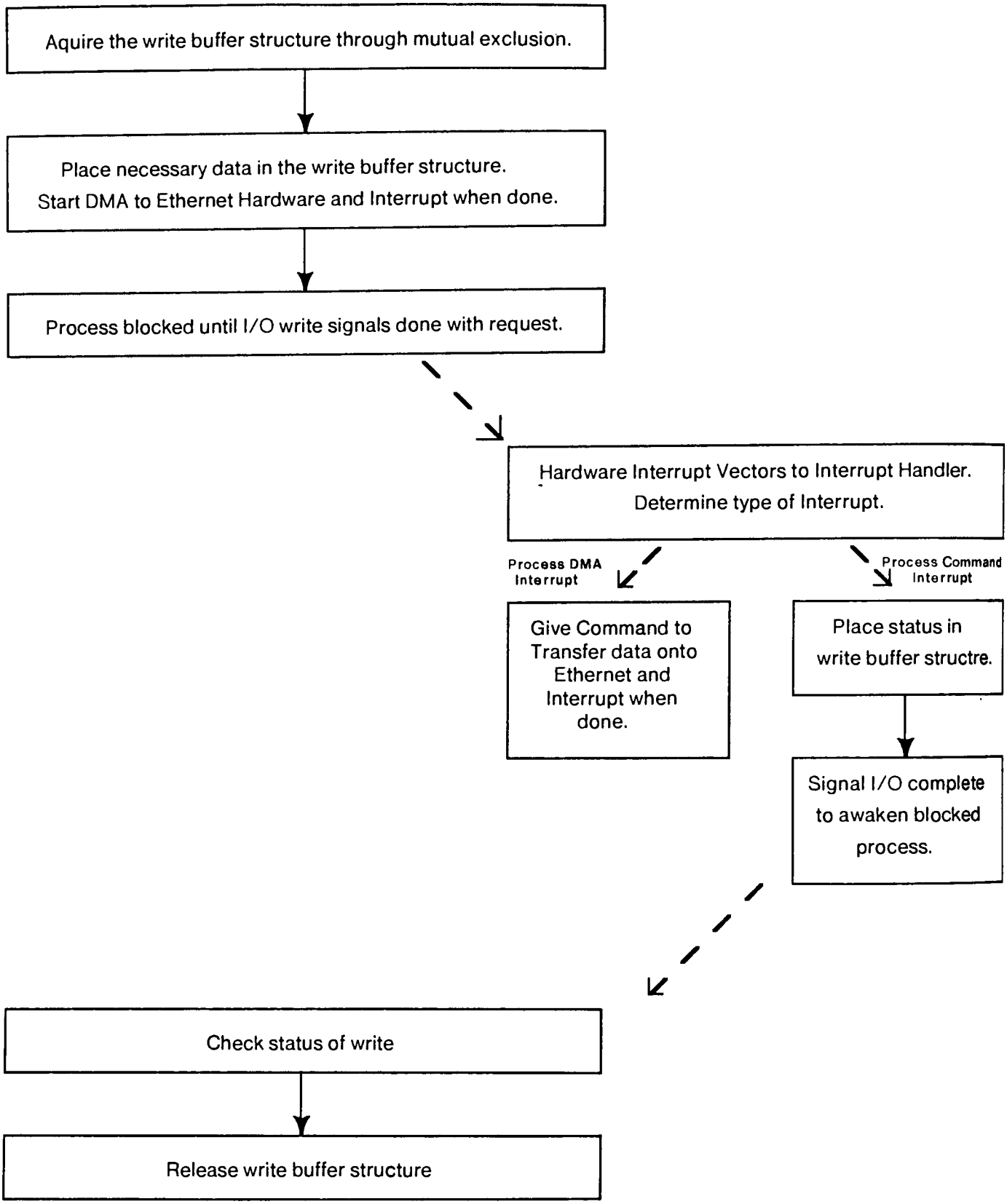


Figure 6.2: Write Data/Control Flow Through Driver

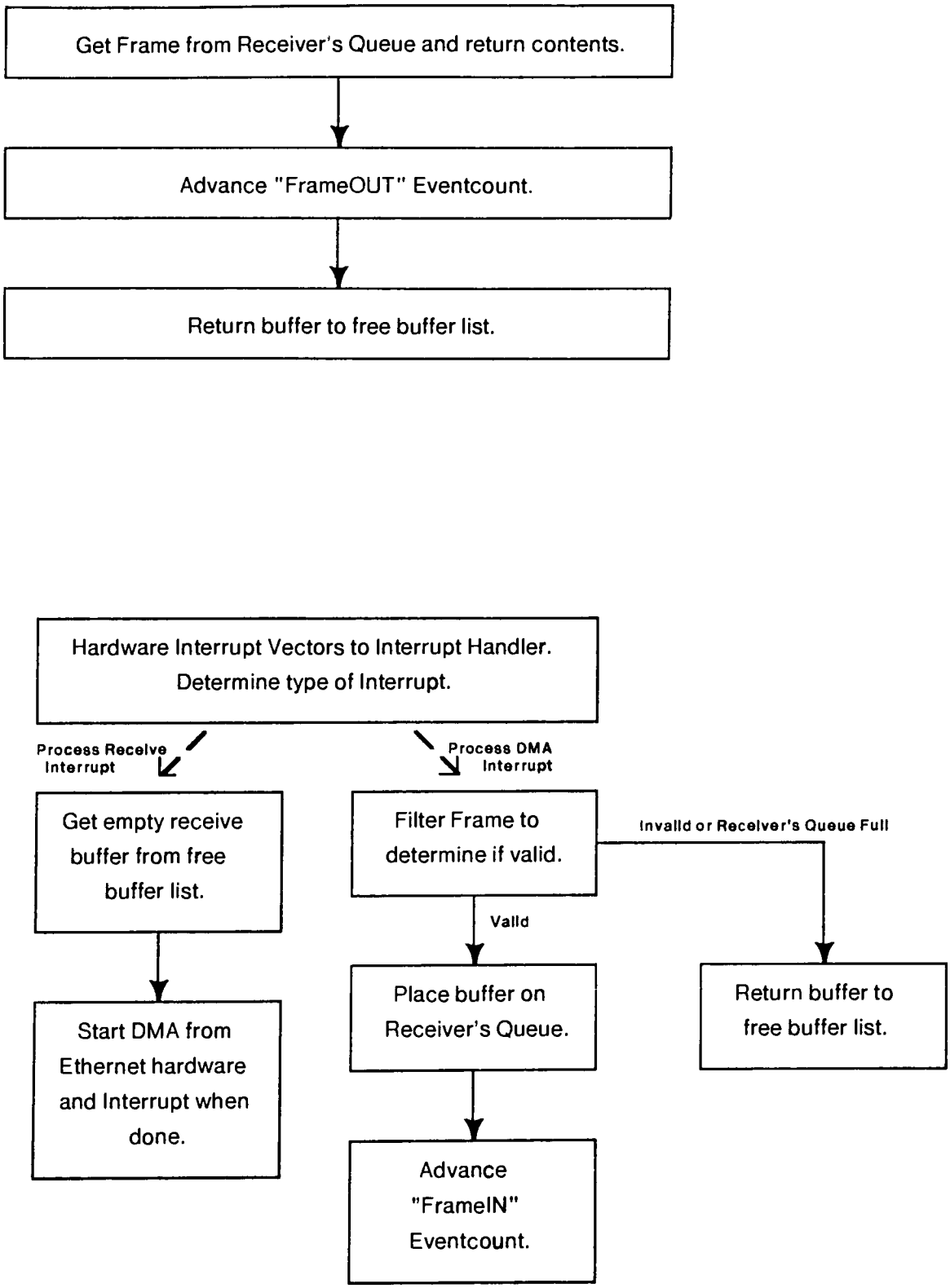


Figure 6.3: Read Data/Control Flow Through Driver

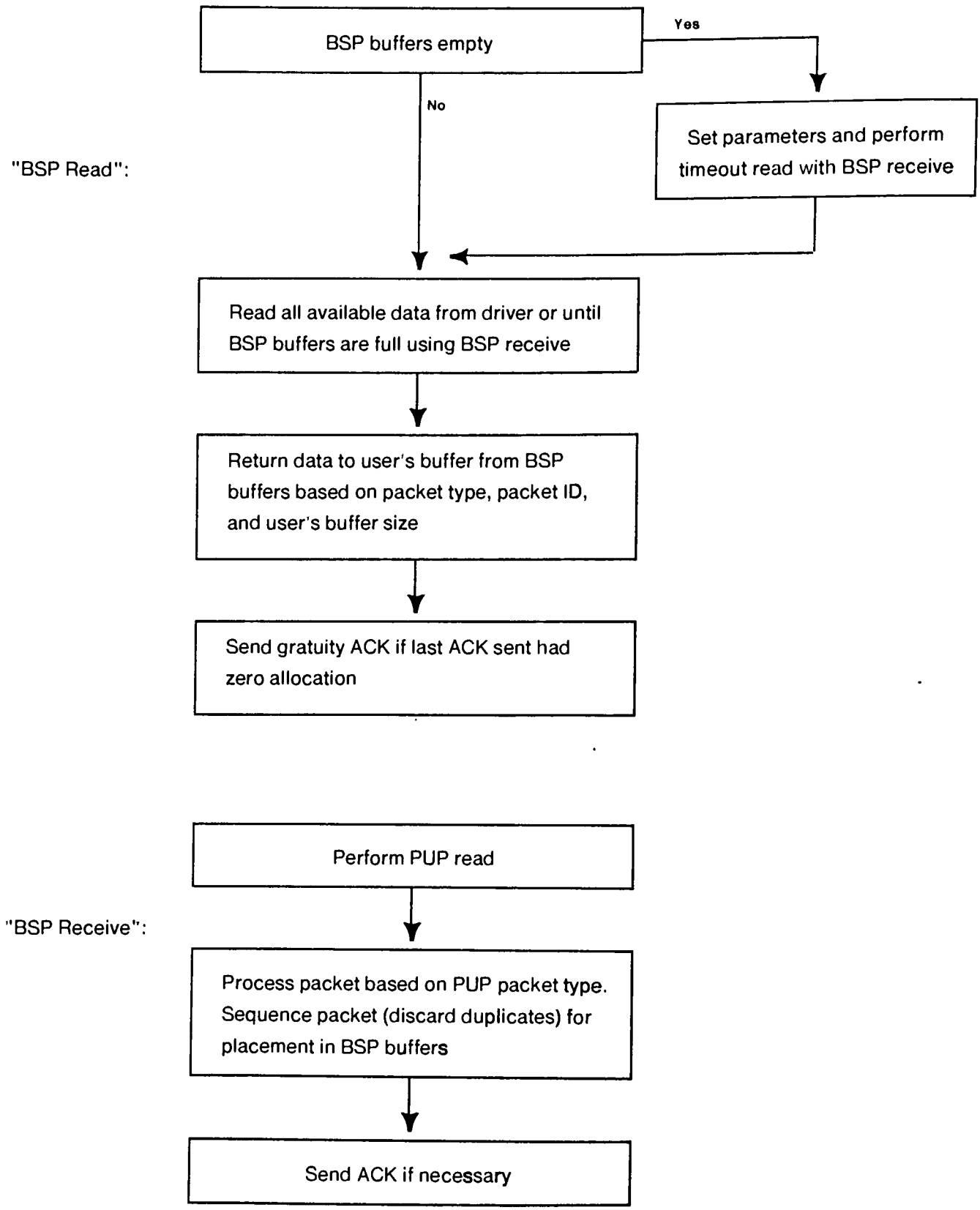


Figure 6.4: BSP Read Function

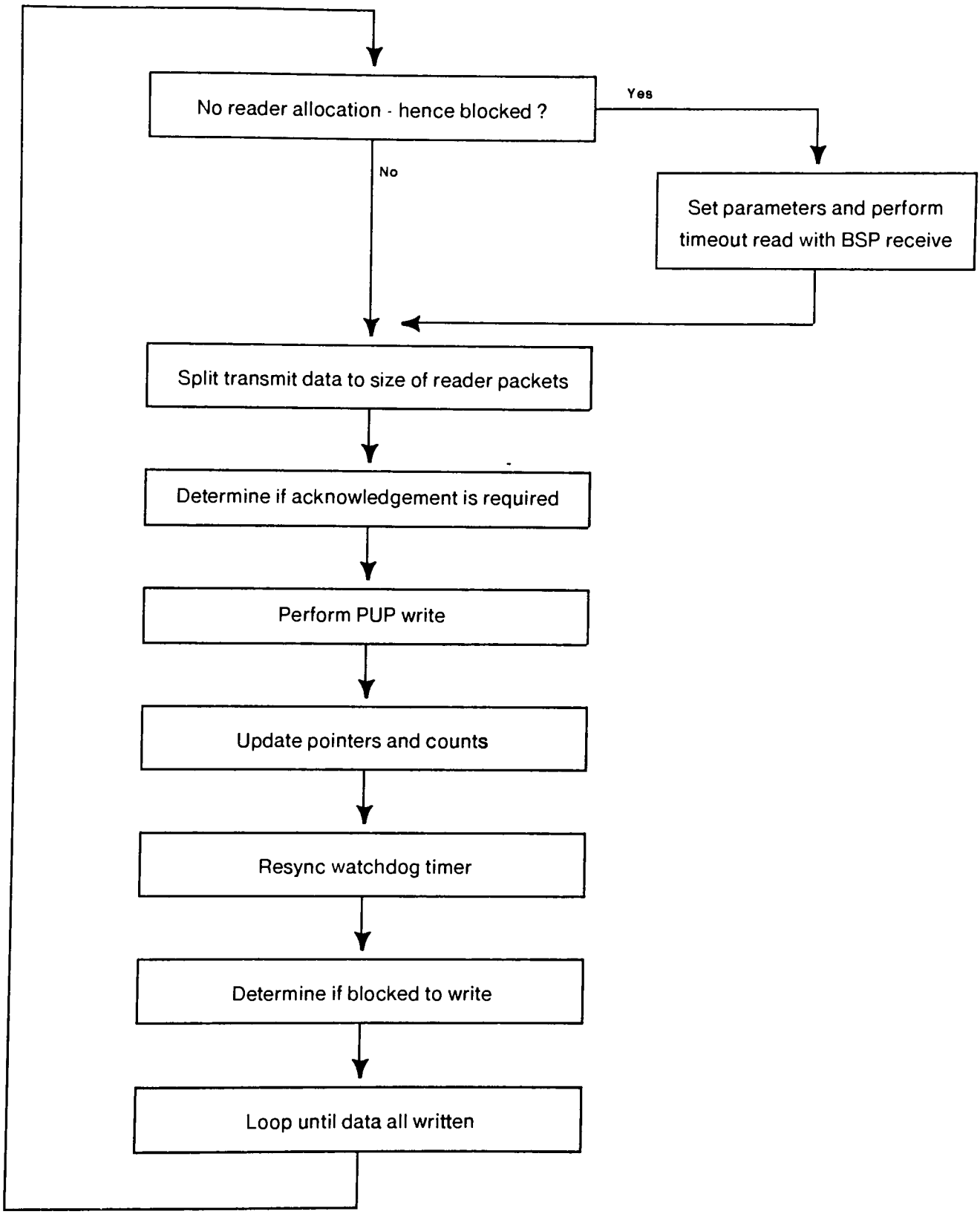


Figure 6.5: BSP Write Function

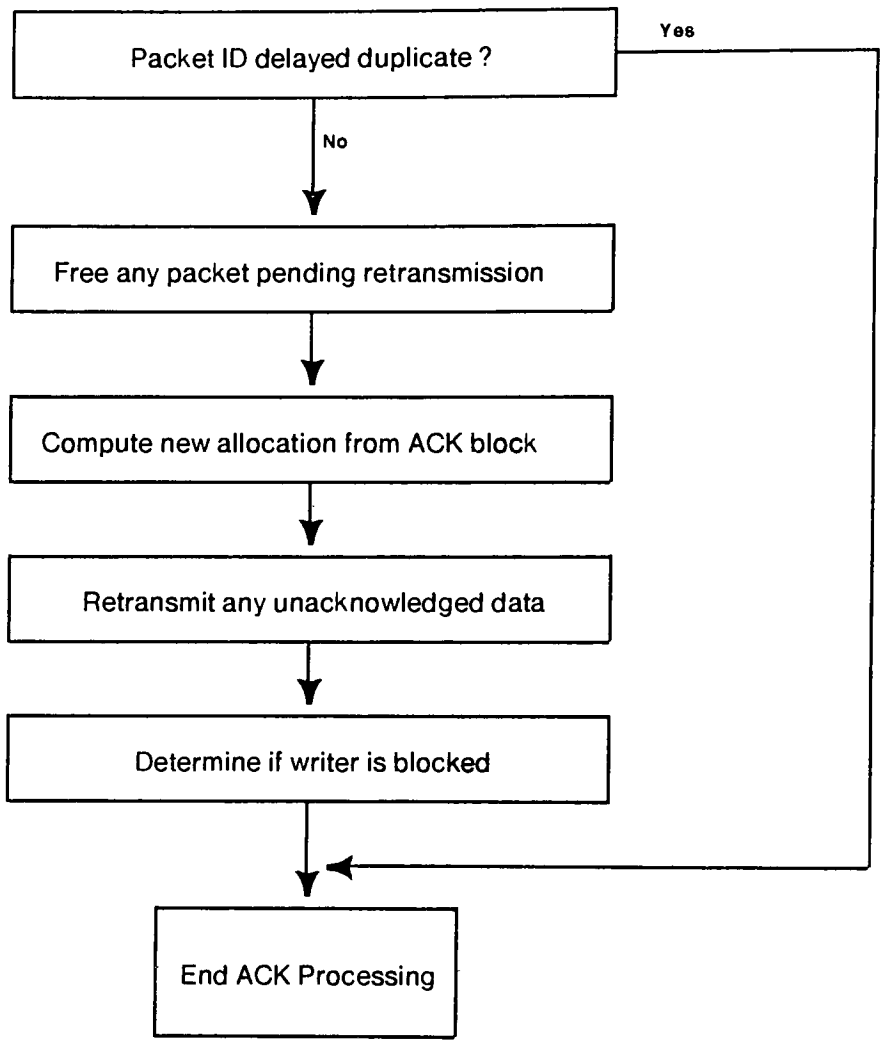


Figure 6.6: BSP Read Acknowledgement Processing

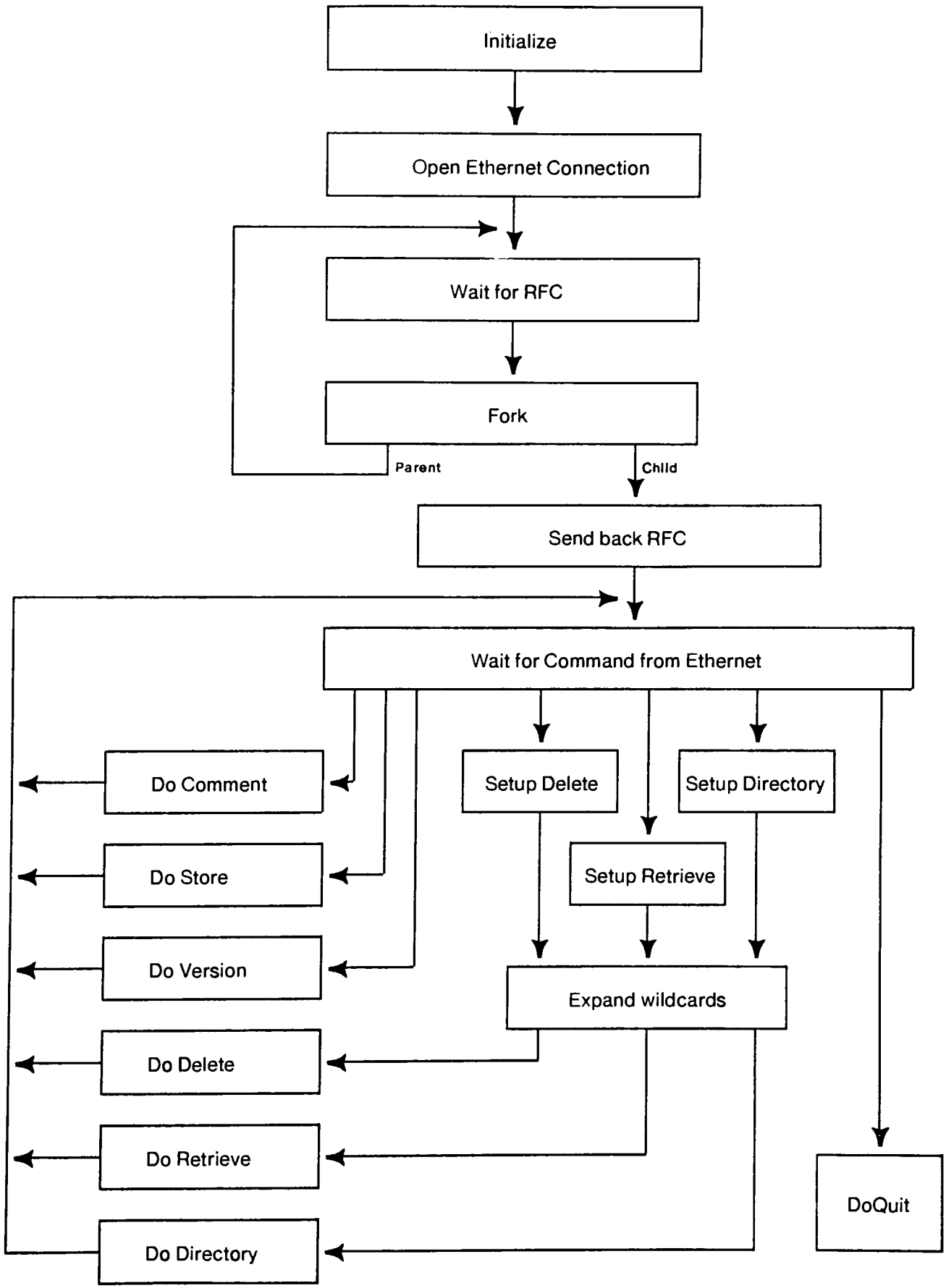


Figure 6.7: FTP Server Process

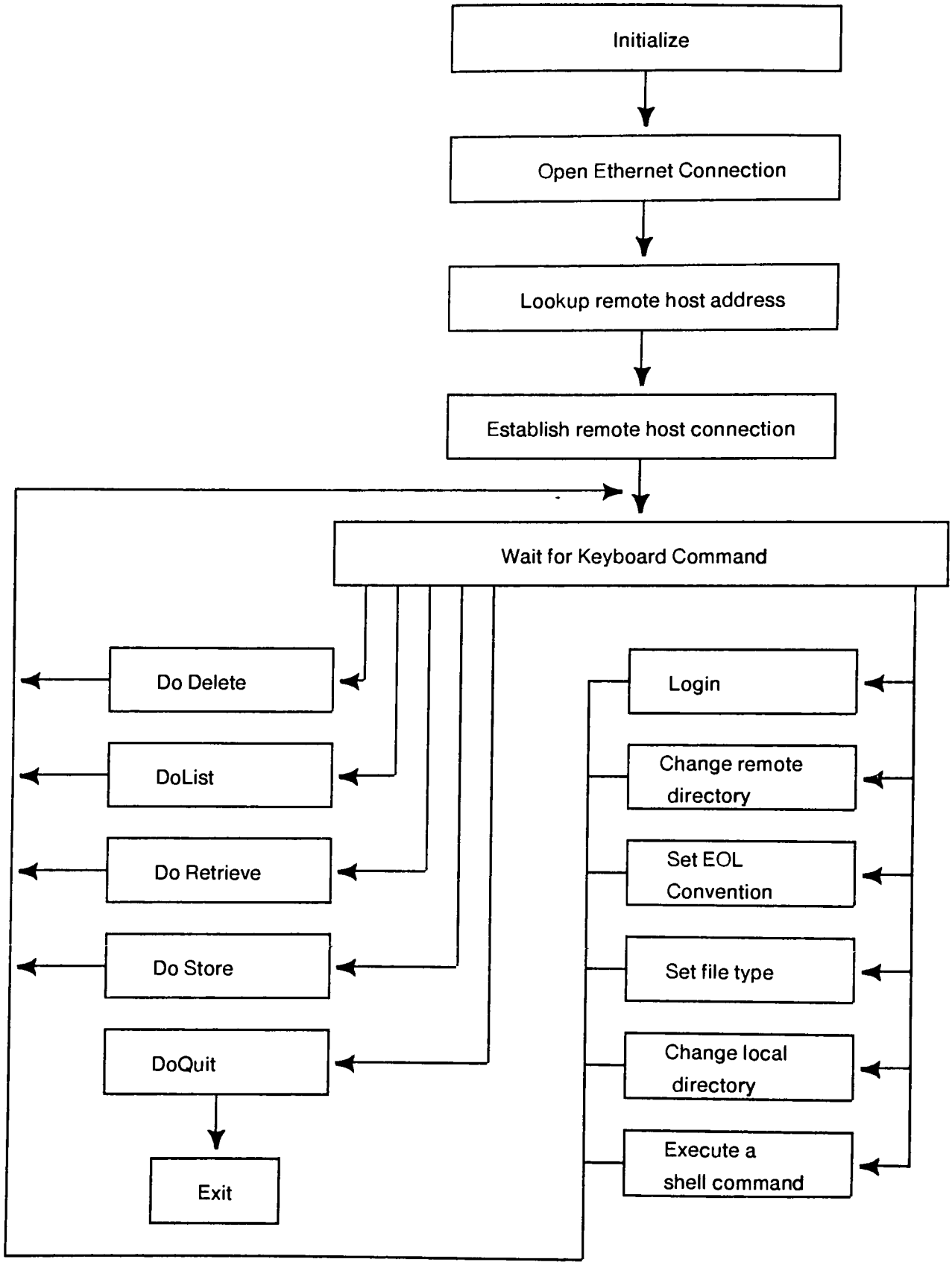


Figure 6.8: FTP User Process

7. Verification and Validation

The structured approach taken for the thesis project gave various checkpoints along the way to test each level's functionality. While it was only the end result of being able to transfer a file that truly mattered, the ability to test each point along the way greatly reduced the overall debug at the final stage. Intermediate checkpoints also produce a higher level of confidence in the reliability of the overall system. The entire FTP, BSP, PUP and Ethernet driver are much too complex to debug all at once. The confidence level produced with an "all at once" debug approach would be much lower since upper levels could tend to mask lower level problems.

Verification of each level of code was approached based upon the functions necessary for that level to perform. Each level was tested separately, with independent test programs, written to focus on the functions that the level must perform. A brief outline of the testing at each intermediate level along with the final level of testing performed follows.

7.1 Checkpoint Testing

The Ethernet driver was tested with two programs, one for transmit and one for receive. The two programs worked together to validate the Interlan hardware and Ethernet driver's ability to read and write data by checking the contents of the frames for specific data and/or displaying the contents on the terminal screen for visual examination.

Once the driver was working, various tools were written to aid in further levels of debug. One such tool was an Ethernet watch program used to display the contents of any frame on the Ethernet coaxial cable. This allows the examination of packet contents and gives a check of Ethernet activity. An Interlan board diagnostic program was also written for testing the hardware.

After writing the PUP software, a "puptest" program was written. This program works in two modes. Invoked with one option, it would receive a packet in the well known echo socket and then send an echo packet back to the original host's socket. Invoked with a different option, it would transmit a packet to the well known echo socket and wait for the echo reply. This echo test gave the ability to debug the routing of packets and examine PUP packet format.

Finally, the BSP level was written and cursory testing performed. Since it was necessary to generate retransmission and sequence testing, the final testing was left to be performed with the FTP package where such conditions were easier to generate.

7.2 FTP Verification

The testing of the FTP program consisted of many stages, from the simplest of logging in to the most complex of transmitting a file under heavy computer system and Ethernet load. File access privileges and security were the most difficult to test due to all the possible permutations that exist. The debug and verification of the FTP commands was not difficult, but it was time consuming with the numerous commands that exist.

Initial verification began by logging in and listing the contents of a remote directory. A file, which previously did not exist in that remote directory, was then transferred to that remote directory. When completed, the directory contents were displayed again and the transferred file existed. The remote file was then compared against the original file to detect differences; none were found, with binary or text files. These tests were performed with the store and retrieve commands. A variation of these tests were also performed to attempt overwriting an existing file. Both, store and retrieve, were able to correctly determine if a file already existed and query the user for an overwrite confirmation.

Once the initial testing was completed, the permutations of file access was debugged. Different world and owner access privileges for file read and write were tried, i.e., retrieving a file with insufficient read privilege or storing a file with insufficient write privilege. Remote file deletion requires the same access priviledge as remote file stores, i.e., write access. Other permutations of this testing included administrator login (which has all privileges), examining file ownership rights, and verifying the existence of all directories in a specified path name. The correct error messages were generated for all cases.

Final testing required inducing errors in the Ethernet transmission to detect error recovery and retransmission capabilities. Modification of the code was avoided since this could disguise the original code's ability to react to the errors. The types of tests that were performed are (1) retransmission recovery and sequencing by physically disconnecting the host from the Ethernet for a short period of time, (2) receiver/transmitter timeout by similar disconnect technique, and (3) loading factor by busying the host and Ethernet with simultaneous FTP transfers. The results were visible by invoking "ftp" with the -b option (BSP debug level) and watching for messages.

7.3 Final Test Results

The above tests were extensive and gave a high confidence in program correctness. However, proving code correctness is a bigger obstacle when a programmer has to verify his or her own code. The role of designer, programmer, and tester imbeds program functionality biases. Thus, some obvious test could be overlooked due to false assumptions. To the best of this programmer's understanding, the design and verification conform to the specifications in the thesis proposal.

8. Conclusions

The implementation of the FTP project required extensive research to understand the general Ethernet capabilities. Such capabilities included the FTP command/response sequence, the BSP flow control, and the PUP packet formats. Samples of previous implementations, together with the documentation obtained, gave a clear understanding of the necessary task that laid ahead. The only difficulties encountered were in writing the Ethernet driver, due to the lack of documentation in the UNOS kernel. While the driver does work, it may not be the optimal coding solution.

The FTP package performs to the specifications stated in the thesis proposal. A user may transfer a binary or text file to or from a remote host. Additional functions allow the listing or deleting of files from that remote host. FTP protects the remote file system by requiring a user to login and only allowing file access based on the file modes. One shortcoming of the current implementation was not checking if a file would fit in the remaining disk space available. This allows a user to overflow the disk by storing a file of larger size than the disk can accommodate. This shortcoming was stated in the proposal, however, it should be fixed.

8.1 Lessons Learned

8.1.1 Alternative Approaches for Improved System

As with any communications package, increased speed would be a desired improvement. Studies would need to be done to locate the bottleneck areas and perform the necessary code modifications to decrease or eliminate the bottleneck. Assembly language routines may be incorporated to replace the more frequently used routine. This is not a preferred approach due to the maintenance issue of assembly language code.

One approach to better the structure of the code, would be moving the receiver frame buffering out of the driver (kernel space). The desired buffering could be performed in either the PUP package or a special buffering package, both of which are user space routines. To enhance this idea, the UNOS operating system now allows programs to share memory, a feature not available at the time this project was started. The buffering program would be started by the system, as the owner of the shared memory. This system buffering program would receive all Ethernet frames, buffering them (to a limit) for each socket. When an FTP program was initiated, it would connect to the system buffering program instead of the Ethernet driver. The shared memory would eliminate the necessity of copying the packet from the system buffering program to the FTP program, thereby increasing the speed of the communication package. The advantage of buffering Ethernet frames in user space helps eliminate the size of the kernel code. Currently, space for receiver buffering is allocated at system boot time. If the Ethernet is never used, that space is wasted. The system buffering program approach could allow for dynamic buffering and only use memory when needed.

8.1.2 Suggestions for Future Extensions

The current FTP program implements the mandatory FTP properties and some of the optional FTP properties. One unimplemented optional properties was *rename*. This would allow the user to change a remote filename or its attributes. Another command not implemented for the thesis project, was *compare*. It would verify that a file on the local host was identical with a file on the remote host. These two command would make nice enhancements to the current FTP package.

For security reasons, FTP requires that the user must login with his or her name and password for the remote host. If while attempting a command, i.e., store, the remote host

detects an illegal user login, the command is aborted and FTP reprompts. FTP could be modified to detect the remote server's response of illegal user login and, instead of the FTP prompt, could prompt for the user's login. After receiving the login from the keyboard, FTP would retry the command that was previously aborted for lack of the correct user login.

8.1.3 Related Thesis Topics for the Future

Once an Ethernet is connected to a machine, there seems to be an unlimited number of applications that can benefit from the high speed, packet switched communications that an Ethernet offers. Two programs that could run on an Ethernet based system are chat and mail. Chat would allow a terminal on the local host to talk over the Ethernet to a remote host and appear as if that terminal was connected to the remote host. This gives a user the ability to edit, compile, and run programs on the remote system that may not be available on the local system, i.e., distributed resources.

Electronic mail could benefit by the increased speed of letter delivery. The Xerox experimental 3 Mbit Ethernet does perform mail delivery in such a fashion.

Bibliography

Boggs, D. R., Shoch, J. F., Taft, E. A., Metcalfe, R. M., "PUP: An Internetwork Architecture", *IEEE Transactions of Communications*, April 1980.

CP-32 User's Manual (CRDS: 97-30508-01), **Charles River Data Systems,** May 1983.

Crane, R. C., "Software Pack and Controller Link DEC Computers in an Ethernet", *Electronics*, December 1981.

The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications, Version 2.0, **Digital Equipment Corporation, Intel Corporation, and Xerox Corporation,** November 1982.

International Organization for Standardization (ISO), "Data Processing Open Systems Interconnection Basic Reference Model (Document no. ISO/TC97/SC16)", *Computer Networks*, April 1981.

Kernighan, B. W. and Ritchie, D. M., The C Programming Language, New Jersey: Prentice-Hall Software Series, 1978.

Metcalfe, R. M. and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, July 1976.

Metcalfe, R. M. and Taft, E A., "Pup Specification", Xerox Palo Alto Research Center, June 1978.

MC68000 16-Bit Microprocessor User's Manual, Second Edition, **Motorola,** January 1980.

The NI3010A MultiBus Ethernet Communications Controller User Manual, Rev 00, **Interlan Incorporated,** November 1982.

Shoch, J. F., Dalal, Y. K., Redell, D. D., and Crane, R. C., "Evolution of the Ethernet Local Computer Network", *Computer*, August 1982.

Shoch, J. F. and Hupp, J. A., "Measured Performance of an Ethernet Local Network", *Communications of the ACM*, December 1980.

Shoch, J. F. and Taft, E. A., "Pup File Transfer Protocol Specification", Xerox Palo Alto Research Center, September 1983.

Shoch, J. F., Cohen, D., and Taft, E. A., "Mutual Encapsulation of Internetwork Protocols", *Computer Networks*, July 1981.

Synergist Product Specification Manual, **Hal-Versa Engineering Incorporation,** 1982.

Tanenbaum, A. S., Computer Networks, New Jersey: Prentice-Hall Software Series, 1981.

Tanenbaum, A. S., "Network Protocols", *Computing Surveys*, December 1981.

UNOS Commands Manual (CRDS: 97-21318-5), **Charles River Data Systems,** January 1984.

UNOS Drivers Release Notes (CRDS: 97-31069), **Charles River Data Systems,** March 1984.

UNOS Installation and Management Guide (CRDS: 97-300500-03), **Charles River Data Systems,** January 1984.

UNOS Subroutines Manual (CRDS: 97-21320-4), **Charles River Data Systems,** January 1984.

Zimmermann, H., "OSI Reference Model The ISO Model of Architecture for Open System Interconnection", *IEEE Transactions on Communication*, April 1980.

Appendices

Appendix A: PUP Checksum Calculation

The PUP Checksum is a 16-bit, one's complement, add-and-cycle checksum computed over the 16-bit words in the PUP's header and contents. It is intended as an end-to-end reassurance of correct transport by intermediate hardware and software components. The Checksum is not associated with or a replacement for any network's existing error checking mechanisms (which are usually specialized to detect the specific sorts of errors commonly encountered on that network).

The checksum is initialized to 0 and computed by repeated one's complement addition and left cycle, starting with the PUP's Length word and ending with the last content word. Note that the checksum includes the garbage data byte if there is one. If the result is the one's complement value is "minus zero" (Hex FFFF), it should be converted to "plus zero" Hex FFFF is specifically defined to mean that the PUP carries no checksum.

The PUP's checksum is carried from source to destination. If and when a PUP is altered enroute, say the hop count in the transport field is incremented, its checksum must be recomputed. The choice of the one's complement add-and-cycle is intended to permit incremental checksum updating. The algorithm for updating the checksum after changing a single word of the PUP is as follows (one's complement arithmetic used throughout):

1. If the PUP Checksum is Hex FFFF, do nothing.
2. Subtract the old contents of the changed word from the new.
3. Left-cycle this difference (n mod 16) bits, where n is the distance (in words) from the changed word to the PUP Checksum word.
4. Add the result to the existing PUP Checksum.

The foregoing procedure produces a correct PUP Checksum if and only if the original PUP Checksum was correct.

Appendix B: PUP File Transfer Protocol Specifications

XEROX Palo Alto Research Center

September 29, 1983; 7th edition

To: Communication Protocols

From: John Shoch and Ed Taft

Subject: **Pup File Transfer Protocol Specification—7th edition**

File: <Pup>FTPSpec.press

This document represents an attempt to define a simple protocol for the movement of files among the many different computers currently connected to the PARC communications networks.

This document provides a general explanation and overview of the full range of the File Transfer Protocol, as well as a specification for the protocol. Specific details of the actual FTP implemented for Maxc or the Alto may be found in the appropriate documentation.

All previous versions of this memo are now obsolete. Material in a small font concerns facilities whose design is incomplete and which have never been implemented.

The development and refinement of the protocol have benefited greatly from comments and ideas originally provided by Bob Metcalfe, Ed Taft, David Boggs, Bill Duvall, Bob Sproull, and others; and supplemented by later experience in Mesa by Smokey Wallace, Jim White, and Hal Murray.

1. Companion documents

Background information on Pup and BSP (on-line versions are available on many public file servers):

—Bob Metcalfe and Ed Taft, "Pup Specification", file <Pup>PupSpec.press.

—D. Boggs, J. Shoch, E. Taft, and R. Metcalfe, "Pup: an Internetwork Architecture", *IEEE Transactions on Communications*, COM-28 4, April 1980, file <Pup>PupPaper.press.

Detailed description of particular FTP implementations:

—Ed Taft, "PUPFTP—Maxc Pup File Transfer Program", file <Pup>PUPFTP.press.

—Dave Boggs and Ed Taft, "BCPL FTP Package", in the *Alto Subsystems* manual or file <AltoDocs>FTPPackage.tty.

—Jim White and Hal Murray, "Mesa FTP Functional Specification", file <Mesa>Doc>FTP.press.

A related protocol:

—Ed Taft, "Pup Mail Transfer Protocol", file <Pup>MailTransfer.press.

2. Assumptions and basic principles

If you are reading this document, it is assumed that you are familiar with the contents of "Pup Specification" by Bob Metcalfe and Ed Taft.

We are talking about files transmitted from one logical process to another logical process, and not necessarily from one file system to another file system. It is possible, for example, to generate a file 'on the fly' with one process and then have it consumed by another process, without it ever residing on a physical disk.

We are not discussing particular programs which might chose to use the File Transfer Protocol, nor are we concerned here with implementation questions. Programs such as a File Transfer Program, a BCPL remote-stream facility, mail systems, or an automatic disk update program can make use of the File Transfer Protocol; but that is a matter of convention.

It should be possible to construct a simple file transfer protocol as a thin veneer placed over the Byte Stream Protocol (BSP). While the mix of machines and networks is very heterogeneous, these differences are subsumed by the definition of Pups and the Byte Stream Protocol. The BSP is designed to provide "...error- and flow-controlled byte streams between cooperating processes." Thus, the only parts of the 'outside world' which the FTP will ever see—or use—are the facilities provided by the BSP.

The File Transfer Protocol only comes into play after a BSP stream has been established. That is, actual programs will use the Rendevous and Termination Protocol (or some other means) to establish a connection between two ports. The FTP does not define this mechanism.

One BSP connection is sufficient for transmitting a file. Unlike the Arpanet protocol, both command information and data pass through the same bi-directional BSP stream.

The two parties to a file transfer are named the *user* and the *server:* the user initiates actions and the server responds. Actual files, however, may move in either direction: a user process may try to *store* a file upon the server system, or try to *retrieve* a file from the server.

The File Transfer Protocol generally prescribes the behavior of the server process; the user process is merely one that presents requests to the server in accordance with the server's described format, and follows the command/response sequence.

3. On the nature of files

It is common to say that a file is merely a collection of bits, but that is, of course, over simplified. A file is a collection of bits viewed in a particular way within a specified context. Much of the work involved in transferring files from one machine to another over some number of different networks is the result of attempting to resolve conflicting contextual representations of a file: 7, 8, or 36 bit, ASCII or binary, naming conventions, CR or EOL, *ad infinitum.*

In addition, such contextual differences make it hard to add a new machine or different network to a multi-network system without redoing the resolution matrix; and it is also important to avoid reformatting data to conform to local conventions at every step through a collection of networks.

To simplify the situation, this contextual information may be viewed as an adjunct part of a file, which might migrate with the file, as necessary. This information, or *file property list,* may be used to describe different kinds of "files", including those:

—residing on a physical disk;

—in transit, on a BSP connection;

—being sought by a particular user; or,

—being generated by a server.

The properties of a file are, in general, contained either explicitly or implicitly within an operating system on a particular machine. An actual property list may contain information provided by the user, from the file system, or from specialized information about the environment. Consider a typical scenario: moving a text file from a file server to an Alto:

1. The server can generate a property list including the file name, date last written, author, protection, byte size, etc.

2. This information, along with the file, is transmitted to the Alto.

3. The Alto process does its best to map the property list into its environment: the file name is used, perhaps without the version number; protection information is discarded; the end of line indication may be changed, if necessary; etc.

There is no guarantee that a receiving process will be able to properly encode all properties.

A file could, however, attempt to preserve some of its original context by including as one of its properties an extraordinary request to maintain the property list. For example, an Alto might be willing to save a property list on an auxiliary file, and use some of that information to regenerate a property list when the file is returned to the server.

A property list may be generated by any process which desires to transmit a file. Similarly, any process receiving a file and its property list can use that information to interpret the contents of the file. (Remember that all data moved under the BSP are sent as 8-bit bytes, and the property list will indicate any unusual packing of data.)

The property list might also be used to merely provide information about one or more files (where they are, when last written, etc.) In addition, a file property list can provide a partial description for a file which you are trying to find, or which you would like to create.

This notion of a file property list is central to the file transfer protocol.

4. Summary of the File Transfer Protocol

There are three major elements defined in the protocol: syntax of a property list, typical property/value pairs, and command sequences. A general description of each is provided here as an introduction; the completely specified details can be found in subsequent sections.

4.1. Syntax of a file property list

A file property list consists of a string of ASCII characters, beginning with a left parenthesis and ending with a matching right parenthesis. Within that list, each property is represented similarly as a parenthesized list. For example:

((Server-Filename TESTFILE.7)(Byte-Size 36))

4.2. Typical properties and their values

One can reasonably expect to find properties such as these in a file property list:

Server-Filename	<SUBSYS>PUPFTP.SAV;3
	<Alto>Bravo.dm
	Bravo.dm
	My.Funny.But.Legal.Alto.Name

Byte-Size	7, 8, 36
Type	Text, Binary
Author	PanchoVilla, System
Date-Written	23-Jan-76 11:30:22 PST

4.3. Commands and command/response sequences

In every FTP connection, one party is the User and the other party is the Server: the User initiates commands, while the Server provides replies. In general, there will be some form of reply generated in answer to every command.

Every transaction in the FTP consists of a BSP Mark Byte, followed by a piece of data, and terminated at the next BSP Mark Byte encountered. The data thus enclosed might be one or more file property lists, a text string, or a file. A command might simply be terminated by the arrival of the next Mark Byte command, or by an End-of-Command Mark Byte (essentially a no-op). All of the Mark Byte codes used in the FTP are listed at the end of this document.

For example, a simple retrieval of a file would include:

1. User sends the Server a command requesting the retrieval, with a file property list describing the desired file.
2. Server responds with an appropriate message.
3. If the Server said Yes, then the file is placed into the byte stream.

If a process replies with a No, the first byte of the following data is a numeric code for possible machine processing, and the remaining bytes are a comment for the user. (The correct functioning of the protocol, however, does not depend upon the many possible numeric codes; the simple Yes and No messages are sufficient for using the protocol.)

Once again, particular implementations of the FTP may not support all possible commands. The protocol provides a standard response (the No message with a specific code) by which a server may indicate that a requested command is not implemented. This must be the server's sole response to an unimplemented command; it *must not* take more drastic action such as breaking the connection.

5. Details of the File Transfer Protocol

This section specifies the minimum conventions required to use the FTP, as well as many optional features which have been found useful. Any FTP server intended for general use is expected to respond properly to the minimum set of commands; any of the optional facilities included will increase the utility of the server. Any implementation of a user process is free to select those features deemed useful, but the minimal set should seem evident from the discussion below.

5.1. Syntax of a file property list

A file property list consists of a string of ASCII characters, beginning with a left parenthesis and ending with a matching right parenthesis. Within that list, each property is represented similarly as a parenthesized list. For example:

((Server-Filename TESTFILE.7)(Byte-Size 36))

This scheme has the advantage of being human readable, although it will require some form of scanner or interpreter. Nevertheless, this is a rigid format, with minimum flexibility in form; FTP is a machine-to-machine protocol, not a programming language.

The first item in each property (delimited by a left parenthesis and a space) is the property name, taken from a fixed but extensible set. Upper- and lower-case letters are considered equivalent in the property name. The text between the first space and the right parenthesis is the property value. All characters in the property value are taken literally, except in accordance with the quoting convention described below.

All spaces are significant, and multiple spaces may not be arbitrarily included. There should be no space between the two leading parentheses, for example, and a single space separates a property name from the property value. Other spaces in a property value will become part of that value, so that the following example will work properly:

((Server-Filename xxxxx)(Read-Date 23-Jan-76 11:30:22 PST))

A single apostrophe is used as the quote character in a property value, and should be used before a parenthesis or a desired apostrophe:

Don't(!)Goof ==> (PropertyName Don"t'(!')Goof)

FTP implementations are *required* to ignore properties whose names they don't know. This enables extensions to be introduced to the set of defined properties without invalidating existing programs.

FTP implementations are expected to send as complete a set of properties as is possible in every message. However, the FTP user may instruct the server to send only a specific subset of the file properties, using the Desired-Property property described below.

5.2. Defined properties and their values

5.2.1. Mandatory properties. An FTP server must be able to respond to these elements in a file property list. These properties may or may not be required in all property lists; this is discussed in the individual descriptions.

(Server-Filename *filename*)

In commands from user to server (such as Store, Retrieve, and Enumerate), specifies the name of the file to be manipulated. This filename must conform to the conventions of the *server* system (unlike the more machine-independent name convention to be described as options below). This need not be an exact filename: it may be a pattern (possibly matching multiple filenames) or contain defaulted fields—whatever is supported by the server's file system.

In responses from server to user, specifies the exact, complete name of the file presently under consideration.

(Type Text|Binary)

Specifies the type of data to be transferred (no default). Text data is 7-bit ASCII (right-justified in 8-bit bytes) conforming to the End-of-Line-Convention (see below). Binary data consists of bytes conforming to the Byte-Size property (see below). If the type is not specified by the user in a Retrieve command (the usual case), the server should determine the appropriate type based on local information about the file being retrieved, if possible.

(End-of-Line-Convention CR|CRLF|Transparent)

Specifies the representation for end-of-line in files transmitted as type Text (default is CR).

This specification is meaningless for Binary files and is ignored. Both server and user are expected to convert between local end-of-line conventions and the agreed-upon network convention. (In general, the FTP server never independently decides on an end-of-line convention but rather always heeds the convention specified by the FTP user.) Transparent means that the server is not to perform a conversion even if local and network conventions differ. All implementations are required to support end-of-line conventions CR and Transparent; implementing CRLF is optional.

(Byte-Size *decimal-number*)

Specifies the logical byte size of files of type Binary (meaningless and ignored if the type is Text). There is no default. Binary data consists of bytes of the specified number of bits, right-justified in 8-bit bytes or multiples thereof. (While there is no inherent limit to the byte size, some implementations may not accept byte sizes larger than the word size of the machine on which they are running.) During a Store, the byte size must be specified by the FTP user, and it determines the byte size of the file written at the server end. During a Retrieve, the user may leave the byte size unspecified; in this case, the server chooses a suitable byte size based on local information (if possible). If the user requests retrieval of a file using a byte size different from that which the server believes is appropriate for the file, an error results.

(Device *device-name*)
(Directory *directory-name*)
(Name-Body *name-body*)
(Version *version-string*)

Specifies the separate components of a filename in a manner intended to facilitate transfer of files between hosts whose file systems lack one or more of these concepts or have different conventions for punctuating them. The values of these properties should *not* include the surrounding punctuation characters.

A filename property sent from user to server may be given either as a Server-Filename or as separate components. If it is supplied as separate components, then the server is expected, where possible, to default missing components and to map locally illegal characters into legal ones. If, however, it is specified as a Server-Filename, then it is expected to conform exactly to the server's file naming conventions (including avoiding or quoting illegal characters as required). If both a Server-Filename and separate components are supplied, the individual components may be used by the server to default unspecified fields in the Server-Filename. (In effect, this might allow a user program to specify a default directory, unless overruled by the human user.)

A filename property sent from server to user is *required* to be specified as separate components, though optionally it may also be specified by a Server-Filename property (generally just for information purposes). This allows a user who sent an incomplete specification (e.g., a pattern) to then interpret the complete specification that comes back.

The purpose of having two filename conventions is on the one hand to permit mechanical processing of filenames by processes ignorant of other than local naming conventions, and on the other hand to permit exact specification of foreign filenames by (human) users who *are* cognizant of foreign naming conventions.

5.2.2. Optional properties. There are other properties which are considered, in general, to be optional; but all FTP servers are encouraged to implement those which may be appropriate. A server may consider some of these to be mandatory components of a file property list (e.g., User-Name, User-Password), although other servers may not utilize them.

(Size *decimal-number*)

Specifies the number of bytes in the file. (These bytes are of the size specified by the Byte-Size property.) The value is not required to be correct. The recipient of this property may treat it as a hint (e.g., to control preallocation of file space); but the true length of the file is defined solely by the number of bytes actually transferred when the file is stored or retrieved.

(User-Name *user-name*)

Specifies a legal user name on the server system, for the purpose of access checking and accounting, if required.

(User-Password *password*)

Supplies the user password, if required for logging in under the specified User-Name.

(User-Account *account-designator*)

Specifies the user "account" (in the Tenex sense), if required. If this is not specified, the user is logged in under his default account (assuming there is one).

(Connect-Name *directory-name*)

Specifies that the user is to be "connected" (in the Tenex sense) to the specified directory, i.e., to be given temporary owner-like access to that directory. The default connected directory is the user's login directory.

(Connect-Password *password*)

Supplies the password for connecting to a directory, if required.

(Creation-Date *date-and-time*)
(Write-Date *date-and-time*)
(Read-Date *date-and-time*)

Supplies the respective date property of the file. The syntax of *date-and-time* is "dd-mmm-yy hh:mm:ss *zzz*", where *zzz* is a standard North American time zone (e.g., PST, EDT), GMT, or a sign followed by a numeric time zone (e.g., +3 or +3:00, meaning 3 hours west of Greenwich). Inclusion of the zone is mandatory for all new implementations; however, for compatibility with older programs, all implementations must be able to consume date properties with or without time zones.

The precise semantics of the date properties are file system specific; however, adherence to the "Alto File Date Standard" (<AltoDocs>FileDates.press) is strongly encouraged. In particular, the Creation-Date is defined to be the date on which the *information* in a file was created; when a file's contents are copied without change, its creation date should be copied with it.

(Author *user-name*)

Supplies the name (e.g., Grapevine registered name) of the user who created the file.

(Checksum *decimal-number*)

Supplies an optional checksum over the file. The checksum is computed over the file's data in the form in which it flows over the BSP stream, but treating it as a sequence of 16-bit words (with the left byte of each word first, and with the leftover byte assumed to be zero if the file consists of an odd number of bytes). It is a 16-bit number computed by applying the Pup checksum algorithm over all words of the file and then adding 1 (modulo 2^{16}). This means the "no checksum" value is zero rather than 177777B. Sending a value of zero is equivalent to not sending a Checksum property at all.

The protocol does not require the recipient to check the checksum, nor does it specify what action to take if the checksum is found to be incorrect. In general, producers and consumers of files will (optionally) generate and check Checksum properties. File servers, on the other hand, will simply store the property along with the file and give it back when the file is retrieved, and will neither generate nor check Checksum properties.

(Desired-Property *property-name*)

In a command from user to server, requests that the server supply the specified property in all file property lists it generates in response to that command. (That is, *property-name* should be one of Author, Byte-Size, Checksum, Creation-Date, Device, Directory, Name-body, Read-Date, Server-Filename, Size, Type, Version, or Write-Date.)

If no Desired-Property property is present in the property list of a file-related command sent from user to server, the server must send all known properties of each file enumerated. But if one or more Desired-Property properties are present, the server may send only the specified properties and omit the rest. The server may choose to ignore Desired-Property requests altogether and instead send all known properties; it may also ignore Desired-Property requests for properties it cannot send or doesn't know about.

5.3. Defined commands

5.3.1. Basic FTP command messages. Commands (and responses) are given by means of Mark bytes of specified types, followed (in some cases) by a single data byte containing a machine-readable code, followed (in most cases) by a property list or a human-readable string. The command terminates at the next Mark.

In the descriptions below, the mnemonic for the Mark is given in square brackets, followed by *code* if a machine-readable code is used in that command, followed by a description of the data that follows. Mark types are given in decimal. Correct command/reply sequences will be given in the next section.

[Retrieve] *property-list* [Mark type 1]

Issued by the FTP user, requests that the server send to the user the file(s) described by the supplied property list.

[Store] *property-list* [Mark type 2]

Issued by the FTP user, requests that the server prepare to accept the file described by the property list. *This command is obsolete, having been supplanted by [New-Store]; however, for compatibility with older software, FTP user implementations should be prepared for [New-Store] to fail and to retry the operation using [Store].*

[New-Store] *property-list* [Mark type 9]

Issued by the FTP user, requests that the server prepare to accept the file described by the property list. (The forms of the [Store] and [New-Store] commands are the same; the distinction lies in the form of the server's response.)

[Yes] *code human-readable-string* [Mark type 3]

A positive acknowledgment that some previous action has completed successfully. This may be generated by either the server or the user as appropriate (see below). The *code* may supply additional machine-readable information of possible interest to the receiver, or it may be zero; however, the meaning of a Yes in terms of the FTP command/response sequence is determined entirely by context and not by the *code*.

[No] *code human-readable-string* [Mark type 4]

A negative acknowledgment for some previous action. This may be generated by either the server or the user as appropriate (see below). The *code* may supply additional machine-readable information of possible interest to the receiver, or it may be zero; however, the meaning of a No in terms of the FTP command/response sequence is determined entirely by context and not by the *code*.

[Here-is-Property-List] *property-list* [Mark type 11]

Supplies the file property list in the server-to-user direction (in Retrieve, New-Store, Enumerate, and New-Enumerate commands). This command is not used in the user-to-server direction since the property list is specified directly in the commands used to initiate operations.

[Here-is-File] *file-data* [Mark type 5]

Effects the actual file transfer in either direction. The *file-data* physically consists of a stream of 8-bit bytes, but its logical interpretation is dependent on the Type and Byte-Size properties currently in effect.

[Version] *code human-readable-string* [Mark type 8]

Identifies the sender's protocol version. *Code* denotes the protocol version, which is currently 1, and *human-readable-string* is arbitrary identifying text. The FTP user should issue this command immediately upon opening a connection, and the server should respond with a Version reply. It is the responsibility of the user to check for compatible protocol versions.

[Comment] *human-readable-string* [Mark type 7]

Used to supply commentary, indicate non-fatal errors, etc. It is a no-op with respect to protocol interactions and need not be acknowledged. The FTP user program should probably display for the human user any such comments it receives from the server.

[End-of-Command] [Mark type 6]

This command (hereafter abbreviated EOC) informs the receiver that the sender has generated a complete command or command sequence and now cannot proceed until some response is returned. In other words, EOC is used to "give control" to the other party. In general, each command issued by the user should be followed by EOC, and the end of the server's response to a command-EOC sequence should in turn be marked by EOC. There is no data following the EOC: it is followed immediately in the data stream (though perhaps after an arbitrary interval of time) by another Mark.

5.3.2. Optional FTP command messages

[Enumerate] *property-list* [Mark type 10]

Issued by the FTP user, requests the server to generate a complete property list for each of the files denoted by *property-list*. If the file name submitted is interpreted by the server to denote multiple files, the server may generate multiple property lists in response (each preceded by a [Here-is-Property-List] Mark byte). *This command is obsolete, having been supplanted by [New-Enumerate]; however, for compatibility with older software, FTP user implementations should be prepared for [New-Enumerate] to fail and to retry the operation using [Enumerate].*

[New-Enumerate] *property-list* [Mark type 12]

Issued by the FTP user, requests the server to generate a complete property list for each of the files denoted by *property-list*. If the file name submitted is interpreted by the server to denote

multiple files, the server may generate multiple property lists in response (concatenated together in a single [Here-is-Property-List] message). (The forms of the [Enumerate] and [New-Enumerate] commands are the same; the distinction lies in the form of the server's response.)

[Delete] *property-list* [Mark type 14]

Issued by the FTP user, requests the server to delete the specified file(s).

[Rename] *old-property-list new-property-list* [Mark type 15]

Issued by the FTP user, asking the server to rename the file specified in the first property list, to match the name (and perhaps other properties) specified in the second.

5.4. Command-Response sequences

The complete user-server protocols for each command are documented in the flowcharts attached to this specification. Commands from user to server are prefixed by "U:"; from server to user by "S:". Commentary is included in *italics.*

These flowcharts describe only the protocols themselves, and do not show any local actions that the two parties must take. Such actions are necessarily file system or application dependent.

Each sequence begins with the FTP user in control, and ends with the user again in control. A party sends messages only when it is in control. Sending [EOC] switches control to the other party. Where a branch occurs in the flowchart, the party then in control may follow *either* branch and send the message shown at the end of that branch. The normal flow of control is straight down from the top (and looping back up at the left in those protocols that contain loops); errors and other exceptions branch off to the right.

At any point in the protocol, the party then in control may insert a [Comment] message immediately preceding its next command; such a message does not alter the flow of control. Apart from Comments, an occurrence of a command-response sequence other than those shown in the flowcharts is a violation of the protocol and is grounds for aborting the connection.

The command at the beginning of every sequence includes a property list designating some file (or potential file) on the server. The file may be described by Server-Filename, Device, Directory, Name-Body, and Version properties in any combination permitted by the server (as was discussed in section 5.2.1). The property list must include not only filename information but also any credentials required to gain access to the server (User-Name, User-Password, etc.), and also any additional properties intended to modify the server's subsequent actions (Desired-Property).

5.4.1. Retrieve and Delete. These two commands have similar protocols. Each is initiated with a property list designating one or more existing files on the server. If no such file exists, or the command is otherwise unacceptable (e.g., invalid User-Name and User-Password), then the server rejects the entire command with a [No] and the command terminates with the user again in control.

Otherwise, the remainder of the protocol is executed for each file matching the initial request. The main loop is not left until all such files have been enumerated; any error that occurs applies only to an individual file.

The server returns a [Here-is-Property-List] message describing as completely as possible the specific file under consideration. Unless otherwise specified by the user (by Desired-Property properties), the Server-Filename, Name-Body, Type, and (if Type is Binary) Byte-Size properties are mandatory, as are Directory and Version if the server file system has such things. The server is strongly encouraged to include all other known file properties that are representable as FTP properties.

At this point, the FTP user process examines the server's property list (possibly interacting with a human user), and then commands the server either to carry out the operation (Retrieve or Delete) or to skip to the next file.

In the case of Retrieve, the server now sends the contents of the file, followed by a [Yes] message to mark the end of data. There are two possible points of failure. If the server cannot begin transmission (e.g., because the file is read-protected against the user), then it sends a [No] message in response to the user's [Yes]. If the server detects a failure in mid-transmission, it sends a [No] following the (possibly truncated) file data. (A user-detected transmission failure results only in user-local recovery actions and requires no provisions in the protocol.)

In any event, the server is now in control (since its last [Yes] or [No] was not terminated by [EOC]), and proceeds either to send a [Here-is-Property-List] message for the next file or to send an [EOC] to terminate the Retrieve protocol.

5.4.2. Store and New-Store. These commands are initiated with a property list designating some file (existing or new) into which the user desires to store data. This property list should be as complete as possible; in particular, the Type and (if Type is Binary) Byte-Size properties are mandatory. The user is strongly encouraged to include all other known file properties that are representable as FTP properties.

If the file cannot be stored into (e.g., write protected, no disk space available, etc.), or the command is otherwise unacceptable (e.g., invalid User-Name and User-Password), then the server rejects the entire command with a [No] and the command terminates with the user again in control.

In the case of [New-Store], the server now returns as complete a property list as is possible before the file data has actually been transferred. Unless otherwise specified by the user (by Desired-Property properties), this must include the Server-Filename and all filename properties implemented by the server. The user may now examine this property list and choose not to store the file after all.

In the normal situation, the user now sends the contents of the file, followed by a [Yes] message to mark the end of data. If the file is stored successfully, the server responds with a [Yes], and the Store or New-Store protocol terminates.

There are two possible points of failure. If the user detects a failure in mid-transmission, it sends a [No] following the (possibly truncated) file data. If the server detects a failure in mid-transmission, it consumes and discards the remainder of the file data and then responds with [No] to indicate failure.

5.4.3. Enumerate and New-Enumerate. These commands are initiated with a property list designating one or more existing files on the server. If no such file exists, or the command is otherwise unacceptable (e.g., invalid User-Name and User-Password), then the server rejects the entire command with a [No] and the command terminates with the user again in control.

Otherwise, the server sends a property list for each file matching the initial request. In [Enumerate], each property list is sent as a separate [Here-is-Property-List] message, whereas in [New-Enumerate] all the property lists are concatenated and sent as a single [Here-is-Property-List] message.

Unless otherwise specified by the user (by Desired-Property properties), these property lists should be as complete as possible. In particular, the Server-Filename, Name-Body, Type, and (if Type is Binary) Byte-Size properties are mandatory, as are Directory and Version if the server file system has such things. The server is strongly encouraged to include all other known file properties that are representable as FTP properties.

5.4.4. Rename. This command is initiated with two property lists designating an existing file and a new file on the server. Any required non-file-related properties (e.g., User-Name) must be present in the first property list but need not be in the second. The server's response is straightforward: either it carries out the request and responds [Yes] or it responds [No].

If the second property list contains any file properties in addition to the new file name, the server may change those properties to the specified values. (This is implemented in IFS release 1.38 but in no other FTP servers at the time of this writing.) The second file name may also be the same as the first; this is useful both for changing file properties and for changing the capitalization of a file name.

5.5. *Yes and No reply code assignments*

A machine-readable argument may optionally be supplied with Yes and No replies. Zero should be supplied in the absence of any such code. The intent of these codes is to facilitate mechanical handling of certain exceptional conditions; however, both generation and interpretation of reply codes is entirely optional, and the command/reply sequence is unaffected by them. The inclusion of informative human-readable strings is strongly encouraged.

Codes have not been assigned to Yes replies, so Yes code bytes should always be zero. Assigned No code bytes are listed in the appendix.

5.6. *Examples*

These are examples of normal cases of the New-Store, Delete, Retrieve, and New-Enumerate commands.

U: [Version] <1> BCPL Pup FTP User, 14 May 82
U: [End-of-Command]
S: [Version] <1> PARC Ivy IFS 1.35.4L, File Server of May 11, 1982; 2 users out of 9
S: [End-of-Command]
U: [New-Store] ((Creation-date 14-May-82 15:26:45 PDT) (Desired-property Server-filename) (End-of-line-convention CR) (Read-date 25-May-82 14:08:01 PDT) (Server-filename User.cm) (Size 2113) (Type Text) (User-name Taft.PA) (User-password xxxxxx) (Write-date 14-May-82 15:26:45 PDT))
U: [End-of-Command]
S: [Here-is-Property-List] ((Server-filename <Taft>User.cm!1))
S: [End-of-Command]
U: [Here-is-File] .....
U: [Yes] <0> Transfer Complete
U: [End-of-Command]
S: [Yes] <0> Store completed
S: [End-of-Command]
U: [Delete] ((Desired-property Server-filename) (Desired-property Name-body) (Server-filename User.cm) (User-name Taft.PA) (User-password xxxxxx))
U: [End-of-Command]
S: [Here-is-Property-List] ((Name-body User.cm) (Server-filename <Taft>User.cm!1))
S: [End-of-Command]
U: [Yes] <0> Please delete that file
U: [End-of-Command]
S: [Yes] <0> File Deleted
S: [End-of-Command]
U: [Retrieve] ((Desired-property Server-filename) (Desired-property Name-body) (Desired-property Type) (Desired-property Byte-size) (Desired-property Creation-date) (Directory Mesa>FTP) (Server-filename FTPDefs.*) (User-name Taft.PA) (User-password xxxxxx))
U: [End-of-Command]
S: [Here-is-Property-List] ((Byte-size 8) (Creation-date 4-Sep-80 22:14:08 PDT) (Name-body FTPDefs.bcd) (Server-filename <Mesa>FTP>FTPDefs.bcd!1) (Type Binary))
S: [End-of-Command]
U: [No] <0> No thanks
U: [End-of-Command]
S: [Here-is-Property-List] ((Creation-date 28-Jul-80 20:38:58 PDT) (End-of-line-convention CR) (Name-body FTPDefs.mesa) (Server-filename <Mesa>FTP>FTPDefs.mesa!1) (Type Text))
S: [End-of-Command]
U: [Yes] <0> File open, ready for data
U: [End-of-Command]
S: [Here-is-File] .....
S: [Yes] <0> Transfer complete
S: [End-of-Command]
U: [New-Enumerate] ((Desired-property Server-filename) (Directory Mesa>FTP) (Server-filename FTPUser.*) (User-name

Taft.PA) (User-password xxxxxx))
U: [End-of-Command]
S: [Here-is-Property-List] ((Server-filename <Mesa>FTP>FTPUser.bcd!1)) ((Server-filename <Mesa>FTP>FTPUser.config!1))
S: [End-of-Command]
U: [New-Enumerate] ((Directory Mesa>FTP) (Server-filename FTPUser.*) (User-name Taft.PA) (User-password xxxxxx))
U: [End-of-Command]
S: [Here-is-Property-List] ((Author Johnsson) (Byte-size 8) (Creation-date 13-Oct-80 20:16:58 PDT) (Device Primary) (Directory Mesa>FTP) (Name-body FTPUser bcd) (Read-date 14-Apr-82 14:49:11 PST) (Server-filename <Mesa>FTP>FTPUser.bcd!1) (Size 25600) (Type Binary) (Version 1) (Write-date 30-Oct-80 5:30:03 PST)) ((Author Johnsson) (Creation-date 3-Aug-80 18:14:36 PDT) (Device Primary) (Directory Mesa>FTP) (Name-body FTPUser config) (Read-date 21-Mar-82 20:05:33 PST) (Server-filename <Mesa>FTP>FTPUser.config!1) (Size 973) (Type Text) (Version 1) (Write-date 30-Oct-80 5:30:32 PST))
S: [End-of-Command]

6. File formats and conversion

When a file property list is sent to a server as part of an FTP command, it describes the file as the user would like to have it moved. The user may be willing to do some subsequent conversions, or may ask the server to perform the conversion.

For example, if a user wants to retrieve a text file from a server, and have the server convert the file to End-of-Line-Convention CRLF, then the user might send the command:

[Retrieve]((Server-Filename Foo.text)(Type Text)(End-of-Line-Convention CRLF)...)

The server might respond with a [Here-is-Property-List], indicating willingness to provide the file in that form; or it might refuse to do the conversion. Similarly, if a user storing a file wants to transmit it with End-of-Line-Convention CRLF, and have the server convert the file (if necessary) to a different internal convention, then the user might send the command:

[New-Store]((Server-Filename Foo.text)(Type Text)(End-of-Line-Convention CRLF)...)

The server might respond with a [Here-is-Property-List], indicating willingness to accept the file in that form; or it might refuse to do the conversion.

It is expected that all implementations of the FTP should at least be able to store and retrieve ASCII text files according to one standard convention:

(...(Type Text)(End-of-Line-Convention CR)..)

and to store and retrieve 8-bit Binary files:

(...(Type Binary)(Byte-Size 8)..)

7. Interrupts, restarts, and aborts

Either process may, if necessary, use the BSP convention for interrupting a byte stream: place an [Abort] Mark Byte in the stream at the current position, and then signal a BSP Interrupt. This will clear out all pending data, and re-establish control at the top level of the protocol.

A process receiving an interrupt can then look for the corresponding Mark Byte in the incoming byte stream, but must also indicate the reset position in its outgoing byte stream, so that the two may regain synchronization. Thus, upon receiving a BSP Interrupt a process must signal a corresponding BSP Interrupt-reply, and drop an [Abort] Mark Byte into the stream.

The details of this mechanism have not been resolved, and the Abort has not generally been implemented.

Remember that the protocol relies upon the BSP for establishing, terminating, or aborting all connections.

8. Other remarks

1. Note that most implementations of the BSP provide buffering of the outgoing bytes, so it may be necessary to poke the Byte Stream in an appropriate manner in order to send a complete command.

2. All BSP connections are full-duplex; however, the FTP protocol constrains the user and server to take turns transmitting. To avoid deadlocks, each party must wait until it receives an [EOC] from the other party before beginning to transmit. In exceptional situations, this may require the party not in control to consume and discard data until an [EOC] is received.

3. In future programs, a user process might request a transfer of only part of a file:

((Name BACKUP.T)(Starting-position 2000)(Number-of-bytes 512)...)

The first byte of a file is always byte 0. Issues such as positioning backwards, leaving files open for efficiency, etc. are left to the program, and are not part of the protocol.

4. In general, third party transfers are not supported. There are several situations which we believe can be handled adequately within the current protocol:

a. A user asks a server for a file, but the server knows that the file is actually backed up on an archive facility. The server can then 1) invisibly fetch the file from the archive, and then transmit it to the user; or 2) send a [No] to the user indicating that the file lives elsewhere, allowing the user to ask for the full property list (including the file's actual location) and then establish a new connection to the archive. In either case, the server will not try to magically force a connection between the user and the archive.

b. A user wants to take a file from the server and print it on a different printing facility. The user can then 1) fetch the file from the server, and then send it to the printer; or 2) ask the server to send the file to the printer in an appropriate way.

5. There are additional commands to another file system that are not currently part of the protocol, but which might prove useful:

[Create] *property-list*

[Append] *property-list property-list*

[Reset-properties] *property-list property-list*

[Copy] *property-list property-list*

9. FTP and the Pup Mail Transfer Protocol

The File Transfer Protocol has also been used as the foundation for a Pup-based Mail Transfer Protocol; the FTP command/response framework has been augmented with a set of commands and responses to manipulate mailboxes (i.e., message files). The Mail-related commands and error codes are disjoint from the ones selected for the FTP; care must be taken, however, to ensure that any new FTP or Mail codes are not already used by the companion protocol. For further information, see Ed Taft's memo cited in section 1.

10. Revision history

September 29, 1983:
—[Rename] may change file properties besides the name.

September 12, 1983:
—Corrected error in flowcharts for [Store] and [New-Store].

May 25, 1982:
—Renamed [Directory] command to [Enumerate] to avoid confusion with Directory property.
—Removed [You-Are-User] and added [New-Enumerate] command.
—Added Checksum and Desired-Property properties.
—Drew complete protocol flowcharts as a substitute for the numerous examples formerly included to document the command-response sequences; added some actual examples.
—Made all numbers decimal.
—Made extensive minor edits, corrections, clarifications, etc.

July 15, 1978:
—Changed to press format.
—Extraneous [No] removed from [Delete] specification.

April 15, 1978:
—Added commands to [Delete] and [Rename] files at the server.
—Added [NewStore] command, which replies not with [No]/[Yes], but with [No]/[Here-is-Property-List] (thus providing the fully qualified name under which the file is stored).
—New error codes for file access: 107, 110, 111.
—Added section 9 on FTP and the Pup Mail Transfer Protocol.
—Discussion of timeouts.
—Other minor corrections.

June 15, 1976:
—Optional file property (Size <decimal number>) incorporated; generated by Maxc in response to a [Directory] command.
—Multiple responses to [Directory] command provided, with each property list in the response preceded by [Here-is-Property-List].
—A few other small typographical errors have been corrected.

May 19, 1976:
—A major re-write, incorporating detailed examples of command/response sequences.

Appendix I: Registry of FTP mark byte commands

Command	*Value (decimal)*
[Retrieve]	1
[Store]	2
[Yes]	3
[No]	4
[Here-is-the-file]	5
[End-of-Command]	6
[Comment]	7
[Version]	8
[New-Store]	9
[Enumerate]	10
[Here-is-Property-List]	11
[New-Enumerate]	12
[Delete]	14
[Rename]	15

Mark bytes 16-31 are reserved for use in the Mail Transfer Protocol, including:

[Store-Mail]	16
[Retrieve-Mail]	17
[Flush-Mailbox]	18
[Mailbox-Exception]	19

Unimplemented commands:

[Abort]
[Reset]
[Reset-Reply]
[Create]
[Append]
[Reset-Properties]
[Copy]

Appendix II: Reply codes within [Yes] and [No]

A machine-readable argument may optionally be supplied with [Yes] and [No] replies. Zero should be supplied in the absence of any such code. The intent of these codes is to facilitate mechanical handling of certain exceptional conditions; however, both generation and interpretation of reply codes is entirely optional, and the command/reply sequence is unaffected by them. The inclusion of informative human-readable strings is strongly encouraged.

Codes have not been assigned to [Yes] replies, so [Yes] code bytes should always be zero. Assigned No code bytes are as follows (all numbers decimal).

General:

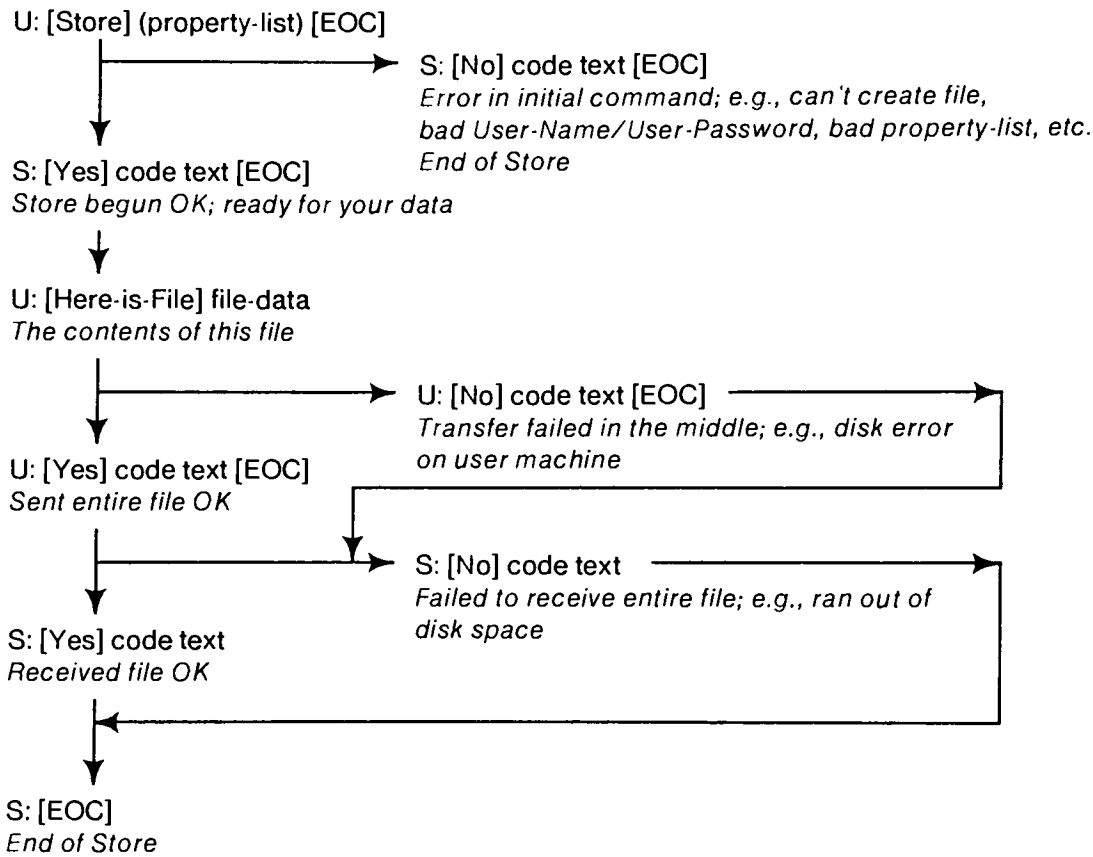
1 Last command undefined or unimplemented.
2 Command requires User-Name to be supplied, and it wasn't.
3 Last command illegal in present context.

Property list errors:

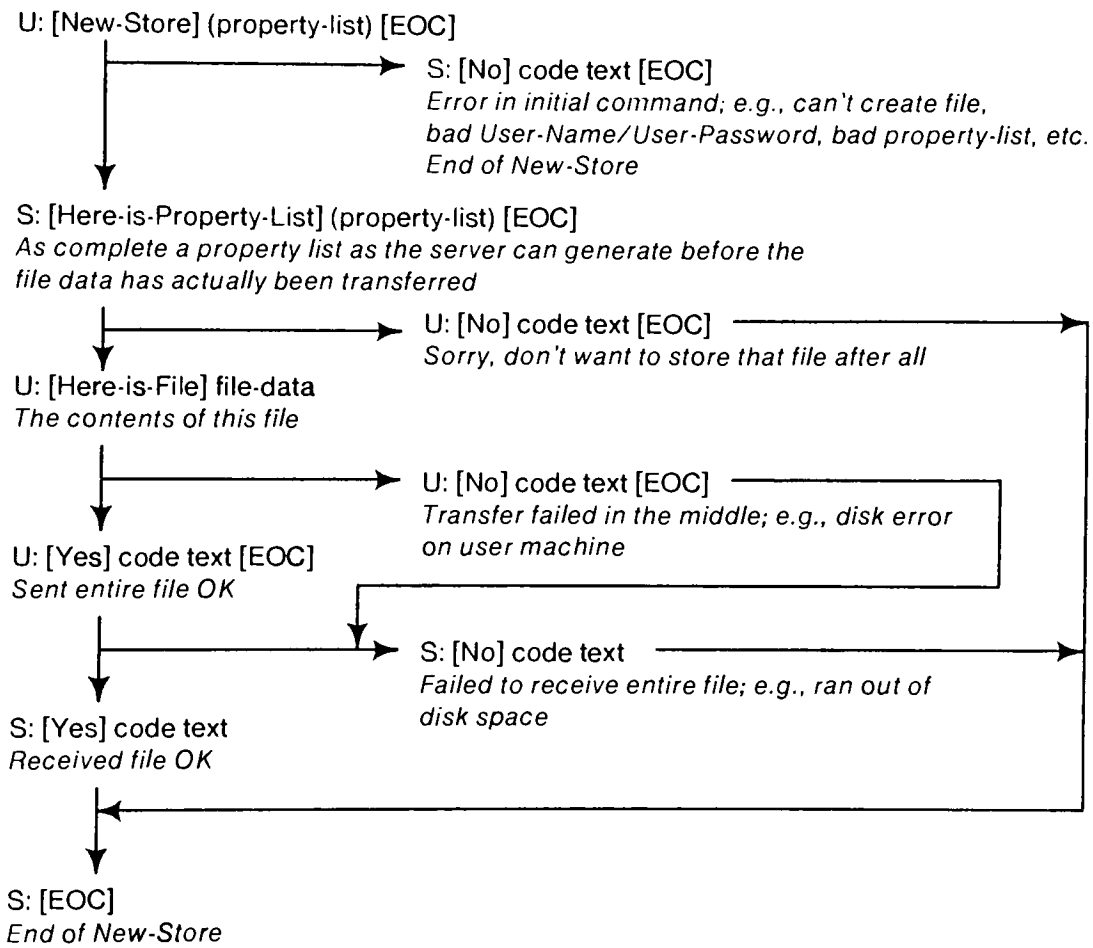
8 Malformed property list.
9 Illegal Server-Filename.
10 Illegal Directory.
11 Illegal Name-Body.
12 Illegal Version.
13 Illegal Type.
14 Illegal Byte-Size.
15 Illegal End-of-Line-Convention.
16 Illegal User-Name.
17 Illegal or incorrect User-Password.
18 Illegal or incorrect User-Account.
19 Illegal Connect-Name.
20 Illegal or incorrect Connect-Password.
21 Illegal Creation-Date.
22 Illegal Write-Date.
23 Illegal Read-Date.
24 Illegal Author.
25 Illegal Device.

Specific to file-access commands:
64 File not found
65 Requested access denied to file
66 Transfer parameters inconsistent with file parameters
67 File data error
68 File too long or storage full
69 Do not send the file (user to server during Retrieve)
70 Store not completed (due to No from user)
71 Transient, non-specific server or file-system failure
72 Permanent, non-specific server or file-system failure
73 File busy
74 Rename destination file already exists

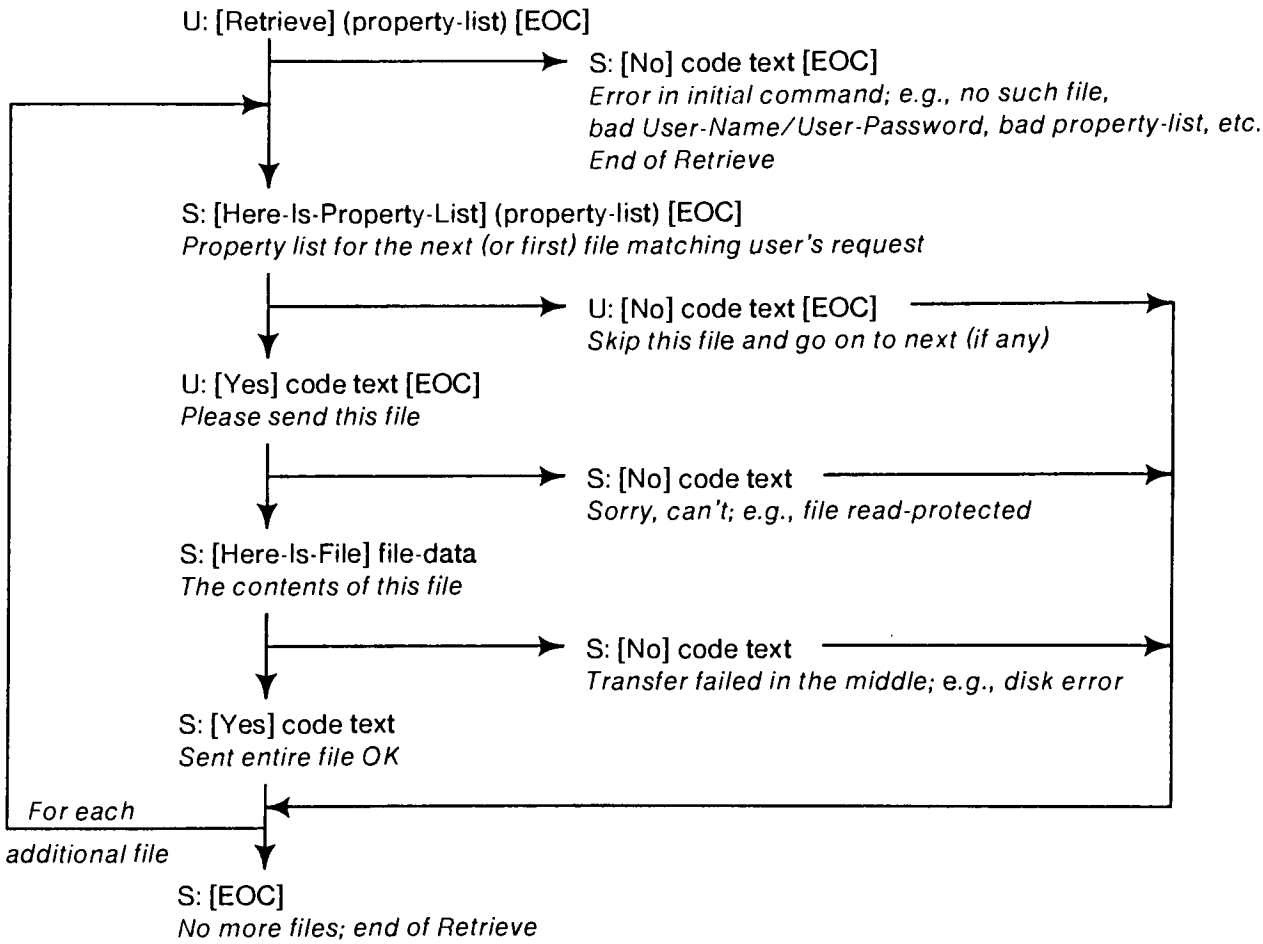
Specific to the Mail Transfer Protocol:
32 No valid Mailbox in property list
33 Illegal Mailbox property syntax
34 Illegal Sender property



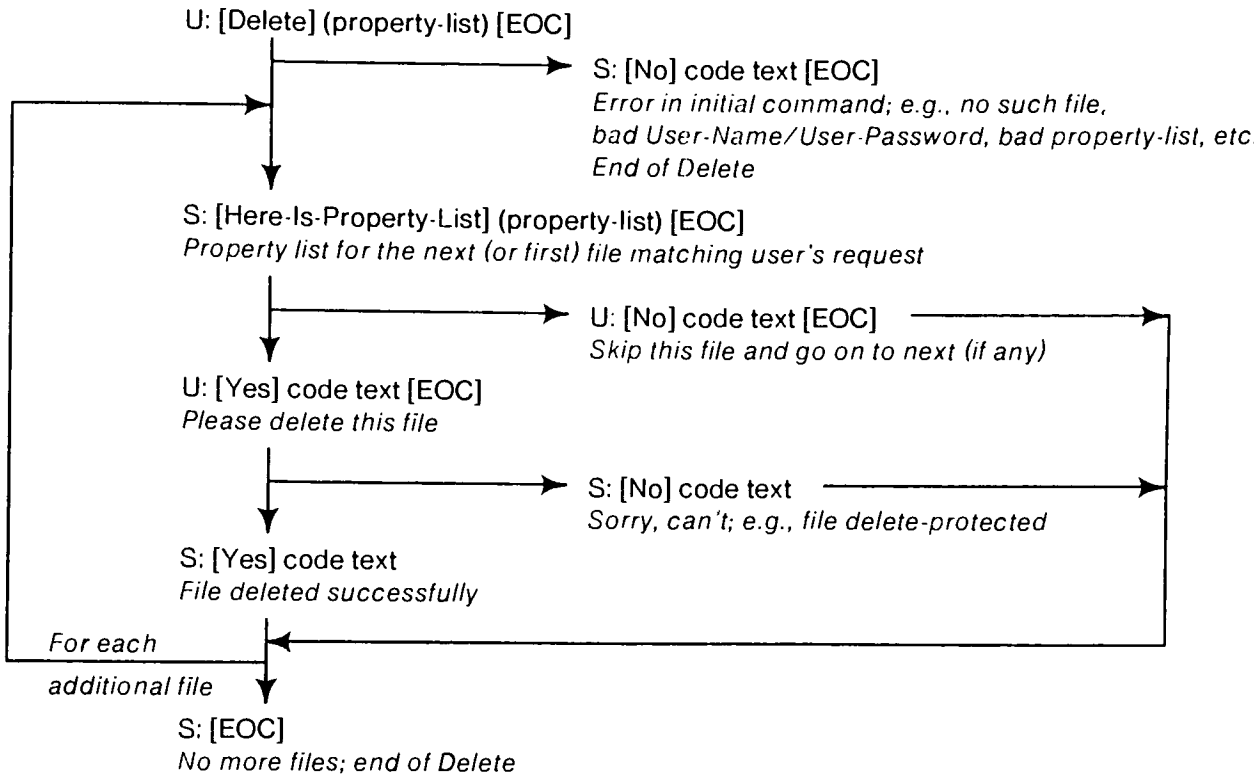
FTP [Store] Protocol



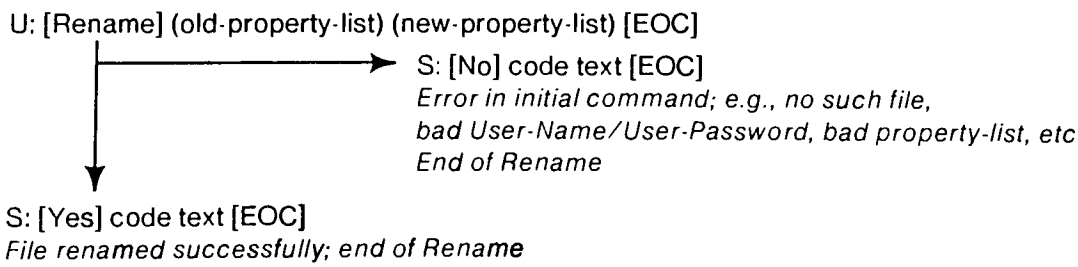
FTP [New-Store] Protocol



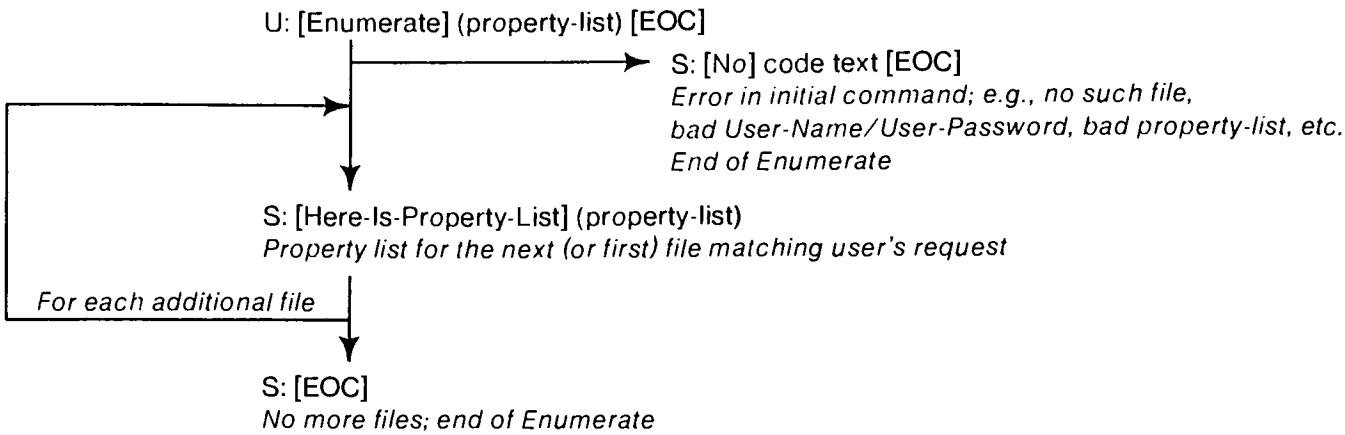
FTP [Retrieve] Protocol



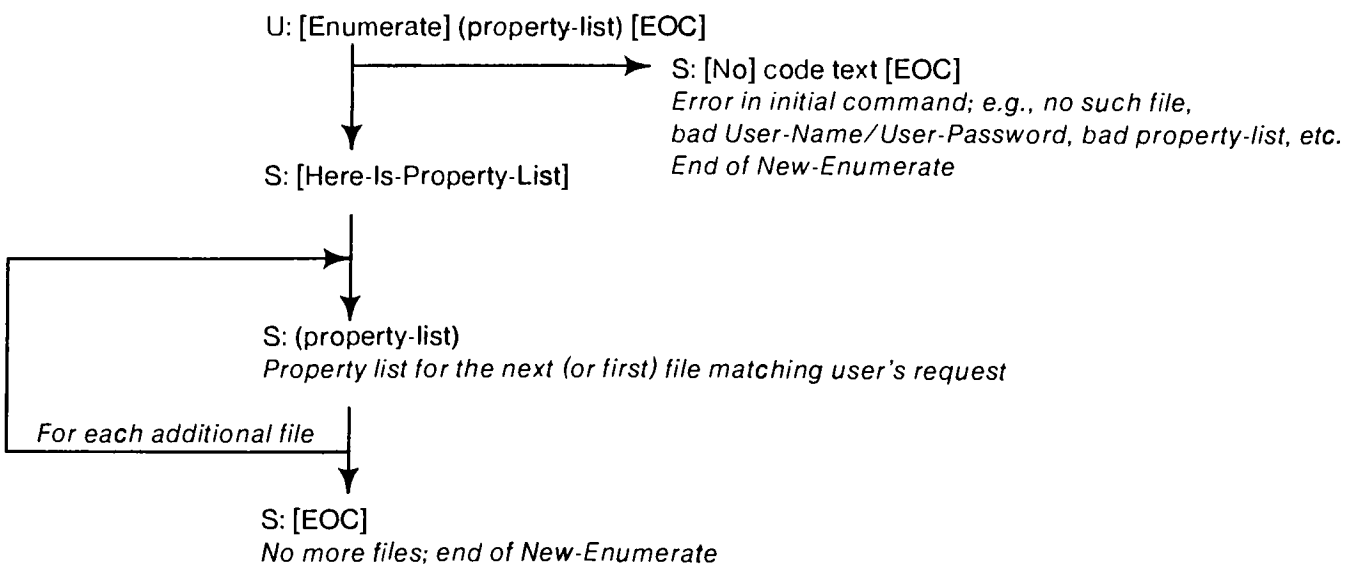
FTP [Delete] Protocol



FTP [Rename] Protocol



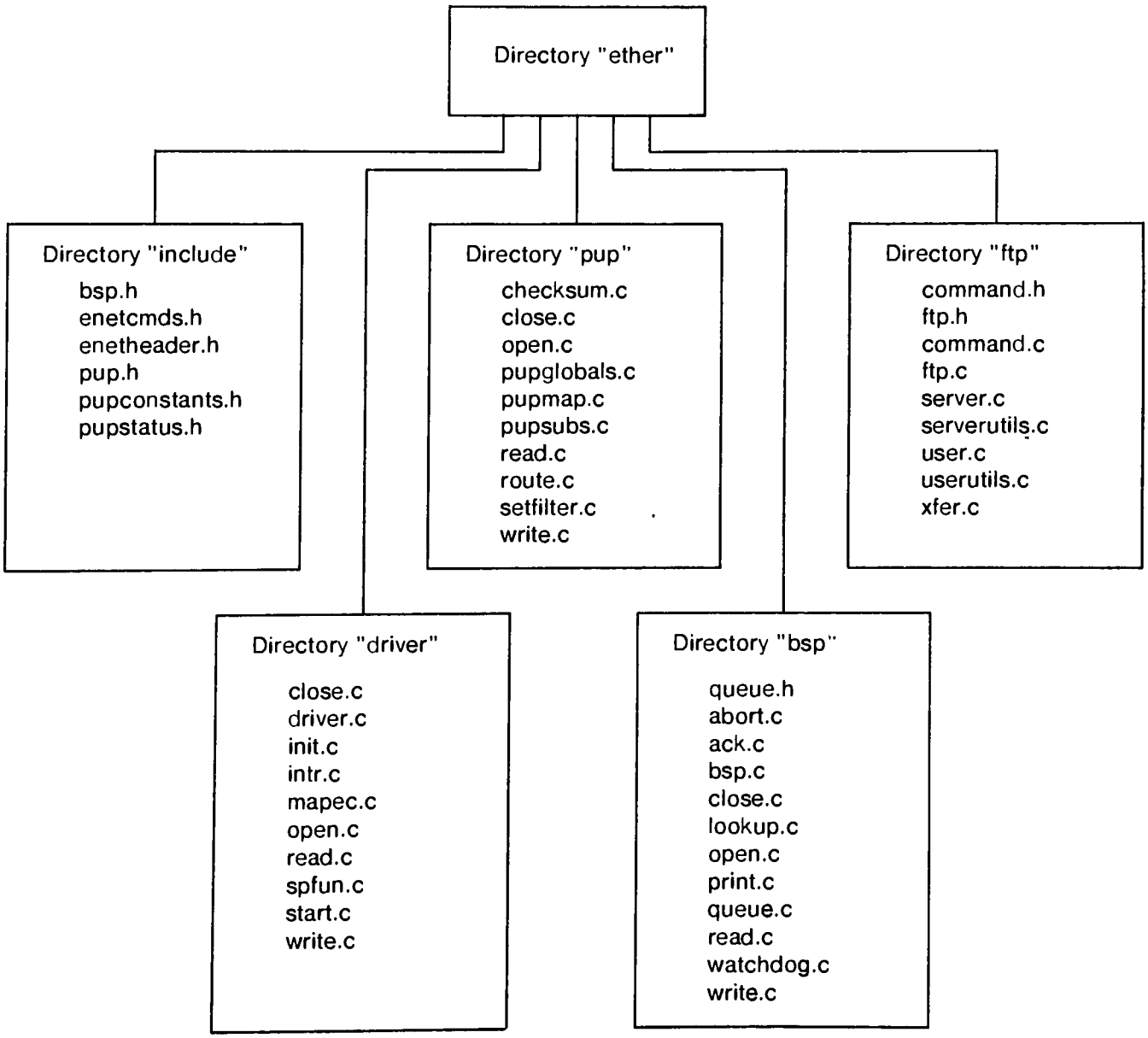
FTP [Enumerate] Protocol



FTP [New-Enumerate] Protocol

Appendix C: Program Listings

The program listings are divided into five sections; these are (1) the include files, (2) the Ethernet driver, (3) the PUP library, (4) the BSP library, and (5) the FTP program listings. The code was arranged in directories as shown below.



```
/*
        File:  include/bsp.h
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Definitions for Byte Stream Protocol

        MODIFIED BY     REASON
*/

#include "../include/pup.h"

#define Initiate        1               /* start a user connection */
#define Listen          2               /* wait for requests for connection */
#define Server          3               /* server process spawned for user */

#define OFF             0               /* stop watchdog timer */
#define ON              1               /* start watchdog timer */

extern int BSPdebug;
extern PORT LocalPort;
extern PORT RemotePort;
```

```
/*
        File:  include/enetcmds.h
        Date:  January 11 1984
        Author:  Mark Van Dellon
        Purpose:  Commands and status for NI3010A ethernet controller.

        MODIFIED BY     REASON
*/

#ifndef _ENETCMDS

/* Define bases for seperating kernal and user areas */

#define BASEUNOS                0       /* special unos codes */
#define BASEUSER                0x100   /* ethernet codes of user */
#define BASESU                  0x200   /* ethernet codes of kernal */
#define BASEERR                 0x400   /* error codes base */

#define SUCCESS                 0       /* no error on wharever */

/* NI3010A commands - all require superuser priviledge */

#define SetModuleLpbk           (BASESU | 0x01) /* module interface lpbk */
#define SetInternalLpbk         (BASESU | 0x02) /* set internal loopback */
#define ClrLpbk                 (BASESU | 0x03) /* clear loopback */
#define SetPromiscuous          (BASESU | 0x04) /* set promiscuous mode */
#define ClrPromiscuous          (BASESU | 0x05) /* clear promiscuous mode */
#define SetReceiveOnErr         (BASESU | 0x06) /* set receive-on-error */
#define ClrReceiveOnErr         (BASESU | 0x07) /* clear receive-on-error */
#define Offline                 (BASESU | 0x08) /* go offline */
#define Online                  (BASESU | 0x09) /* go online */
#define RunDiagnostics          (BASESU | 0x0a) /* run onboard diagnostics */
#define SetInsertSrcAddr        (BASESU | 0x0d) /* set insert source addr */
#define ClrInsertSrcAddr        (BASESU | 0x0e) /* clear insert source addr */
#define SetPhyAddrDefault       (BASESU | 0x0f) /* ethernet addr to default */
#define SetRecMulticast         (BASESU | 0x10) /* set rec multicast packet */
#define ClrRecMulticast         (BASESU | 0x11) /* clr rec multicast packet */
#define NetworkLpbkTest         (BASESU | 0x12) /* do network lpbk test */
#define CollisionTest           (BASESU | 0x13) /* do collision detect test */
#define ReportStatus            (BASESU | 0x18) /* report and reset stats */
#define ReportCollision         (BASESU | 0x19) /* report collision delays */
#define LoadTransData           (BASESU | 0x28) /* load transmit data */
#define LoadTransDataSend       (BASESU | 0x29) /* load x-mit data & send */
#define LoadGroupAddr           (BASESU | 0x2a) /* load group address */
#define DeleteGroupAddr         (BASESU | 0x2b) /* delete group address */
#define LoadPhyAddr             (BASESU | 0x2c) /* load physical address */
#define Reset                   (BASESU | 0x3f) /* reset */

#define IOCMD(X)        (X & ~BASESU)           /* translate bd cmds */

/* NI3010A command status codes */

#define CMDOK           (BASEERR | 0x00)
#define CMDOK_RETRIES   (BASEERR | 0x01)        /* success with retries */
#define ILLEGAL         (BASEERR | 0x02)        /* illeagal command */
#define INAPPROPRIATE   (BASEERR | 0x03)        /* inappropriate command */
#define FAILURE         (BASEERR | 0x04)
```

```
#define BUFFER_SMALL    (BASEERR | 0x05)        /* buffer size exceeded */
#define FRAME_SMALL     (BASEERR | 0x06)        /* frame too small */
#define COLLISIONS      (BASEERR | 0x08)        /* excessive collisions */
#define BUFFER_ALIGN    (BASEERR | 0x0a)        /* buffer allignment error */
#define NO_HEARTBEAT    (BASEERR | 0x0b)        /* no heartbeat detected */
#define NO_CRC          (BASEERR | 0x0c)        /* no CRC error occurred */
#define BAD_CRC         (BASEERR | 0x0d)        /* inappropriate CRC error */
#define BAD_LASTBYTE    (BASEERR | 0x0e)        /* last data byte not
                                                   received correctly */

/* NI3010A diagnostic status codes */

#define CMDOK           (BASEERR | 0x00)
#define E_NM10A_ROM     (BASEERR | 0x01)        /* ROM error */
#define E_NM10A_RAM     (BASEERR | 0x02)        /* RAM error */
#define E_ADDRESS       (BASEERR | 0x03)        /* address error */
#define E_LOOPBACK      (BASEERR | 0x04)        /* loopback error */
#define E_NOCARRIER     (BASEERR | 0x05)        /* carrier sense failure */

#define IOST(X)         (X & ~BASEERR)          /* translate error codes */

/* structure of ReportStatus command - reports and resets statistics */

typedef struct
        {
        unsigned char es_fill1[2];      /* unused */
        unsigned short framlen;         /* # of bytes in this frame */
        unsigned char physadr[6];       /* physical net address */
        unsigned short in_tot;          /* # of frames received */
        unsigned short in_q;            /* # of frames still in receive Q */
        unsigned short out_tot;         /* # of frames transmitted */
        unsigned short out_col_16;      /* # of 16 successive collisions */
        unsigned short in_runt;         /* # of runt packets received */
        unsigned short in_lost;         /* # of frames lost due to no mem */
        unsigned short multi_accept;    /* # of multicast frames accepted */
        unsigned short multi_reject;    /* # of multicast frames no match */
        unsigned short in_crc;          /* # of frames with crc error */
        unsigned short in_align;        /* # of frames with alignment error */
        unsigned short out_col_1;       /* total # of collisions */
        unsigned short out_col_wind;    /* # of out-of-window collisions */
        unsigned char es_fill2[16];     /* unused */
        unsigned char mod_id[8];        /* module id */
        unsigned char firm_id[8];       /* firmware id */
        } ESTAT;

/* structure of ethernet receive frame filter */

typedef struct
        {
        unsigned char f_filler1[16];
        unsigned short f_enetpacket;    /* encapsulated packet type */
        unsigned char f_filler2[3];
        unsigned char f_puptype;        /* pup packet type */
        unsigned char f_filler3[5];
        unsigned char f_dsthost;        /* destination host */
        unsigned long f_dstsocket;      /* destination socket */
        } RFF;
```

```
typedef struct
        {
        struct
                {
                unsigned do_enetpacket:1;       /* encapsulated packet type */
                unsigned do_puptype:1;          /* pup packet type */
                unsigned do_dsthost:1;          /* destination host */
                unsigned do_dstsocket:1;        /* destination socket */
                } fwhich;
        RFF fdata;
        } FILTER;

/* Special function codes for driver/user space communication */

#define GetDriverValues         (BASEUSER | 1)
#define GetMinorQcount          (BASEUSER | 2)
#define SetMinorFilter          (BASEUSER | 3)

/* Structure for GetDriverValues spfun code   (intended for driver debug) */

typedef struct
        {
        unsigned char netaddress[6];    /* 48 bit ethernet address */
        unsigned short numminors;       /* minors driver installed for */
        unsigned short maxrecvq;        /* size of each minor receive Q */
        unsigned short bdvector;        /* board interrupt vector */
        unsigned long bdaddress;        /* board versabus address */
        unsigned short bdilevel;        /* board interrupt level */
        unsigned short bdiwait;         /* current interrupt wanted */
        unsigned short bdstate;         /* current hardware state */
        } DVALUES;

/* Driver eventcount names */

#define LEFTQEC         1               /* packets been read-from-Q ec */
#define PUTQEC          2               /* packets been put-in-Q ec */

#define _ENETCMDS
#endif
```

```
/*
        File:  include/enetheader.h
        Date:  January 11 1984
        Author:  Mark Van Dellon
        Purpose:  Driver information for NI3010A ethernet controller.

        MODIFIED BY     REASON
*/

#ifndef _ENETHEADER

/* board and driver redefinable parameters */

#define ETHERADDR       0xffffe0        /* versabus board address */
#define ETHERINTR       4               /* interrupt priority level */
#define ETHERVEC        0xC0 << 2       /* interrupt vector address */
#define MAXMINORS       8               /* # of minor devices for driver */
#define MAXWAITING      3               /* # in Rx Q allowed by each minor */

/* defines for driver but not redefinable */

#define MAXPACKET       1534            /* # bytes host sees in max packet */

/* NI3010A memory mapped registers */

struct reg_table
        {
        unsigned char c_reg;
        unsigned char s_reg;
        unsigned char t_data;
        unsigned char r_data;
        unsigned char e_skip1;          /* skip in addressing */
        unsigned char is_reg;
        unsigned char e_skip2[2];       /* skip in addressing */
        unsigned char ie_reg;
        unsigned char e_bar;
        unsigned char h_bar;
        unsigned char l_bar;
        unsigned char h_bcr;
        unsigned char l_bcr;
        }

#define C_REG   (ETHERADDR->c_reg)      /* write only command register */
#define S_REG   (ETHERADDR->s_reg)      /* read only status register */
#define T_DATA  (ETHERADDR->t_data)     /* write only transmit data reg */
#define R_DATA  (ETHERADDR->r_data)     /* read only receive data register */
#define IS_REG  (ETHERADDR->is_reg)     /* read only interrupt status reg */
#define IE_REG  (ETHERADDR->ie_reg)     /* write only interrupt enable reg */
#define E_BAR   (ETHERADDR->e_bar)      /* write only extended bus addr reg */
#define H_BAR   (ETHERADDR->h_bar)      /* write only high bus addr reg */
#define L_BAR   (ETHERADDR->l_bar)      /* write only low bus addr reg */
#define H_BCR   (ETHERADDR->h_bcr)      /* write only high byte count reg */
#define L_BCR   (ETHERADDR->l_bcr)      /* write only low byte count reg */

/* NI3010A interrupt status register bit definitions */

#define ISR_SRF         0x01            /* high - status reg is full */
```

```
#define ISR_SBA         0x02            /* low   status block available */
#define ISR_RBA         0x08            /* low   receive block available */
#define ISR_NORMALIZE   0xfa            /* xor to uncomplement above */

/* NI3010A interrupt enable register definitions */

#define IER_DISABLE     0               /* disable interrupts */
#define IER_STATUSREG   1               /* status register full interrupt */
#define IER_STATUSBLK   2               /* status block interrupt */
#define IER_RECEIVE     4               /* receive blk available interrupt */
#define IER_XMITDMA     6               /* interrupt when xmit DMA done */
#define IER_RECVDMA     7               /* interrupt when recv DMA done */

/* Structure and definitions for circular receive queues.  The head of
*  the queue contains the two links.  Each element placed on the queue
*  will be a RQ structure.
*/

typedef char (*QLINK)
typedef struct                          /* head of Q structure */
        {
        QLINK FL;                       /* forward link */
        QLINK BL;                       /* backward link */
        } QHEAD;

typedef struct                          /* element queued structure */
        {
        QLINK nextRQ;                   /* forward & backward links */
        unsigned long Rlen;             /* data length in received packet */
        unsigned char Rpkt[MAXPACKET];  /* received data */
        } RQ;

#define InitQ(QHP)              QHP.FL = NULL;

/* structure for each minor device of the driver */

typedef struct
        {
        unsigned short mstate;  /* flags */
        FILTER enfilt;          /* minor's filter for receive */
        unsigned short rsize;   /* # of packets this minor has waiting */
        QHEAD rwait;            /* queued received packets waiting to be read.
                                 * rsize contains the number of packets
                                 * waiting and rsize will never be larger than
                                 * MAXWAITING. */
        EVENTCOUNT putqec;      /* incremented when packet put in Q */
        EVENTCOUNT leftqec;     /* incremented when packet taken from Q */
        } MINOR;

/* structure for information and tables kept for driver */

typedef struct
        {
        unsigned short hw_intr; /* current hardware interrupt mode */
        unsigned short hw_state;/* current hardware state */
        unsigned char enaddr[6];/* ethernet address */
```

```
        unsigned long last_receive;     /* time of last receive */
        RQ *rcurrent;           /* ptr to struct receiving current packet */
        QHEAD rfree;            /* receive queue list available for packets */

        unsigned long last_transmit;    /* time of last transmit */
        short wstate;           /* current state of write transfer */

        char spfuncode;         /* holds user special function */

        MINOR minor[MAXMINORS]; /* parameters for each minor */

        SYSBUF enet_rbuf;       /* raw I/O buffer */
        SEMAPHORE enet_rsemph;  /* raw semaphore for exclusion on I/O buf */
        } DRIVERDATA;

/* hardware state   for hw_state variable */

#define HW_NOBOARD      00      /* board totally down & dead */
#define HW_BOARD        01      /* board acks DTACK */
#define HW_RESET        02      /* board reset */
#define HW_NORESET      (~HW_RESET)
#define HW_ONLINE       04      /* normal operating state */
#define HW_OFFLINE      (~HW_ONLINE)
#define HW_BUSY         010     /* transaction in progress */
#define HW_IDLE         (~HW_BUSY)

/* minor device flags - for mstate variable */

#define MS_CLOSED       0x00    /* device closed */
#define MS_OPEN         0x01    /* device open */
#define MS_FILTER       0x02    /* device has valid receive filter */

/* write state flags - for wstate variable */

#define INACTIVE        0x00
#define PENDING         0x01
#define DMATRANS        0x02
#define SPECIAL         0x04

#define _ENETHEADER
#endif
```

```
/*
        File:  include/pup.h
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Includes for PUP stuff.

        MODIFIED BY     REASON
*/

#ifndef _PUP
#include "../include/enetcmds.h"                /* ethernet stuff */

#include "../include/pupconstants.h"            /* W.K.S., PupTypes, etc */
#include "../include/pupstatus.h"               /* error codes */

#define OurNetNumber    0               /* initial guess of net */

#define MAXPUPDATALEN   532             /* max # of data bytes in PUP */
#define PUPPACKOVER     22              /* pup packet overhead */
#define RECVOVER        22              /* 10 Mbit ethernet packet overhead
                                        *  for receive. (2 Stat, 2 Len,
                                        *  6 Dest, 6 Src, 2 Type & 4 CRC).
                                        */
#define TRANSOVER       8               /* 10 Mbit ethernet packet overhead
                                        *  for transmitt with source address
                                        *  insertion. (6 Dest & 2 Type).
                                        */

#define ONESEC          10              /* 100ms clock */
#define INFINITY        0               /* wait forever */

extern PUPdebug;                        /* debug messages */
extern char *PUPErrorMsg;               /* one spot for error messages */

/* PUPCHAN   variable "mode" definitions */
#define PCM_BROADCAST   01              /* read: accept broadcasts */
#define PCM_IGNBADCKS   02              /* read: ignore packet w/bad chksum */
#define PCM_RCHECKSUM   04              /* read: check checksum */
#define PCM_WCHECKSUM   010             /* write: compute and send checksum */

#define PCM_NOBROADCAST 0
#define PCM_NOIGNBADCKS 0
#define PCM_NORCHECKSUM 0
#define PCM_NOWCHECKSUM 0

#define NOCHKSUM        (-1)

/* PUPCHAN - valid only when variable "magic" contains this */
#define PC_MAGIC        0x2e707570      /* .pup */

typedef struct                          /* <net,host,socket> triple */
        {
        unsigned char net;
        unsigned char host;
        unsigned long socket;
        } PORT;                         /* triple is known as "port" */
```

```
typedef struct                          /* Unencapsulated PUP packet */
        {
        unsigned short PupLength;       /* # of bytes in PupPacket */
        unsigned char PupTransport;     /* used by gateways (0 @ source) */
        unsigned char PupType;          /* 1   255 types */
        unsigned long PupID;            /* for higher protocol use (BSP) */
        PORT PupDst;                    /* <net,host,socket> triple */
        PORT PupSrc;                    /* <net,host,socket> triple */
        unsigned char PupData[MAXPUPDATALEN];
        unsigned short Checksum;        /* checksum follows first word after
                                         * data and would only appear here
                                         * if PupData contained 532 bytes
                                         */
        } PUPPACKET;

typedef struct                          /* 10 Mbit Encapsulated PUP packet
                                         * for receive.
                                         */
        {
        unsigned short InFrameStat;     /* incoming frame status */
        unsigned short InFrameLen;      /* incoming frame length */
        unsigned char InDstHost[6];     /* destination address */
        unsigned char InSrcHost[6];     /* destination address */
        unsigned short InFrameType;     /* ethernet type of packet */
        PUPPACKET InPup;                /* the actual pup (data) */
        unsigned long InFrameCRC;       /* CRC follows first word after
                                         * data and would only appear here
                                         * if data field contained 1500 bytes
                                         */
        } RECVFRAME;

typedef struct                          /* 10 Mbit Encapsulated PUP packet
                                         * for transmission with source
                                         * address insertion mode set */
        {
        unsigned char OutDstHost[6];    /* destination address */
        unsigned short OutFrameType;    /* ethernet type of packet */
        PUPPACKET OutPup;               /* the actual pup (data) */
        } TRANSFRAME;

typedef struct                          /* PUP transport channel descriptor */
        {
        PORT DstPort;                   /* pup destination port */
        PORT SrcPort;                   /* pup source port */
        unsigned char ImmHost[6];       /* ethernet address for first hop */
        unsigned short mode;            /* transport channel mode bits */
        unsigned long timeout;          /* read timeout on this channel */
        int fid;                        /* open file id for driver */
        int inec;                       /* packet placed in read Q */
        int outec;                      /* packet removed from read Q */
        int clkec;                      /* 100ms system clock ec */
        unsigned long magic;            /* PUPCHAN initialized flag */
        } PUPCHAN;

#define _PUP
#endif
```

```
/*
        File:  include/pupconstants.h
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Definitions of pup types, sockets.

        MODIFIED BY     REASON
*/

#ifndef _PUPCONSTANT

/*
*  Encapsulated ethernet packet types from <PUP>PupConstants.press
*  NOTE:  values are in octal
*/
#define PeekReportPacket                0402
#define BreathOfLifePacket              0602
#define EchoMePacket                    0700
#define ImAEchoPacket                   0701
#define PupPacket                       01000
#define ArpaInternetPacket              01001
#define OISPacket                       03000
#define TrekPacket                      010000

/*
*  Well-known pup sockets from <PUP>PupConstants.press
*  NOTE:  values are in octal
*/
#define TelenetSocket                   01
#define GateWayInfoSocket               02
#define FTPSocket                       03
#define MiscServicesSocket              04
#define EchoSocket                      05
#define BSPTestSocket                   06
#define MailSocket                      07
#define EFTPReceiveSocket               020
#define EarsStatusSocket                021
#define StatisticsSocket                022
#define CopyDiskSocket                  025
#define EventReportSocket               030
#define PrinterReportSocket             031
#define JuniperPackConversionSocket     034
#define JuniperEventSocket              035
#define RPCPSocket                      036
#define ClearinghouseSocket             037
#define LibrarianSocket                 041
#define WIFSSocket                      042
#define LeafSocket                      043
#define TeleSwatSocket                  060
#define JuniperPineSocket               0100
#define WFSSocket                       0420

/*
*  Registered (0 - 0177) PUP Types from <PUP>PupConstants.press
*  NOTE:  values are in octal
*/
#define EchoMePup                       01
```

```
#define IAmEchoPup                      02
#define IAmBadEchoPup                   03
#define ERRORPUP                        04
#define TracePup                        05
#define RFC                             010
#define RTP_ABORT                       011
#define RTP_END                         012
#define RTP_ENDR                        013
#define BSP_DATA                        020
#define BSP_ADATA                       021
#define BSP_ACK                         022
#define BSP_MARK                        023
#define BSP_INTR                        024
#define BSP_INTRR                       025
#define BSP_AMARK                       026
#define EFTPDataPup                     030
#define EFTPAckPup                      031
#define EFTPEndPup                      032
#define EFTPAbortPup                    033

#define _PUPCONSTANT
#endif
```

```
/*
        File:  include/pupstatus.h
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Define status codes returned from various routines.

        MODIFIED BY     REASON
*/

#ifndef _PUPSTATUS

#define OK              0

#define ABORT           -1      /* connection terminated, misc errors */
#define BADCKSUM        -2      /* received bad checksum */
#define NOCHAN          -3      /* no channel avaiable */
#define NOROUTE         -4      /* can't get there from here (route err) */
#define NOTAPUP         -5      /* encapsulated packet not a PUP */
#define NOTFOUND        -6      /* lookup or mapping failed */
#define PCNOTOPEN       -7      /* pup channel not open */
#define TIMEOUT         -8      /* waiting an didn't get it */

#define _PUPSTATUS
#endif
```

```
/*
        File:  driver/close.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Close a valid minor device and assure receive
                  queue empty.
        Returns:  Success 0 or Failure system error code.

        MODIFIED BY     REASON
        05/30/84 mvd    Updated to version 4.0 of OS.
*/

enetclose (dev, opencount, filep)
devtype dev;                            /* major/minor device number */
int opencount;                          /* current open count */
OPEN_FILEDESC *filep;                   /* file descriptor */
{
        register MINOR *mp;             /* minor device closing */
        RQ *rpq;                        /* receive packet queue pointer */

#ifdef DEBUG
printf ("ENET: CLOSE <%d,%d>\n", dev.d_minor, opencount);
#endif
        if (dev.d_minor >= MAXMINORS)           /* reject bad devices */
                return (EBADDEV);

        mp = &dd.minor[dev.d_minor];
        mp->mstate = MS_CLOSED;

        /*
        *  Remove any queued packets in minors local receive queue
        *  and place them back on free queue list.
        */
        for ( ; mp->rsize; mp->rsize--)
                {
                rpq = (RQ *)GetQ (&mp->rwait);
                ecupdate (&mp->leftqec, read_ec (&mp->leftqec) + 1);
                PutQ (&dd.rfree, &rpq->nextRQ);
                }

        return (SUCCESS);
}
```

```
/*
        File:  driver/driver.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Pulls all enthernet source modules together to allow
                  one relocatable module and thereby reduce the number
                  of global symbols.

        MODIFIED BY     REASON


        The following are routines and data from kernal space used in
        this driver.  With new OS releases, these routines should be
        verified as to same usage and function as previous OS.

        atoba           - maps virtual addr -> absolute addr.
        atoba_len       - check virtual addr + length does not exceed bounds.
        activep         - ptr current process' process control block.
        addressable     - determine if hardware exists.
        biowait           wait for I/O completion on specified SYSBUF.
        ecupdate          advance eventcount and start any waiting process.
        enter_monitor     gain mutual exclusion of SYSBUF.
        exit_monitor      release mutual exclusion of SYSBUF.
        getubytes         copy user space to kernal space.
        ialloc            allocate kernal memory at boot time.
        idef            - load interrupt vector.
        initEC          - initialize eventcounts.
        iodone          - signal completion of I/O request on a SYSBUF.
        newmapeclu      - setup user access to eventcounts.
        Memtop          - last physical space memory address.
        putubytes         copy kernal space to user space.
        read_ec           macro to read eventcount value.
        spl               set priority level interrupt mask.
        superuser       - determine if access priviledge of administrator.
        sys_clock       - address of 100ms system eventcount clock
*/

#include <error.h>                      /* system error codes */
#include <ttymodes.h>                   /* special function codes (GETEC) */
#include <sys/stypes.h>                 /* system types */
#include <sys/filedesc.h>               /* file descriptor structure */
#include <sys/filesys.h>                /* file system structures */
#include "/src/header/sys.d"            /* system constants */
#include "/src/header/eventcount.h"     /* eventcount structures */
#include "/src/header/buf.h"            /* system buffer structures */
#include "/src/header/idef.h"           /* structure for system interrupts */
#include "/src/header/devtab.h"         /* dispatch table descriptor */
#include "/src/header/proc.h"           /* process control block */
#include "../include/enetcmds.h"        /* ethernet commands */
#include "../include/enetheader.h"      /* ethernet driver structures */

#define BELL            '\007'
#define MILLISEC        300             /* ~ on CP32 with cashe active */

#define ALLQS           (MAXMINORS * MAXWAITING + 1)

static DRIVERDATA dd = 0;               /* state information */
```

```
struct devicetable ethertable          /* system device table definitions */
        {
        enetinit,                      /* initial when system booted */
        noent,                         /* queue a request */
        enetread,                      /* raw io read */
        enetwrite,                     /* raw io write */
        enetspfun,                     /* spfun entry */
        enetopen,                      /* device open */
        enetclose,                     /* device close */
        enetmapec                      /* map eventcounts */
        };

static idefblk enet_icb   0;           /* interrupt control block */
extern enetintr ();                    /* interrupt handler */

extern PROC *activep;                  /* current process control block */
extern unsigned int Memtop;            /* last memory address */
extern EVENTCOUNT sys_clock;           /* system 100ms clock */

#include "init.c"                      /* device table init function */
#include "read.c"                      /* device table raw io read func */
#include "write.c"                     /* device table raw io write func */
#include "spfun.c"                     /* device table spfun function */
#include "open.c"                      /* device table open function */
#include "close.c"                     /* device table close function */
#include "mapec.c"                     /* device table mapec function */
#include "intr.c"                      /* device interrupt handler */
#include "start.c"                     /* start device function */
```

```
/*
        File:  driver/init.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Initialize ethernet driver.  The init function is
                  called once when the system is booted.  It will
                  determine if the board exists with system function
                  "addressable" (checks for DTACK) and mark the board
                  availability accordingly.  Also initializes software
                  variables and tables.
        Returns:  Nothing.

        MODIFIED BY     REASON
*/

enetinit ()
{
        register DRIVERDATA *ddp;
        register int i;
        ESTAT sb;                       /* memory for status block */
        register char status;           /* status after a command */
        char temp[32];
        register RQ *ptr;               /* ptr to allocated receive q */

#ifdef DEBUG
extern char *EtherUpdate;
printf ("Ethernet driver updated: %s\n", EtherUpdate);
#endif
        ddp = &dd;                      /* make register variable */
        if (addressable (&C_REG, 1))    /* bd there, DTACK */
                {
                ddp->hw_state   HW_BOARD;
                printf ("Ethernet board installed .... ");

                /*
                *  Initialize interrupt vector.
                */
                idef (&enet_icb, ETHERVEC, enetintr, NULL);

                /*
                *  Initialize raw write transfer buffer.
                */
                initEC (&ddp->enet_rbuf.b_iodones);

                /*
                *  Initialize SYSBUF write.
                */
                ddp->wstate = INACTIVE;

                /*
                *  Show all devices closed and initialize eventcounts
                *  showing packets that have move through receive queue.
                */
                for (i = 0; i < MAXMINORS; i++)
                        {
                        ddp->minor[i].mstate = MS_CLOSED;
                        initEC (&ddp->minor[i].leftqec);
```

```
        initEC (&ddp->minor[i].putqec);
        }

/*
*  Setup memory and pointer for receive Q.
*/
ptr = (RQ *) ialloc (sizeof (RQ) * ALLQS);
if (ptr == NULL)
        {
        printf ("but no memory for receive Q%c\n", BELL);
        return;
        }
InitQ (ddp->rfree);
for (i   0; i < ALLQS; i++, ptr++)
        PutQ (&ddp->rfree, &ptr->nextRQ);

/*
*  Mark last receive and last transmit time.
*/
ddp->last_receive   0;
ddp->last_transmit = 0;

/*
*  Start hardware initialization.
*/
while (!statreg (MILLISEC))     /* while no timeout */
        status = S_REG;         /* flush status reg */

IE_REG = ddp->hw_intr   IER_DISABLE;    /* no interrupts */

ddp->hw_state &= HW_NORESET;            /* assume fail */
C_REG   IOCMD(Reset);
if (statreg (MILLISEC * 1000)) /* reset time to finish */
        {
        printf ("but timeout on reset IS_REG%c\n", BELL);
        return;
        }

if ((status   S_REG) != IOST(CMDOK))    /* reset ok ? */
        {
        printf ("but reset failed - status code: 0x%x%c\n",
                                        status, BELL);
        return;
        }

ddp->hw_state |= HW_RESET;              /* reset passed */

if (getbdstats (&sb))                   /* get net addr */
        {
        printf ("but timeout on report status%c\n", BELL);
        return;
        }

movebytes (&sb.physadr, ddp->enaddr, sizeof (ddp->enaddr));
printf ("net address 0x%s\n",
        chex (&sb.physadr, sizeof (sb.physadr), &temp));
```

```
                C_REG   IOCMD(Online);                  /* bring it online */
                if (statreg (MILLISEC))                 /* time to finish */
                        {
                        printf ("Ethernet board timeout while %s%c\n",
                                        "attempting to go Online", BELL);
                        return;
                        }

                if ((status = S_REG) != IOST(CMDOK))    /* online ok ? */
                        {                               /* should not fail */
                        printf ("%s %s - status code: 0x%x%c\n",
                                        "Ethernet board failure while",
                                        "attempting to go Online",
                                        status, BELL);
                        return;
                        }
                ddp->hw_state |= HW_ONLINE;             /* mark hw state */
                IE_REG = ddp->hw_intr = IER_RECEIVE;    /* now receiving */

#ifdef DEBUG
if (Memtop > 0x100000)
{
printf ("ENET: Warning   system has more than 1MByte of memory.\n");
printf ("Assure that Hal-Versa bd is configured for full VersaBus range.\n");
}
#endif
                }
        else
                {
                printf ("Ethernet board not installed or DTACK problem%c\n",
                                                                BELL);
                ddp->hw_state = HW_NOBOARD;
                }

        return;
}

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

Routine below polls the interrupt status register during initialization,
since interrupts are not enabled yet by processor, waiting for the status
register full bit.

*/

static int statreg (tlimit)
register int tlimit;
{
        register int to;                /* time out */

        for (to = 0; to < tlimit && (IS_REG & ISR_SRF) != ISR_SRF; to++)
                ;

        return (to >= tlimit);
}

/*****************************************************************************
```

```
Convert hex chars keeping leading zeroes.  The operating system version of
printf (format) does not handle this correctly.

*/

static char *chex (start, length, destination)
register char *start;
int length;
register char *destination;
{
        int i;
        char *saveptr;
        register unsigned int temp;

        saveptr = destination;
        for (i = 0; i < length; i++, start++)
                {
                temp = (*start & 0xf0) >> 4;
                *destination++ = (temp > 9 ? 'A' + temp - 10 : '0' + temp);
                temp = *start & 0x0f;
                *destination++ = (temp > 9 ? 'A' + temp   10 : '0' + temp);
                }

        *destination   NULL;            /* follow string convention */
        return (saveptr);
}
```

```
/*
        File:  driver/intr.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Ethernet driver interrupt handler.
        Returns:  Nothing.

        MODIFIED BY     REASON
*/

enetintr ()
{
        register SYSBUF *bp;
        register DRIVERDATA *ddp;
        register unsigned char *rp;
        register int status;

        ddp = &dd;                          /* make register variable */

#ifdef DEBUG
printf ("ENET: INTR <0x%x,%d>\n", IS_REG ↑ ISR_NORMALIZE, ddp->hw_intr);
#endif

        /*
        *  The NI3010A lacks the ability to be able to check if
        *  interrupt occurred is the one desired in every case.
        *  We have to assume that it is.
        *
        *       copy IE_REG     IS_REG
        *       1               0x01            StatusRegisterFull
        *       2               0x02            StatusBlockAvailable'
        *       4               0x08            ReceiveBlockAvailable'
        *       6               none            DMA transmit done
        *       7               none            DMA receive done
        */
        switch (ddp->hw_intr)
                {
case IER_STATUSREG:
                /*
                *  Check the status of the command just completed.
                *  Mark state of hardware as idle and start any pending
                *  transfer requests.
                */
                if ((IS_REG ↑ ISR_NORMALIZE) & ISR_SRF)
                        {
                        IE_REG = ddp->hw_intr = IER_DISABLE;    /* clear */
                        status - S_REG | BASEERR;               /* status */
                        enetnext (&ddp->enet_rbuf, status);
                        ddp->hw_state &= HW_IDLE;
                        enetstart ();
                        }
#ifdef DEBUG
else printf ("ENET: INTR waiting for SRF != IS_REG\n");
#endif
                break;

case IER_RECEIVE:
```

```
                /*
                *  Receive frame from ethernet by starting a DMA
                *  after insuring hardware is idle.  This is more
                *  of a consistency check of driver because it
                *  should never get in this state.  If hardware is
                *  idle then show it busy.
                */
                if ((IS_REG ↑ ISR_NORMALIZE) & ISR_RBA)
                        {
                        IE_REG   IER_DISABLE;           /* clear interrupt */
                        if ((ddp->hw_state & HW_BUSY) == FALSE)
                                {
                                ddp->hw_state |= HW_BUSY;
                                ddp->last_receive = read_ec (&sys_clock);
                                start_receive ();
                                }
                        else
                                IE_REG - ddp->hw_intr;
                        }
#ifdef DEBUG
else printf ("ENET: INTR waiting for RBA != IS_REG\n");
#endif
                break;

case IER_XMITDMA:
                /*
                *  DMA transfer from host memory to NI3010A board
                *  transmit FIFO completed.  Issue command to
                *  transmit data onto ethernet and interrupt when done.
                */
                IE_REG   IER_DISABLE;                   /* clear interrupt */
                C_REG   IOCMD(LoadTransDataSend);
                IE_REG   ddp->hw_intr = IER_STATUSREG;
                break;

case IER_RECVDMA:
                /*
                *  DMA transfer from NI3010A receive FIFO to host
                *  memory completed.  Filter received frame and place
                *  in minors waiting queue.  Mark state of hardware
                *  idle and start any pending transfer requests.
                */
                IE_REG = IER_DISABLE;                   /* clear interrupt */
                enetfiltfram (ddp);
                ddp->hw_state &= HW_IDLE;               /* hardware free */
                enetstart ();
                break;

default:
#ifdef DEBUG
printf ("ENET: INTR <0x%x,%d> not expected\n", IS_REG ↑ ISR_NORMALIZE,
ddp->hw_intr);
#endif
                IE_REG = IER_DISABLE;                   /* clear interrupt */
                IE_REG   ddp->hw_intr;
                break;
                }
```

```
        return;
}

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

Check if an error occurred.  If successful, place the number of bytes
transfered in rawret.  Signal completion on this SYSBUF to system.

Args:
        bp      - SYSBUF pointer.
        error   - error value to return to system.
Returns:
        none
*/

static enetnext (bp, error)
SYSBUF *bp;
int error;
{

        if ((dd.wstate & SPECIAL) || (dd.wstate & DMATRANS))
                {
                if (error == CMDOK ||
                    bp->b_bufp != &dd.spfuncode && error == CMDOK_RETRIES ||
                    bp->b_bufp == &dd.spfuncode &&
                                dd.spfuncode == IOCMD (NetworkLpbkTest))
                        {
                        bp->b_rawret = bp->b_bcnt;
                        error = 0;                      /* no err */
                        }
                else
                        error = -error;

#ifdef DEBUG
printf ("ENET: DONE <%d> error %d\n", bp->b_device.d_minor, error);
#endif

                dd.wstate = INACTIVE;           /* SYSBUF is free */
                iodone (bp, error);             /* tell system */
                }

        return;
}

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

Filter received frame to see which minor device owns it.

Args:
        ddp       pointer to DRIVERDATA.
Returns:
        none
*/

static enetfiltfram (ddp)
register DRIVERDATA *ddp;
```

```
{
        register FILTER *ef;            /* pointer to minor's filter */
        register int i;
        register int match;             /* filter match flag */
        register MINOR *mp;             /* pointer to matched minor */
        register RFF *rp;               /* ptr to frame just received */

        rp   &ddp->rcurrent->Rpkt;      /* get receive buffer ptr */
        for (i = 0, match   FALSE; match == FALSE && i < MAXMINORS; i++)
                {
                if ((ddp->minor[i].mstate & MS_FILTER) == FALSE)
                        continue;       /* skip - minor has no filter */

                match   TRUE;                   /* assume this minors frame */
                ef = &ddp->minor[i].enfilt;     /* get minors filter */

                /*
                *  Perform filter process on packet vs what minor desires.
                */
                if (ef->fwhich.do_enetpacket)
                        match &= ef->fdata.f_enetpacket == rp->f_enetpacket;
                if (ef->fwhich.do_puptype)
                        match &= ef->fdata.f_puptype == rp->f_puptype;
                if (ef->fwhich.do_dsthost)
                        match &= ef->fdata.f_dsthost == rp->f_dsthost;
                if (ef->do_dstsocket)
                        match &= ef->fdata.f_dstsocket == rp->f_dstsocket;
                }

        /*
        *  When we fall out of loop it is due to a match or all minors
        *  devices have been check and no match exists.  If it was a
        *  match "i" will be pointing one past the minor that matched.
        *  If minor has room in its own receive queue, fix byte swaps
        *  and compute frame length.  Advance "packet waiting" eventcount.
        */
        if (match)                      /* frame belongs to minor */
                {
                mp = &ddp->minor[i - 1];
                if (mp->rsize < MAXWAITING)
                        {               /* room in minors receive queue */
                        swapbytes (&ddp->rcurrent->Rpkt[0]);
                        swapbytes (&ddp->rcurrent->Rpkt[2]);

                        ddp->rcurrent->Rlen = *((unsigned short *)
                                        &ddp->rcurrent->Rpkt[2]) + 4;

                        PutQ (&mp->rwait, &ddp->rcurrent->nextRQ);

                        mp->rsize++;    /* increment available count */
#ifdef DEBUG
printf ("ENET: FILTER <%d,%d>\n", i   1, mp->rsize);
#endif
                        ecupdate (&mp->putqec, read_ec (&mp->putqec) + 1);
                        }
                else                    /* this minors Q exceeded */
                        PutQ (&ddp->rfree, &ddp->rcurrent->nextRQ);
```

```
                }
        else                                    /* no one wants this frame */
                PutQ (&ddp->rfree, &ddp->rcurrent->nextRQ);

        return;
}
```

```
/*
        File:  driver/mapec.c
        Date:  January 16 1984
        Author:  Mark Van Oellon
        Purpose:  Map an ethernet eventcount for the file system.
        Returns:  Address of desired eventcount structure.

        MODIFIED BY     REASON
*/

EVENTCOUNT *enetmapec (filep, ecnum)
OPEN_FILEDESC *filep;                           /* file descriptor*/
int ecnum;                                      /* ec number to driver */
{
        register int dev;                       /* minor device reading */
        register MINOR *mp;
        register retval;

#ifdef DEBUG
printf ("ENET: MAPEC <%d,%d>\n", filep->of_device.d_minor, ecnum);
#endif

        dev = filep->of_device.d_minor;
        if (dev > MAXMINORS)                            /* out of range */
                return (EBADDEV);

        mp = &dd.minor[dev];
        if ((mp->mstate & MS_OPEN) == FALSE)            /* must be open */
                return (ENOTOPEN);

        switch (ecnum)
                {
case PUTQEC:
                retval   &mp->putqec;
                break;
case LEFTQEC:
                retval = &mp->leftqec;
                break;
default:
                retval = EBADCALL;
                break;
                }

        return (retval);
}
```

```
/*
        File:  driver/open.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Open valid minor device assuring not busy
                  and hardware ok.
        Returns:  Success 0 or Failure system number.

        MODIFIED BY     REASON
        05/30/84 mvd    Updated to version 4.0 of OS.
*/

enetopen (dev, opencount, filep)
devtype dev;                                    /* major/minor number */
int opencount;                                  /* current open count */
OPEN_FILEDESC *filep;                           /* file descriptor */
{
#ifdef DEBUG
printf ("ENET: OPEN <%d,%d>\n", dev.d_minor, opencount);
#endif

        if (dev.d_minor >= MAXMINORS)           /* reject bad devices */
                return (EBADDEV);

        if (dd.minor[dev.d_minor].mstate & MS_OPEN)
                return (EFBUSY);                /* this minor busy */

        if (superuser ())
                {                               /* superuser restrictions */
                if (dd.hw_state == HW_NOBOARD)
                        return (EBADCALL);
                }
        else
                {                               /* normal user restrictions */
                if ((dd.hw_state & HW_ONLINE) == FALSE)
                        return (ENOTONLINE);
                }

        /*
        *  Mark device open and initialize minor structure.
        *  Show no frame waiting and init receive frame queue.
        */
        dd.minor[dev.d_minor].mstate = MS_OPEN;
        dd.minor[dev.d_minor].rsize = 0;
        InitQ (dd.minor[dev.d_minor].rwait);

        return (SUCCESS);
}
```

```
/*
        File:  driver/read.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Do a direct read transfer from ethernet.
        Returns:  The number of bytes read or system error.

        MODIFIED BY     REASON
*/

enetread (filep, index, bufp, length)
OPEN_FILEDESC *filep;                           /* file descriptor*/
long index;                                     /* disk index (not used) */
char *bufp;                                     /* user buffer */
int length;                                     /* buffer length */
{
        int dev;                        /* minor device reading */
        register MINOR *mp;             /* pointer to minor structure */
        register int oldpl;
        RQ *rpq;                        /* receive packet queue pointer */
        register int retval;

#ifdef DEBUG
printf ("ENET: READ <%d,%d>\n", filep->of_device.d_minor,
dd.minor[filep->of_device.d_minor].rsize);
#endif

        dev = filep->of_device.d_minor;
        if (dev > MAXMINORS)                            /* out of range */
                return (EBADDEV);

        mp = &dd.minor[dev];
        if ((mp->mstate & MS_OPEN) == FALSE)            /* must be open */
                return (ENOTOPEN);

        if (mp->rsize != 0)                             /* data waiting ? */
                {
                /*
                *  Get packet from that minors receive queue
                *  and return from kernal space to user space.
                *  If user area too small bytes are lost.
                *  Replace packet queue element back on free list
                *  and advance eventcount showing packet left Q.
                */
                oldpl = spl (LEVEL(ETHERINTR));
                rpq   GetQ (&mp->rwait);
                mp->rsize--;
                retval = putubytes (&rpq->Rpkt, bufp,
                                                min (length, rpq->Rlen));
                PutQ (&dd.rfree, &rpq->nextRQ);
                ecupdate (&mp->leftqec, read_ec (&mp->leftqec) + 1);
                spl (oldpl);
                }
        else
                retval = 0;             /* nothing to read */

        return (retval);
```

```
}

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

Queue package designed for handling received packets from ethernet.
If we had source to operating system something like this probably
already exists.

*/

static PutQ (qh, newdata)                   /* place on end of queue */
register QHEAD *qh;                         /* queue head */
QLINK *newdata;                             /* new data for queue */
{

        *newdata = NULL;                    /* new item pointer */
        if (qh->FL == NULL)                 /* q empty */
                qh->BL = qh;
        qh->BL->FL   newdata;               /* forward link */
        qh->BL = newdata;                   /* backward link */

        return;
}


static QLINK *GetQ (qh)                     /* take from front of queue */
register QHEAD *qh;                         /* queue head */
{
        QLINK *retdata;                     /* data from queue to return */

        if (qh->FL != NULL)                 /* queue empty ? */
                {
                retdata = qh->FL;           /* this is what we return */
                qh->FL = *retdata;          /* point over data returned */
                return (retdata);
                }
        return (NULL);
}
```

```
/*
        File:  driver/spfun.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Perform special function for ethernet board driver.
                  Only a process with superuser priviledge may issue
                  board commands.
        Returns:  Success 0 or Failure system error.

        MODIFIED BY     REASON
*/

enetspfun (file, function, arg)
OPEN_FILEDESC *file;
int function;
char *arg;
{
        DVALUES block;                  /* create block to copy from
                                         * kernal space to user space */
        register DRIVERDATA *ddp;
        int error;
        register MINOR *mp;             /* pointer to file minor structure */
        int oldpl;                      /* old priority interrupt level */
        int result;                     /* putubytes result */
        ESTAT sb;                       /* create block to copy from
                                         * kernal space to user space */
        unsigned long temp;

#ifdef DEBUG
printf ("ENET: SPFUN <%d,0x%x>\n", file->of_device.d_minor, function);
#endif

        ddp = &dd;

        if (file->of_device.d_minor > MAXMINORS)
                return (EBADDEV);               /* minor out of range */

        mp = &ddp->minor[file->of_device.d_minor];

        if ((mp->mstate & MS_OPEN) == FALSE)
                return (ENOTOPEN);              /* file must be open */

        if (superuser ())
                {                               /* superuser limitations */
                if (ddp->hw_state == HW_NOBOARD)
                        return (EBADCALL);      /* no hardware */
                }
        else
                {                               /* normal usr limitations */
                if ((ddp->hw_state & HW_ONLINE) == FALSE)
                        return (ENOTONLINE);    /* hardware not online */
                if (function >= BASESU)
                        return (EACCESS);       /* correct priviledge */
                }


        switch (function)
```

```
                {
case SetModuleLpbk:                     /* set module interface loopback */
case SetInternalLpbk:                   /* set internal loopback */
case Offline:                           /* go offline */
case RunDiagnostics:                    /* run onboard diagnostics */
case Reset:                             /* reset */
                ddp->hw_state &= HW_OFFLINE;
                goto givecmd;

case Online:                            /* go online */
                ddp->hw_state |= HW_ONLINE;
                goto givecmd;

case ClrLpbk:                           /* clear loopback */
case SetPromiscuous:                    /* set promiscuous mode */
case ClrPromiscuous:                    /* clear promiscuous mode */
case SetReceiveOnErr:                   /* set receive-on-error mode */
case ClrReceiveOnErr:                   /* clear receive-on-error mode */
case SetInsertSrcAddr:                  /* set insert source address mode */
case ClrInsertSrcAddr:                  /* clear insert source address mode */
case SetPhyAddrDefault:                 /* set physical address to default */
case SetRecMulticast:                   /* set receive all multicast packet */
case ClrRecMulticast:                   /* clear receive multicast packets */
case NetworkLpbkTest:                   /* perform network loopback test */
case CollisionTest:                     /* perform collision detect test */
case LoadTransData:                     /* load transmit data */
case LoadTransDataSend:                 /* load transmit data and send */

givecmd:
                /*
                *  Grab the I/O buffer.
                */
                enter_monitor (&ddp->enet_rsemph);

                /*
                *  Construct buffer header to pass to
                *  request routine in the driver.
                *
                *  (1) bufp is used as the "message" buffer pointer
                *  (2) BIOINPROGRESS is set in the flags so that
                *      biowait will perform properly.
                */
                ddp->enet_rbuf.b_device = devof (file);
                ddp->enet_rbuf.b_bufp = &ddp->spfuncode;
                ddp->enet_rbuf.b_flags = BIOINPROGRESS;

                /*
                *  Put correct information in "message" buffer
                */
                ddp->spfuncode = IOCMD (function);

                /*
                *  Start command ... then wait
                */
                oldpl   spl (LEVEL(ETHERINTR));
                ddp->wstate |= PENDING;
                enetstart ();
```

```
                spl (oldpl);
                biowait (&ddp->enet_rbuf);

                /*
                *  Reset bufp address so that future I/O
                *  requests will not look like special functions
                */
                ddp->enet_rbuf.b_bufp   NULL;

                /*
                *  Record any error and release the buffer
                */
                error = ddp->enet_rbuf.b_error;
                exit_monitor (&ddp->enet_rsemph);

                return (error);
                break;

case ReportStatus:                      /* report and reset statistics */
                enter_monitor (&ddp->enet_rsemph);
                oldpl = spl (LEVEL(ETHERINTR));
                IE_REG   IER_DISABLE;           /* no bd interrupts */
                spl (oldpl);
                if (getbdstats (&sb))
                        result = EIOERR;
                else
                        result = putubytes (&sb, arg, sizeof (ESTAT));
                IE_REG = dd.hw_intr;            /* set to previous */
                exit_monitor (&ddp->enet_rsemph);
                return (result < 0 ? result : SUCCESS);
                break;

case ReportCollision:                   /* report collision delay times */
case LoadGroupAddr:                     /* load group address */
case DeleteGroupAddr:                   /* delete group address */
case LoadPhyAddr:                       /* load physical address */
                printf ("ENET: SPFUN 0x%x - not implemented\n", function);
                return (EBADCALL);
                break;

case GetDriverValues:
                movebytes (ddp->enaddr, block.netaddress,
                                                sizeof (block.netaddress));
                block.numminors = MAXMINORS;
                block.maxrecvq   MAXWAITING;
                block.bdvector = ETHERVEC >> 2;
                block.bdaddress   ETHERADDR;
                block.bdilevel = ETHERINTR;
                block.bdiwait   ddp->hw_intr;
                block.bdstate   ddp->hw_state;
                result = putubytes (&block, arg, sizeof (DVALUES));
                return (result < 0 ? result : SUCCESS);
                break;

case GetMinorQcount:
                temp   mp->rsize;       /* short to long */
                result = putubytes (&temp, arg, sizeof (long));
```

```
                return (result < 0 ? result : SUCCESS);
                return (result);

case SetMinorFilter:
                /*
                *  A nonzero value is an address pointing to filter,
                *  where a zero value means reset to no filter.
                */
                if (arg)
                        {
                        result   getubytes (arg, &mp->enfilt, sizeof(FILTER));
                        if (result > 0)
                                mp->mstate |= MS_FILTER;
                        return (result < 0 ? result : SUCCESS);
                        }
                mp->mstate &= ~MS_FILTER;
                return (SUCCESS);
                break;

case GETEC:
                /*
                *  Called from ecget which opens an eventcount for
                *  a device.  ecget also does some other mapping in
                *  user space.
                */
                return (newmapeclu (file, arg));
                break;
                }

        return (EBADCALL);
}

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

Execute ReportStatus command for NI3010A with no interrupts on board
enabled.  Its up to the calling process to disable board interrupts.

*/

static int getbdstats (sptr)
unsigned char *sptr;                    /* pointer into status block */
{
        int delay;                      /* delay between read counter */
        register int i;
        ESTAT *saveptr;                 /* save ESTAT address */

        C_REG   IDCMD(ReportStatus);

        if (statreg (MILLISEC))         /* time to finish */
                return (EIOERR);

        saveptr = sptr;                 /* for use later */
        for (i = 0; ((IS_REG ↑ ISR_NORMALIZE) & ISR_SBA) &&
                                        i < sizeof (ESTAT); i++)
                {
                if (statreg (MILLISEC))
                        return (EIOERR);
```

```
                *sptr++ = S_REG;
                for (delay = 0; delay < 4; delay++)
                        ;                       /* processor loops too fast */
                }

        /*
        *  high order   low order byte swap fix of all shorts
        */
        swapbytes (&saveptr->framlen);
        swapbytes (&saveptr->in_tot);
        swapbytes (&saveptr->in_q);
        swapbytes (&saveptr->out_tot);
        swapbytes (&saveptr->out_col_16);
        swapbytes (&saveptr->in_runt);
        swapbytes (&saveptr->in_lost);
        swapbytes (&saveptr->multi_accept);
        swapbytes (&saveptr->multi_reject);
        swapbytes (&saveptr->in_crc);
        swapbytes (&saveptr->in_align);
        swapbytes (&saveptr->out_col_1);
        swapbytes (&saveptr->out_col_wind);

        return (SUCCESS);
}

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

Routine to swap high order and low order bytes.  Since the NI3010A
is a multibus board, this problem exists in a few places.

*/

static swapbytes (upbyte)
register char *upbyte;
{
        register char swaptemp;
        register char *lowbyte;

        lowbyte = upbyte + 1;
        swaptemp = *upbyte;
        *upbyte = *lowbyte;
        *lowbyte = swaptemp;

        return;
}
```

```
/*
        File:  driver/start.c
        Date:  January 16 1984
        Author:  Mark Van Dellon
        Purpose:  Check if a new transfer must be started.  We must be
                  careful not to give preference to (or block totally)
                  a transmit or receive.

        MODIFIED BY     REASON


        NOTE:  This routine should only be run when interrupt priority
               level is equal (or greater) than that of the device.

        ALGORITHM:
        1.  Check if hardware is busy with transaction by examining the
            hardware state variable.  Exit if busy.
        2.  Disable ethernet board interrupts.
        3.  If receive was previous transaction then serve transmit as
            first choise.  If nothing to transmit then serve receive.
        4.  If transmit was previous transaction then serve receive as
            first choise.  If nothing to receive then serve transmit.
        5.  If neither transmit or receive initiated then enable RBA
            interrupt.
*/

static enetstart ()
{
        register DRIVERDATA *ddp;

        ddp   &dd;                          /* make device data register */

        if ((ddp->hw_state & HW_BUSY) == FALSE)
                {
                IE_REG = ddp->hw_intr = IER_DISABLE;
                if (ddp->last_transmit >= ddp->last_receive)
                        {               /* service receive first */
                        if ((IS_REG ↑ ISR_NORMALIZE) & ISR_RBA)
                                {
                                ddp->hw_state |= HW_BUSY;
                                ddp->last_receive = read_ec (&sys_clock);
                                start_receive ();
                                }
                        else if (ddp->wstate & PENDING)
                                {
                                ddp->hw_state |= HW_BUSY;
                                ddp->last_transmit = read_ec (&sys_clock);
                                start_transmit ();
                                }
                        }

                else
                        {               /* service transmit first */
                        if (ddp->wstate & PENDING)
                                {
                                ddp->hw_state |= HW_BUSY;
                                ddp->last_transmit   read_ec (&sys_clock);
```

```
                        start_transmit ();
                        }
                else if ((IS_REG ↑ ISR_NORMALIZE) & ISR_RBA)
                        {
                        ddp->hw_state |= HW_BUSY;
                        ddp->last_receive = read_ec (&sys_clock);
                        start_receive ();
                        }
                }

        if ((ddp->hw_state & HW_BUSY) == FALSE)
                IE_REG   ddp->hw_intr = IER_RECEIVE;
        }                                       /* hardware idle */

    return;
}

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

Start a receive DMA by setting up address and count registers on
hardware.  Interrupts must have been disabled by calling routine.

*/

static start_receive ()
{
    register char *rp;

    dd.rcurrent = (RQ *) GetQ (&dd.rfree);
    if (dd.rcurrent != NULL)
        {
#ifdef DEBUG
printf ("ENET: START RECEIVE <0x%x>\n", dd.rcurrent);
#endif
        rp   &dd.rcurrent->Rpkt;
        E_BAR   (long) rp >> 16;
        H_BAR   (long) rp >> 8;
        L_BAR   (long) rp;
        H_BCR = MAXPACKET >> 8;
        L_BCR = MAXPACKET;
        IE_REG = dd.hw_intr   IER_RECVDMA;
        }
    else
        IE_REG = dd.hw_intr = IER_RECEIVE;

    return;
}

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

Start a transmit DMA by setting address and count registers on
hardware.  Interrupts must have been disabled by calling routine.

*/

static start_transmit ()
{
```

```
        register SYSBUF *bp;

        bp   &dd.enet_rbuf;

        /*
        *  If buffer address is that of my own than it is
        *  a special function being performed.
        */
        if (bp->b_bufp == &dd.spfuncode)
                {
#ifdef DEBUG
printf ("ENET: START SPECIAL <%d,0x%x>\n", bp->b_device.d_minor,
bp->b_bufp->b_buf[0]);
#endif
                dd.wstate |  SPECIAL;   /* set write state */
                bp->b_rawret   0;       /* init raw transfer total */
                C_REG = bp->b_bufp->b_buf[0];
                IE_REG   dd.hw_intr   IER_STATUSREG;
                }

        else
                {
#ifdef DEBUG
printf ("ENET: START TRANSMIT <%d,%d>\n", bp->b_device.d_minor,
bp->b_bcnt);
#endif
                dd.wstate |= DMATRANS;  /* set write state */
                bp->b_rawret = 0;       /* init raw transfer total */
                E_BAR = bp->b_busadr >> 16;
                H_BAR = bp->b_busadr >> 8;
                L_BAR = bp->b_busadr;
                H_BCR   bp->b_bcnt >> 8;
                L_BCR   bp->b_bcnt;
                IE_REG   dd.hw_intr = IER_XMITDMA;
                }

        return;
}
```

```
/*
        File:  driver/write.c
        Date:  January 16 1984
        Author:  Mark Van Oellon
        Purpose:  Do a direct write transfer from ethernet.
        Returns:  The number of bytes read or system error code.

        MODIFIED BY     REASON
        05/30/84 mvd    Replaced "rawio" with own code (rawio will not allow
                        odd length).
*/

enetwrite (filep, index, bufp, length)
OPEN_FILEDESC *filep;                           /* file descriptor*/
long index;                                     /* disk index (not used) */
char *bufp;                                     /* user buffer */
int length;                                     /* buffer length */
{
        register SYSBUF *bp;
        int dev;                                /* minor device writting */
        register int oldpl;
        register int retval;

#ifdef DEBUG
printf ("ENET: WRITE <%d,%d>\n", filep->of_device.d_minor, length);
#endif
        dev = filep->of_device.d_minor;
        if (dev > MAXMINORS)
                return (EBADDEV);

        if ((dd.minor[dev].mstate & MS_OPEN) == FALSE) /* must be open */
                return (ENOTOPEN);

        if ((int)bufp & 0x00000001)                     /* addr even bound */
                return (EDDDADDR);

        if (atoba_len (bufp, length) == 0)              /* valid user space */
                return (EBUSERR);

        /*
        *  Mark process state as I/O in progress and grab I/O buffer.
        */
        activep->p_flags |= PRAWIO;
        enter_monitor (&dd.enet_rsemph);

        /*
        *  Construct buffer header for start.
        */
        bp   &dd.enet_rbuf;
        bp->b_device = devof (filep);
        bp->b_flags = BIOINPROGRESS;
        bp->b_busadr = atoba (bufp);
        bp->b_bcnt = length;
        bp->b_blkno   0;

        /*
        *  Start command ... then wait.
```

```
        */
        oldpl = spl (LEVEL(ETHERINTR));
        dd.wstate |= PENDING;
        enetstart ();
        spl (oldpl);
        biowait (bp);

        /*
        *  Mark process state as I/O done and record any errors.
        */
        activep->p_flags &= ~PRAWIO;
        if (bp->b_error != 0)
                retval   bp->b_error;
        else
                retval = bp->b_rawret;

        /*
        *  Release buffer.
        */
        exit_monitor (&dd.enet_rsemph);

        return (retval);
}
```

```
/*
        File:  pup/checksum.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Compute PUP packet checksum.
        Returns:  Checksum.

        MODIFIED BY     REASON

        Algorithm:
        The checksum is initilaized to 0 and computed by repeated
        one's complement addition and left cycle, starting at the
        PUP Length word and ending with the last content word.
        Note that the checksum includes the garbage byte if there
        is one.  If the result of the checksum is the ones-complement
        value of minus zero (0xffff), it should be converted to
        plus zero.  0xffff is specifically defined to mean that the
        PUP carries no checksum.

        Since C does not have a one'c complement addition operator
        we have to work a little harder.
*/

#include "../include/pup.h"

static union
        {
        unsigned long bothwords;
        struct
                {
                unsigned short upper;
                unsigned short lower;
                } split;
        } cksum;

unsigned short pupChecksum (buf, length)
unsigned short *buf;
int length;
{

        for (cksum.bothwords = 0; length > 0; length -= 2)
                {
                cksum.bothwords += *buf++;              /* add in short */
                cksum.split.lower += cksum.split.upper; /* make 1's comp */
                cksum.split.upper   0;                  /* reset carry */
                cksum.bothwords <<= 1;                  /* shift */
                cksum.split.lower += cksum.split.upper; /* make it rotate */
                cksum.split.upper   0;                  /* reset carry */
                }

        return (cksum.split.lower == NOCHKSUM ? 0 : cksum.split.lower);
}
```

```
/*
        File:  pup/close.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Close a previously opened PUP channel.
        Returns:  Nothing.

        MODIFIED BY     REASON
*/

#include "../include/pup.h"

pupClose (Pchan)
PUPCHAN *Pchan;                         /* channel data structure */
{

        if (Pchan->magic == PC_MAGIC)
                {
                fclose (Pchan->fid);
                ecclose (Pchan->inec);
                ecclose (Pchan->outec);
                ecclose (Pchan->clkec);
                Pchan->magic   ~ PC_MAGIC;      /* show Pchan uninited */
                return;
                }

        if (PUPdebug)
                printf ("pupClose: tried to close a PUPCHAN not open\n");

        return;
}
```

```
/*
        File:  pup/open.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Return an open pup channel between a socket on this
                  host and a <net,host,socket>.
        Returns:  OK or error code.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include <defs.h>
#include <error.h>
#include "../include/pup.h"

pupOpen (Pchan, srcsock, DstPort)
register PUPCHAN *Pchan;                /* channel data structure */
unsigned long srcsock;                  /* source socket */
PORT *DstPort;                          /* destination <net,host,socket> */
{

        if ((Pchan->fid = enopen ()) < 0)
                return (NOCHAN);

        Pchan->magic   PC_MAGIC;        /* show channel inited */
        Pchan->inec = ecget (Pchan->fid, PUTQEC, "r");
        Pchan->outec = ecget (Pchan->fid, LEFTQEC, "r");
        Pchan->clkec = ecopen ("/dev/clock");
        Pchan->SrcPort.net = OurNetNumber;
        Pchan->SrcPort.host = pupGetOurHost (Pchan);
        Pchan->SrcPort.socket = srcsock;
        Pchan->mode = 0;
        Pchan->timeout   0;             /* default to no timeout */

        if (DstPort)
                if (pupRoute (Pchan, DstPort) != OK)
                        return (NOROUTE);

        return (OK);
}

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

Attempt to open an ethernet port.  Assumes that minors are sequential
decimal numbers starting at /dev/enet0.

*/

static int enopen ()
{
        char devname[16];
        register FILE fptr;
        register int minor;

        for (minor = 0; ; minor++)
                {
```

```
                sprintf (devname, "/dev/enet%d", minor);
                fptr   fileopen (devname, "rw");
                switch (geterrno())
                        {
case SUCCESS:           return (fptr);          /* got a good one */

default:                return (NOCHAN);        /* no more devices */

case EFBUSY:            if (fptr)               /* this one in use */
                                fclose (fptr);
                        break;
                        }
                }

}
```

```
/*
        File:  pup/pupglobals.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Place all PUP symbols you want to be referenced
                  externally in one place and declare then global.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "../include/pup.h"

int PUPdebug   FALSE;

char *PUPErrorMsg = charblock;

static char charblock[MAXPUPDATALEN] - 0;
```

```
/*
        File:  pup/pupmap.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Map 10 Mbit addresses to 3 Mbit and visa-versa.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "../include/pup.h"

#define MATCH           1

pupmap48 (tenMbit)              /* map 48 bits (10M) to 8 bits (3M) */
unsigned char tenMbit[];
{
        int ans;
        char *argp[3];
        extern char *astoi ();
        unsigned char buf[32];
        unsigned int host;
        int noerror;

        sprintf (buf, "0x%02x%02x%02x%02x%02x%02x%02x",
                                tenMbit[0], tenMbit[1], tenMbit[2],
                                tenMbit[3], tenMbit[4], tenMbit[5]);
        if ((ans   findline ("/etc/enetmappings", ';', buf, 0, argp,
                                sizeof (argp) / sizeof (char *))) == MATCH)
                {
                noerror = (*astoi (argp[1], &host) == NULL);
                free (argp[0]);
                free (argp[1]);
                free (argp[2]);
                if (noerror)
                        return (host);
                }

        else if (PUPdebug)
                {
                switch (ans)
                        {
case -1:                printf ("pupmap48: no match\n");        break;
case -2:                printf ("pupmap48: arg error\n");       break;
case -3:                printf ("pupmap48: can't open file\n");break;
case -4:                printf ("pupmap48: alloc error\n");     break;
default:                printf ("pupmap48: astoi error\n");     break;
                        }
                }

        return (NOTFOUND);
}

/*-----------------------------------------------------------------------*/

pupmap8 (host, tenMbit)         /* map 8 bits (3M) to 48 bits (10M) */
unsigned int host;
```

```
unsigned char *tenMbit;
{
        int ans;
        unsigned int addr;
        char *argp[3];
        char buf[16];
        int error;

        sprintf (buf, "%d", host);
        if ((ans = findline ("/etc/enetmappings", ';', buf, 1, argp,
                                sizeof (argp) / sizeof (char *))) == MATCH)
                {
                strncpy (&buf, argp[0], 8);
                buf[8] = NULL;                          /* string convention */
                error - (*astoi (&buf, &addr) == NULL) ? 0 : 1;
                movebytes ((char *)&addr + 1, tenMbit, 3);
                strncpy ((char *)&buf + 2, argp[0] + 8, 6);
                error |= (*astoi (&buf, &addr) == NULL) ? 0 : 2;
                movebytes ((char *)&addr + 1, (char *)tenMbit + 3, 3);
                free (argp[0]);
                free (argp[1]);
                free (argp[2]);
                if (error == 0)
                        return (OK);
                }

        else if (PUPdebug)
                {
                switch (ans)
                        {
case -1:                printf ("pupmap8: no match\n");         break;
case -2:                printf ("pupmap8: arg error\n");        break;
case -3:                printf ("pupmap8: can't open file\n"); break;
case -4:                printf ("pupmap8: alloc error\n");      break;
default:                printf ("pupmap8: astoi 'error' = %d\n", error);
                        }
                }

        return (NOTFOUND);
}
```

```
/*
        File:  pup/pupsubs.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Bunch of support subroutines.

        pupAvailable        checks if packet avaliable for receiver.
        pupGetOurHost       gets mapping of our 10 Mbit addr to 3 Mbit.
        pupPortCopy         copy PORT structures to one another.
        pupPrintPacket      debug routine to print PUP packet contents.
        pupPrintPort        debug routine to print PORT structure triple.
        pupSetMode          sets the mode into the channel structure.
        pupSetTimeout       sets the timeout into the channel structure.
        pupUniqueSocket     generate a unique socket that is not well known.

        MODIFIED BY     REASON
*/

#include <ctypes.h>
#include <stdio.h>
#include "../include/pup.h"

/*-----------------------------------------------------------------*/

pupAvailable (Pchan)
PUPCHAN *Pchan;                         /* channel data structure */
{
        int count;

        spfun (fdown (Pchan->fid), GetMinorQcount, &count);
        return (count);
}

/*-----------------------------------------------------------------*/

pupGetOurHost (Pchan)
PUPCHAN *Pchan;                         /* channel data structure */
{
        DVALUES d;

        spfun (fdown (Pchan->fid), GetDriverValues, &d);
        return (pupmap48 (&d.netaddress));
}

/*-----------------------------------------------------------------*/

pupPortCopy (from, to)
register PORT *from;
register PORT *to;
{
        to->net = from->net;
        to->host = from->host;
        to->socket = from->socket;
}

/*-----------------------------------------------------------------*/
```

```
pupPrintPacket (p, type)
register PUPPACKET *p;
char type;
{
        register char c;
        register int i;
        char line[80];
        register int mod;
        register char *cp;

        printf ("%c PupLength   %d.\t\t\tPupType = 0%o\n", type,
                                p->PupLength, p->PupType & 0xff);
        printf ("  PupTransport = 0x%02x,\t\t\tPupID   0x%08x\n",
                                p->PupTransport & 0xff, p->PupID);
        printf ("  PupDest = [0x%02x#0x%02x#0x%08x]",
                p->PupDst.net & 0xff, p->PupDst.host & 0xff,
                p->PupDst.socket);
        printf ("\tPupSrc   [0x%02x#0x%02x#0x%08x]\n",
                p->PupSrc.net & 0xff, p->PupSrc.host & 0xff,
                p->PupSrc.socket);

        for (i = 0, mod = 0; i < p->PupLength   PUPPACKOVER; i++)
                {
                c = p->PupData[i];
                sprintf (line+1+(mod*3), " %02x", c & 0xff);
                sprintf (line+52+mod++, "%c", isascii(c) && isprint(c) ?
                                                        c : '.');

                if (i % 16 == 15 || i == p->PupLength - PUPPACKOVER   1)
                        {
                        line[0] = ' ';
                        for (cp = line+1+(mod*3); cp <= line+51; )
                                *cp++ = ' ';
                        line[52+mod] = NULL;
                        printf ("%s\n", line);
                        mod = 0;
                        }
                }
}

/*-------------------------------------------------------------*/

pupPrintPort (s1, Port, s2)
char *s1;
PORT *Port;
char *s2;
{
        printf ("%s[%d#%d#%08x]%s", s1, Port->net, Port->host,
                                        Port->socket, s2);
}

/*-------------------------------------------------------------*/

pupSetMode (Pchan, newmode)
PUPCHAN *Pchan;                         /* channel data structure */
short newmode;                          /* new modes */
{
```

```
        Pchan->mode = newmode;
}

/*------------------------------------------------------------------*/

pupSetTimeout (Pchan, timeout)
PUPCHAN *Pchan;                         /* channel data structure */
{
        Pchan->timeout = timeout;
}

/*------------------------------------------------------------------*/

unsigned long pupUniqueSocket ()
{
        static unsigned char number = 0;

        if (++number == 0) number = 1;
        return ((number << 24) | (gettime () & 0xffffff));
}
```

```
/*
        File:  pup/read.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Read a PUP packet and return the requested information.
                  A user may pass NULL for any information not desired.
                  Information will not be returned unless the checksum
                  is ok or the ignore bad checksum is enabled.
        Returns:  OK or error code (usually TIMEOUT).

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "../include/pup.h"

pupRead (Pchan, buf, buflen, Ptype, ID, DstPort, SrcPort)
PUPCHAN *Pchan;                         /* pup write channel */
char *buf;                              /* pup data */
int *buflen;                            /* size of data buffer */
unsigned char *Ptype;                   /* pup type (from header) */
unsigned long *ID;                      /* pup ID (from header) */
PORT *DstPort;                          /* destination port */
PORT *SrcPort;                          /* source port */
{
        register unsigned short csum;   /* checksum on received packet */
        RECVFRAME enp;                  /* receive packet in here */
        register int round;             /* even-bounded packet length */
        register int status;

        status = enread (Pchan->fid, &enp, Pchan->inec, Pchan->outec,
                                        Pchan->clkec, Pchan->timeout);

        if (status == OK)
                {
                if (PUPdebug)
                        pupPrintPacket (&enp.InPup, 'R');

                round   (enp.InPup.PupLength & 0x01) ? enp.InPup.PupLength -
                        PUPPACKOVER + 1 : enp.InPup.PupLength - PUPPACKOVER;

                if (round < 0 || round > MAXPUPDATALEN)
                        return (NOTAPUP);

                if (Pchan->mode & PCM_RCHECKSUM)
                        {
                        csum = *(unsigned short *)&enp.InPup.PupData[round];
                        if (csum != NOCHKSUM &&
                            csum != pupChecksum (&enp.InPup,
                                                 round + PUPPACKOVER   2))
                                {
                                status = BADCKSUM;
                                if (!(Pchan->mode & PCM_IGNBADCKS))
                                        return (status);
                                }
                        }
```

```
                if (buf)
                        movebytes (&enp.InPup.PupData, buf, round);

                if (buflen)
                        *buflen   enp.InPup.PupLength - PUPPACKOVER;

                if (Ptype)
                        *Ptype = enp.InPup.PupType;

                if (ID)
                        *ID   enp.InPup.PupID;

                if (SrcPort)
                        pupPortCopy (&enp.InPup.PupSrc, SrcPort);

                if (DstPort)
                        pupPortCopy (&enp.InPup.PupDst, DstPort);
                }

        return (status);
}

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

Read a PUP packet if one is immediately available else if timeout is
set do a timeout/read operation else do a wait forever/read operation.

*/

static int enread (fptr, frame, inec, outec, clkec, timeout)
FILE fptr;                              /* ether file id */
RECVFRAME *frame;                       /* ptr to frame space for read */
EC inec;                                /* placed in read q ec */
EC outec;                               /* removed from read q ec */
EC clkec;                               /* 100ms incrementing clock */
unsigned long timeout;                  /* read value timeout */
{
        int numwaiting;
        unsigned long value;

        value   ecread (inec) + 1;

        if (fileread (fptr, frame, sizeof (RECVFRAME)) > 0)
                return (OK);

        if (timeout > 0)
                {
                ecwait (inec, value, clkec, ecread (clkec) + timeout, NULL);
                }
        else
                {
                for (numwaiting = 0; numwaiting == 0; )
                        {
                        ecwait (inec, value, NULL);
                        spfun (fdown (fptr), GetMinorQcount, &numwaiting);
                        }
                }
```

```
        if (fileread (fptr, frame, sizeof (RECVFRAME)) > 0)
                return (OK);

        return (TIMEOUT);
}
```

```
/*
        File:  pup/route.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Decide first hop host for destination port.
        Returns:  OK or error code.


        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "../include/pup.h"

pupRoute (Pchan, Dst)
PUPCHAN *Pchan;
PORT *Dst;                              /* port we want to go to */
{

        if (Dst->net == OurNetNumber || Dst->net == 0)
                {
                pupPortCopy (Dst, &Pchan->DstPort);
                return (pupmap8 (Dst->host, &Pchan->ImmHost));
                }

        /*
        *  more code goes here if going off net
        */
        printf ("puproute   not successful ?? didn't match our net\n");

        return (NOROUTE);
}
```

```
/*
        File:  pup/setfilter.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Build and set a packet filter for a PUP channel.
        Returns:  Nothing.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "../include/pup.h"

pupSetFilter (Pchan, Fpuptype, Fdsthost, Fdstsocket)
PUPCHAN *Pchan;                         /* pup channel descriptor */
unsigned char *Fpuptype;                /* filter on pup type */
unsigned char *Fdsthost;                /* filter on destination host */
unsigned long *Fdstsocket;              /* filter on destination socket */
{
        FILTER f;                       /* build filter here */

        /*
        *  Always filter on ethernet packet type PUP
        */
        f.fwhich.do_enetpacket = TRUE;
        f.fdata.f_enetpacket = PupPacket;

        if (Fpuptype)                   /* filter on PUP type ? */
                {
                f.fwhich.do_puptype = TRUE;
                f.fdata.f_puptype   *Fpuptype;
                }
        else
                f.fwhich.do_puptype = FALSE;

        if (Fdsthost)                   /* filter on destination host ? */
                {
                f.fwhich.do_dsthost = TRUE;
                f.fdata.f_dsthost   *Fdsthost;
                }
        else
                f.fwhich.do_dsthost = FALSE;

        if (Fdstsocket)                 /* filter on destination socket ? */
                {
                f.fwhich.do_dstsocket = TRUE;
                f.fdata.f_dstsocket = *Fdstsocket;
                }
        else
                f.fwhich.do_dstsocket = FALSE;

        spfun (fdown (Pchan->fid), SetMinorFilter, &f);

        return;
}
```

```
/*
        File:  pup/write.c
        Date:  February 15 1984
        Author:  Mark Van Dellon
        Purpose:  Packetize a buffer and send through pup channel.
        Returns:  OK.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "../include/pup.h"

pupWrite (Pchan, Ptype, ID, buf, buflen)
PUPCHAN *Pchan;                         /* pup write channel */
unsigned char Ptype;                    /* pup type (for header) */
unsigned long ID;                       /* pup ID (for header) */
char *buf;                              /* pup data */
int buflen;                             /* size of data buffer */
{
        TRANSFRAME enp;                 /* build packet in here */
        register round;

        round = (buflen & 0x01) ? buflen + 1 : buflen;

        enp.OutPup.PupLength = PUPPACKOVER + buflen;
        enp.OutPup.PupTransport = 0;    /* gateway byte 0 @ source */
        enp.OutPup.PupType = Ptype;
        enp.OutPup.PupID = ID;
        pupPortCopy (&Pchan->DstPort, &enp.OutPup.PupDst);
        pupPortCopy (&Pchan->SrcPort, &enp.OutPup.PupSrc);

        if (buflen)
                movebytes (buf, &enp.OutPup.PupData, round);

        if (Pchan->mode & PCM_WCHECKSUM)        /* checksum ? */
                *((unsigned short *)&enp.OutPup.PupData[round]) =
                        pupChecksum (&enp.OutPup, PUPPACKOVER + round   2);
        else
                *((unsigned short *)&enp.OutPup.PupData[round]) =
                        NOCHKSUM;

        if (PUPdebug)
                pupPrintPacket (&enp.OutPup, 'W');

        return (enwrite (&Pchan->ImmHost, PupPacket, Pchan->fid, &enp,
                        round + PUPPACKOVER));
}

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

Encapsulate a PUP with 10Mbit header and write to a host.

*/

static int enwrite (DstHost, FrmType, fptr, frame, length)
unsigned char *DstHost;                 /* ethernet destination address */
```

```
unsigned short FrmType;                    /* ethernet frame type */
FILE fptr;                                 /* file pointer for write */
TRANSFRAME *frame;                         /* data to write */
int length;                                /* encapsulated packet length */
{

        movebytes (OstHost, &frame->OutOstHost, 6);
        frame->OutFrameType - FrmType;

        filewrite (fptr, frame, length + TRANSOVER);

        return (OK);
}
```

```
/*
        File:  bsp/queue.h
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Defines and structures for BSP queues.

        MODIFIED BY     REASON
*/

typedef char (*QLINK);

typedef struct
        {
        QLINK *Fitem;           /* first item pointer */
        QLINK *Litem;           /* last item pointer */
        } QUEUE;

#define QInit(q)                (q.Fitem = NULL)
#define QEmpty(q)               (q->Fitem == NULL)
```

```
/*
        File:  bsp/abort.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Abort a BSP connection by sending an abort packet
                  signifying termination of connection.
        Returns:  Result of closing port.

        MODIFIED BY     REASON
*/

BSPabort (code, string)
int code;                       /* abort code for packet */
char *string;                   /* text for packet */
{
        char buffer[MAXPUPDATALEN];

        /*
        *  Packet format is two bytes for abort code followed abort text.
        */
        *(short *)(buffer) - code;
        movebytes (string, buffer + 2, strlen (string));

        if (UserChannel.fid != 0)
                pupWrite (&UserChannel, RTP_ABORT, ConnectionID, buffer,
                                                        strlen (string)+2);
        if (ConnectionChannel.fid != 0)
                pupWrite (&ConnectionChannel, RTP_ABORT, ConnectionID, buffer,
                                                        strlen (string)+2);

        /*
        *  Now self destruct by closing our connections.
        */
        return (BSPclose (OK));
}
```

```
/*
        File:  bsp/ack.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Handle Read and Write of acknowledges.
        Returns:  Nothing (either routine).

        MODIFIED BY     REASON
*/

BSPrdACK (pbip)
register PBI *pbip;                     /* ACK packet buffer item */
{
        register PBI *ckpbip;
        register int i;
        register int needA;             /* need acknowledgment flag */
        register int totalbytes;        /* total bytes reader allows */

        /*
        *  Check that the ACK ID is the desired ACK and not a delayed
        *  duplicate.  ACK ID's can be valid in either of two ways:
        *  1.  The current pup's ID is greater than the last ACK ID,
        *  2.  The current pup's ID is equal to the last ACK ID and
        *      the last ACK ID had zero allocation.
        */
        if (pbip->pbi_id > lastAckID || (pbip->pbi_id == lastAckID &&
                                         pupnumAlloc <= 0))
                {
                lastAckID = pbip->pbi_id;

                /*
                *  Free any packets being held for retransmission that
                *  have been acknowledged.  The ACK PUP indicates all
                *  bytes previous to the PUP ID have been received correctly.  .
                */
                while (unackedPups > 0)
                        {
                        ckpbip = QGet (&wtq);           /* get packet */
                        if (ckpbip->pbi_id + ckpbip->pbi_cnt <= lastAckID)
                                {
                                unackedPups--;
                                unackedBytes -= ckpbip->pbi_cnt;
                                QForward (&freeq, ckpbip);
                                }
                        else
                                {
                                QReverse (&wtq, ckpbip);
                                break;
                                }
                        }

                /*
                *  The ACK PUP carries a 3-word block indicating:
                *  1) maximum bytes per PUP.
                *  2) number of PUPs.
                *  3) total number of bytes receiver can accept.
                *
```

```
                *  Figure new allocations receiver has available.
                *  If for a weird reason 1 * 2 > 3 (the numbers
                *  from above) we compensate by decreasing the
                *  number of pups (2) until the equation is true.
                *  This has to work....think about it.
                */
                pupsizeAlloc = min (MAXPUPDATALEN,
                                    ((short *) &pbip->pbi_buf)[0]);
                pupnumAlloc = noneg (((short *) &pbip->pbi_buf)[1] -
                                    unackedPups);
                totalbytes = noneg (((short *) &pbip->pbi_buf)[2] -
                                    unackedBytes);

                while (pupsizeAlloc * pupnumAlloc > totalbytes)
                        pupnumAlloc--;

                if (BSPdebug)
                        printf ("BSP got ACK, ID 0x%08x, PupAlloc %d\n",
                                        lastAckID, pupnumAlloc);

                /*
                *  Retransmit unacknowledged data, if any.  There will
                *  be enough room in receiver otherwise these packets
                *  would not have been sent previously.  However the
                *  last packet may need an acknowledgement.
                */
                for (i   0; i < unackedPups; i++)
                        {
                        pbip   QGet (&wtq);
                        needA = (pupnumAlloc <= 0 && i == unackedPups);
                        switch (pbip->pbi_type)
                                {
case BSP_ADATA:
case BSP_DATA:                  pbip->pbi_type = needA ? BSP_ADATA : BSP_DATA;
                                break;
                                }

                        if (BSPdebug)
                                printf ("BSP retransmit %d / %d, ID 0x%08x\n",
                                        i + 1, unackedPups, pbip->pbi_id);
                        writeMACRO (&UserChannel, pbip);
                        QForward (&wtq, pbip);
                        }

                /*
                *  Determine if still blocked.
                */
                wtblocked = pupsizeAlloc * pupnumAlloc <= 0;
                }
        else
                if (BSPdebug)
                        printf ("BSP ACK ID (0x%08x) <= lastAckID (0x%08x)\n",
                                        pbip->pbi_id, lastAckID);

        return;
}
```

```
/*************************************************************************

Acknowledge to the sender where we think the receiver byte stream is
by sending the receiver byte ID, and our next allocation of how many
packets the receiver can accept.

*/

BSPwtACK ()
{
        short buf[3];                           /* allocation to send */
        register PBI *dragptr;
        register PBI *qp;

        /*
        *  Flush received data packets we are not acknowledging since
        *  we know they will be retransmitted.  This keeps our read
        *  buffers from filling up.  (Optional, since the sequencing
        *  algorithm will discard any duplicate pups).
        */
        for (dragptr = &rdq, qp   rdq.Fitem; qp != NULL; )
                {
                if (qp->pbi_id > ReceiverByteID)
                        {                       /* remove packet */
                        if (BSPdebug)
                                printf ("BSP flush, ID: 0x%08x\n",
                                                qp->pbi_id);
                        dragptr->Fitem = qp->Fitem;
                        QForward (&freeq, qp);
                        qp = dragptr->Fitem;
                        unreadPups--;
                        }
                else
                        {                       /* keep packet */
                        dragptr   qp;
                        qp   qp->Fitem;
                        }
                }

        /*
        *  Compute allocation we now have available and send to writer.
        */
        buf[0]   BSPSIZE;                       /* max bytes per pup */
        buf[1]   READPBIS - unreadPups;         /* number of pups */
        buf[2] = buf[0] * buf[1];               /* total bytes available */

        if (BSPdebug)
                printf ("BSP ACK sent, ID: 0x%08x, PupAlloc: %d\n",
                                        ReceiverByteID, buf[1]);

        pupWrite (&UserChannel, BSP_ACK, ReceiverByteID, buf, sizeof (buf));

        ackPending   FALSE;                     /* show ACK was sent */

        /*
        *  Check if zero bytes in allocation was sent.  If it was, when
        *  we get allocations available (from read) we will send a
```

```
        *  gratuity acknowledge to startup writer.
        */
        sentzeroAlloc   buf[2] <= 0;

        return;
}
```

```
/*
        File:  bsp/bsp.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Define data common to all bsp modules and pull all
                  modules together for compilation.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include <signal.h>
#include "../include/bsp.h"
#include "queue.h"

/*****************************************************************************/
/*                                                                           */
/*      Channels for communication to ethernet.  "ConnectionChannel"         */
/*      is used for a listening server and "UserChannel" is used for         */
/*      connected processes in transmitting or receiving data.               */

PUPCHAN ConnectionChannel;
PUPCHAN UserChannel;

PORT LocalPort;         /* user process port */
PORT RemotePort;                /* server process port */


/*****************************************************************************/
/*                                                                           */
/*      PUP ID's for BSP.  Each is initialized to the connection             */
/*      ID and are incremented accordingly for the type of PUP               */
/*      received (Interrupt, Receiver, and Sender).                          */

static long ConnectionID;       /* remembered for termination protocol */
static long InterruptID;        /* ID of next incoming interrupt */
static long ReceiverByteID;     /* first byte beyond what I have acked */
static long SenderByteID;       /* ID of next outgoing packet created */


/*****************************************************************************/
/*                                                                           */
/*      Structure of a PBI (packet buffer item) for BSP                      */

#define BSPSIZE         MAXPUPDATALEN
struct PacketBufferItem
        {
        struct PacketBufferItem *qlink; /* queue link */
        unsigned char pbi_buf[BSPSIZE]; /* the data */
        int pbi_cnt;                    /* actual data in packet */
        unsigned long pbi_id;           /* id of packet */
        char *pbi_rdnext;               /* used in read (partial remains) */
        unsigned char pbi_type;         /* pup type of packet */
        };
typedef struct PacketBufferItem PBI;

static QUEUE freeq = 0;                 /* Q of free packets */
```

```
/*****************************************************************************/
/*                                                                           */
/*                        BSP writer information                             */

#define ALLOCTIME       6 * ONESEC      /* read time to get allocation */
#define ALLOCWAIT       5               /* # of times to do ALLOCTIME */
#define WRITEPBIS       5               /* # of outstanding packets */

static PBI wtpbi[WRITEPBIS] = 0;        /* BSP write buffers */
static QUEUE wtq   0;                   /* Q of unacked packets */

static long lastAckID = 0;              /* PUP ID of last ACK packet */
static int pupnumAlloc = 0;             /* allocated number of pups */
static int pupsizeAlloc   0;            /* allocated bytes per pup */
static int unackedBytes   0;            /* outstanding bytes needing ACK */
static int unackedPups = 0;             /* outstanding PUPs needing ACK */
static int wtblocked = TRUE;            /* blocked for writting */

#define writeMACRO(chan, pp)    (pupWrite (chan, pp->pbi_type, pp->pbi_id, \
                                        pp->pbi_buf, pp->pbi_cnt))


/*****************************************************************************/
/*                                                                           */
/*                        BSP reader information                             */

#define KEEP            1               /* keep packet just read */
#define NOKEEP          0               /* discard packet just read */
#define READPBIS        5               /* packets capable of buffering */

static PBI rdpbi[READPBIS] = 0;         /* BSP read buffers */
static QUEUE rdq = 0;                   /* Q of unread packets */

static int ackPending = FALSE;          /* writer wants ACK */
static int sentzeroAlloc = FALSE;       /* sent writer zero allocation */
static int unreadPups   0;              /* # of unread PUPs */

#define readMACRO(chan, pp)     (pupRead (chan, &pp->pbi_buf, &pp->pbi_cnt, \
                                        &pp->pbi_type, &pp->pbi_id, \
                                        NULL, NULL))


/*****************************************************************************/
/*                                                                           */
/*                        BSP open data to keep around                       */

static long lastConnectionID = 0;       /* watches for duplicate RFC's */
static int listening = FALSE;           /* true when socket for RFC open */
static long serversocket = 0;           /* socket for new server process */


/*****************************************************************************/
/*                                                                           */
/*                        Watchdog information                               */
```

```
#define CLKRESOLUTION   3               /* frequency of watchdog (sec) */
static int server   FALSE;              /* true if this is the server
                                        *  portion of a connection.
                                        */

/*****************************************************************/
/*                                                               */
/*                          Miscellaneous                        */

#define MATCH           1

int BSPdebug = FALSE;           /* true gives you error diagnostics */


#include "abort.c"                      /* abort a BSP connection */
#include "ack.c"                        /* read / write ACK code */
#include "close.c"                      /* close a BSP connection */
#include "lookup.c"                     /* remote host name => net address */
#include "open.c"                       /* open a BSP connection */
#include "print.c"                      /* print a port (debug user) */
#include "queue.c"                      /* BSP queue routines */
#include "read.c"                       /* read BSP data */
#include "watchdog.c"                   /* watchdog timer code */
#include "write.c"                      /* write BSP data */

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

Support routines.

*/

static min (a, b)                       /* minimum of "a" and "b" */
int a, b;
{

        return (a < b ? a : b);
}

static noneg (a)                        /* assure poitive or zero */
int a;
{
        return (a < 0 ? 0 : a);
}
```

```
/*
        File:  bsp/close.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Close a BSP connection by the terminate protocol.
        Returns:  OK.

        MODIFIED BY     REASON
*/

BSPclose (mode)
int mode;                       /* mode: RTP_END, RTP_ENDR, or OK */
{
        char buf[MAXPUPDATALEN];        /* pending data read buffer */
        int bytes;                      /* pending data read count */
        unsigned long pupid;            /* RFC pup packet ID read */
        unsigned char puptype;          /* RFC pup type read */
        register int retry;

        BSPsettimer (OFF);
        switch (mode)
                {
case RTP_END:
                /*
                *  If data is available in BSP buffers examine it
                *  looking for an abort or an END sent from server.
                */
                for ( ; unreadPups > 0; unreadPups--)
                        {
                        switch (BSPread (buf, sizeof (buf), NULL, &bytes))
                                {
        case RTP_ABORT:
                                printf (" ...%s", buf + 2);
                                unreadPups = 0;
                                BSPclose (OK);
                                return (OK);
        case RTP_END:
                                unreadPups = 0;
                                BSPclose (RTP_ENDR);
                                return (OK);
                                }
                        }

                /*
                *  Initiate an end of a connection by sending an
                *  END PUP whose ID matches the Connection ID.
                *  Retransmit until a matching ENDREPLY PUP is
                *  received.  Upon receiving the ENDREPLY send an
                *  ENDREPLY and promptly self destruct.
                */
                pupSetTimeout (&UserChannel, ONESEC * 3);
                for (retry   0; retry < 3; retry++)
                        {
                        pupWrite (&UserChannel, RTP_END, ConnectionID,
                                                        NULL, 0);
                        if (pupRead (&UserChannel, NULL, NULL, &puptype,
                                                &pupid, NULL, NULL) == OK &&
```

```
                                        puptype == RTP_ENDR &&
                                        pupid == ConnectionID)
                        break;
                }
        pupWrite (&UserChannel, RTP_ENDR, ConnectionID,
                                        NULL, 0);
        pupClose (&UserChannel);
        return (OK);

case RTP_ENDR:
        /*
        *  Receiver of END PUP responds by returning an ENDREPLY
        *  PUP with matching ID, (Connection ID), and then dallying
        *  in order to respond to a retransmitted END PUP should
        *  its initial ENDREPLY be lost.  Upon receipt of the
        *  ENDREPLY self destruct.
        */
        pupSetTimeout (&UserChannel, ONESEC * 3);
        for (retry = 0; retry < 3; retry++)
                {
                pupWrite (&UserChannel, RTP_ENDR, ConnectionID,
                                                NULL, 0);
                if (pupRead (&UserChannel, NULL, NULL, &puptype,
                                        &pupid, NULL, NULL) == OK &&
                                        puptype == RTP_ENDR &&
                                        pupid == ConnectionID)
                        break;
                }

case OK:
        /*
        *  Close channels - the connection has already been
        *  broken (probably aborted or timed out).
        */
        pupClose (&UserChannel);
        pupClose (&ConnectionChannel);
        return (OK);

default:
        if (BSPdebug)
                printf ("BSPclose supplied with unknown mode: %d\n",
                                                        mode);
        return (OK);
        }

}
```

```
/*
        File:  bsp/lookup.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Setup network address for a given name or address in
                  the RemotePort structure.
        Returns:  Lookup NOTFOUND or OK.

        MODIFIED BY     REASON
*/

BSPlookup (name)
char *name;                     /* char string to convert to port addr */
{
        char *argp[3];
        extern char *astoi ();
        int noerror;
        int temp;
        char work[64];          /* working version of name string */

        /*
        *  Assume possible error in lookup of server host address.
        */
        sprintf (PUPErrorMsg, "%s: %s",
                        "Can't locate server host address for", name);

        /*
        *  If the name has number signs in it, the user has specified
        *  host via optional_net#host#optional_socket.
        */
        strcpy (work, name);            /* make working copy */
        if (find ('#', work))
                {
                breakline (work, '#', argp, sizeof (argp) / sizeof (char *));
                if (strlen (argp[0]) > 0)
                        {
                        if (*astoi (argp[0], &temp) != NULL)
                                return (NOTFOUND);
                        RemotePort.net = temp;
                        }
                else
                        RemotePort.net = 0;

                if (strlen (argp[1]) > 0)
                        {
                        if (*astoi (argp[1], &temp) != NULL)
                                return (NOTFOUND);
                        RemotePort.host   temp;
                        }
                else
                        return (NOTFOUND);

                if (strlen (argp[2]) > 0)
                        {
                        if (*astoi (argp[2], &temp) != NULL)
                                return (NOTFOUND);
                        RemotePort.socket = temp;
```

```
                }
        else
                RemotePort.socket = 0;

        return (OK);
        }
else
        {
        if (findline ("/etc/enetmappings", ';', work, 2, argp,
                        sizeof (argp) / sizeof (char *)) == MATCH)
                {
                noerror - (*astoi (argp[1], &temp) == NULL);
                free (argp[0]);
                free (argp[1]);
                free (argp[2]);
                if (noerror)
                        {
                        RemotePort.net   OurNetNumber;
                        RemotePort.host = temp;
                        RemotePort.socket = 0;
                        return (OK);
                        }
                }
        }

return (NOTFOUND);
}
```

```
/*
        File:  bsp/open.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Establish a PUP BSP connection with an exchange of
                  "Request for Connection" packets known as RFC's.
        Returns:  Error code or OK.

        MODIFIED BY     REASON
*/

BSPopen (mode, socket)
int mode;                       /* mode: Initiate, Listen, or Server */
unsigned long socket;           /* socket to use or NULL */
{
        unsigned char buffer[MAXPUPDATALEN];    /* used in reads */
        int length;                             /* used in reads */
        unsigned char myhost;                   /* used in filter */
        unsigned char puptype;                  /* used in reads */
        register int retry;
        unsigned char rfctype;                  /* used in filter */

        switch (mode)
                {
case Initiate:
                /*
                *  If socket argument specifed in function call is nonzero
                *  and destination port socket is not already known -
                *  fill it with socket argument.
                */
                if (socket && RemotePort.socket == 0)
                        RemotePort.socket   socket;

                /*
                *  Generate a unique connection ID to be used for the
                *  RFC socket which will remain throughout the rest of
                *  this connection.  Sync receiver, sender, and
                *  interrupt ID to connection ID.
                */
                ConnectionID = pupUniqueSocket () + getpid ();
                ReceiverByteID =
                SenderByteID
                InterruptID      ConnectionID;

                switch (pupOpen (&UserChannel, ConnectionID, &RemotePort))
                        {
        case OK:        break;

        case NOROUTE:   if (BSPdebug)
                                printf ("PUP routing failed\n");
                        return (NOROUTE);

        default:        return (NOCHAN);
                        }

                /*
                *  Set up to send an RFC containing our local port
```

```
            *  as data as specified by the RFC protocol.
            */
            pupSetMode (&UserChannel, PCM_WCHECKSUM | PCM_RCHECKSUM);
            pupSetTimeout (&UserChannel, ONESEC * 5);
            pupPortCopy (&UserChannel.SrcPort, &LocalPort);
            myhost   pupGetOurHost (&UserChannel);
            pupSetFilter (&UserChannel, NULL, &myhost, &LocalPort.socket);;;

            for (retry   0, puptype = NULL; retry < 3; retry++)
                    {
                    pupWrite (&UserChannel, RFC, ConnectionID,
                                    &LocalPort, sizeof (LocalPort));

                    if (pupRead (&UserChannel, buffer, &length,
                                    &puptype, NULL, NULL, NULL) == OK)
                            break;
                    }

            switch (puptype)
                    {
    case RFC:
                    /*
                    *  We have a port to connect to in server task,
                    *  now finish setting up pup channel, etc.
                    */
                    pupPortCopy (buffer, &RemotePort);
                    pupRoute (&UserChannel, &RemotePort);
                    pupSetMode (&UserChannel, PCM_WCHECKSUM |
                                                PCM_RCHECKSUM |
                                                PCM_IGNBADCKS);
                    pupSetTimeout (&UserChannel, ONESEC);
                    server = FALSE;
                    if (BSPdebug)
                            {
                            printf ("BSP user [%s] -> ",
                                            printPort (&LocalPort));
                            printf ("[%s]\n", printPort (&RemotePort));
                            }
                    BSPChannelInit ();
                    return (OK);

    case RTP_ABORT:
                    movebytes (buffer + 2, PUPErrorMsg, length - 2);
                    BSPclose (OK);
                    return (RTP_ABORT);

    case ERRORPUP:
                    movebytes (buffer + 24, PUPErrorMsg, length - 24);
                    BSPclose (OK);
                    return (RTP_ABORT);

    default:
                    return (TIMEOUT);
                    }
            break;

case Listen:
```

```
/*
*  Here we just wait for RFC's on our socket (which
*  is usually a well known socket) and return when
*  we get one.  Note, we only open our connection
*  listening port once.
*/
if (listening == FALSE)
        {
        if (pupOpen (&ConnectionChannel, socket, NULL))
                return (NOCHAN);

        pupPortCopy (&ConnectionChannel.SrcPort, &LocalPort);

        myhost   pupGetOurHost (&ConnectionChannel);
        rfctype = RFC;
        pupSetFilter (&ConnectionChannel, &rfctype, &myhost,
                                                &socket);

        pupSetMode (&ConnectionChannel,
                        PCM_WCHECKSUM | PCM_RCHECKSUM);
        pupSetTimeout (&ConnectionChannel, INFINITY);
        listening = TRUE;       /* show channel open */
        }

while ()                /* sit and listen forever */
        {
        if (pupRead (&ConnectionChannel, buffer, NULL, NULL,
                        &ConnectionID, NULL, NULL) !=OK)
                continue;

        pupPortCopy (buffer, &RemotePort);

        ReceiverByteID =
        SenderByteID   =
        InterruptID      ConnectionID;

        if (ConnectionID != lastConnectionID)
                {
                /*
                *  We've got a new connection return with
                *  it, after generating a suitably random
                *  number for our connection socket.
                */
                pupRoute (&ConnectionChannel, &RemotePort);
                serversocket = pupUniqueSocket ();
                LocalPort.socket - serversocket;
                lastConnectionID = ConnectionID;
                return (RFC);
                }

        /*
        *  A duplicate connection ID.  While this should
        *  not happen it could if our previously returned
        *  RFC got lost or misunderstood so we have to be
        *  able to handle it.  Send back an RFC containing
        *  our last local port.
        */
```

```
                        if (BSPdebug)
                                printf("Duplicate RFC - %s: 0x%08x\n",
                                        "resending ID", ConnectionID);
                        pupWrite (&ConnectionChannel, RFC, ConnectionID,
                                        &LocalPort, sizeof (LocalPort));
                        }

case Server:
                /*
                *  The listener has spawned off a server process.
                *  Open a new ethernet port and reply to the RFC
                *  with our port information.
                */
                pupClose (&ConnectionChannel);
                switch (pupOpen (&UserChannel, serversocket, &RemotePort))
                        {
        case OK:        break;

        case NOROUTE:   return (NOROUTE);

        default:        return (NOCHAN);
                        }

                ReceiverByteID
                SenderByteID    =
                InterruptID     = ConnectionID;

                pupWrite (&UserChannel, RFC, ConnectionID, &LocalPort,
                                                        sizeof (LocalPort));

                myhost = pupGetOurHost (&UserChannel);
                pupSetFilter (&UserChannel, NULL, &myhost, &serversocket);
                pupSetMode (&UserChannel, PCM_WCHECKSUM | PCM_RCHECKSUM |
                                        PCM_IGNBADCKS);
                pupSetTimeout (&UserChannel, ONESEC);

                server = TRUE;
                if (BSPdebug)
                        {
                        printf ("BSP server [%s] -> ", printPort(&LocalPort)); ;
                        printf ("[%s]\n", printPort(&RemotePort));
                        }
                BSPChannelInit ();
                return (OK);
                }

        return (OK);
}

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

Initialization in common between Initiated and Server connection opens.

*/

BSPChannelInit ()
```

```
{
        int i;

        /*
        *  Do the queues and put all available packets on the
        *  free Q.
        */
        QInit (freeq);
        QInit (rdq);
        QInit (wtq);
        for (i = 0; i < READPBIS; i++)
                QForward (&freeq, &rdpbi[i]);
        for (i = 0; i < WRITEPBIS; i++)
                QForward (&freeq, &wtpbi[i]);

        /*
        *  Do the writter variables.
        */
        lastAckID   ConnectionID - 1;
        pupnumAlloc = pupsizeAlloc   0;
        unackedBytes - unackedPups   0;
        wtblocked   TRUE;

        /*
        *  Do the reader variables.
        */
        ackPending = sentzeroAlloc   FALSE;
        unreadPups = 0;

        /*
        *  Send the other host our current allocation.
        */
        BSPwtACK ();

        /*
        *  Start watchdog timer.
        */
        BSPsettimer (ON);

        return;
}
```

```
/*
        File:  bsp/print.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Format port for output.
        Returns:  Pointer to formated string.

        MODIFIED BY     REASON
*/

char *printPort (port)
PORT *port;                             /* net, port and socket */
{
        static char pps[25];

        sprintf (pps, "%d#%d#0x%08x", port->net, port->host, port->socket);

        return (pps);
}
```

```
/*
        File:  bsp/queue.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Functions for maintaining BSP queue organization.

        MODIFIED BY     REASON
*/

QForward (q, item)                      /* place on end of queue */
register QUEUE *q;                      /* queue head */
QLINK *item;                            /* new data for queue */
{
        *item = NULL;                   /* new item pointer */
        if (q->Fitem == NULL)           /* q empty */
                q->Litem = q;
        *(q->Litem) = item;             /* forward link */
        q->Litem = item;                /* backward link */
        return;
}

QReverse (q, item)                      /* place on front of queue */
register QUEUE *q;                      /* queue head */
QLINK *item;                            /* new data for queue */
{
        *item = q->Fitem;               /* point to old first item */
        q->Fitem = item;                /* head points to me */
        if (*item == NULL)              /* only element on q */
                q->Litem = item;
        return;
}

QGet (q)                                /* take from front */
register QUEUE *q;                      /* queue head */
{
        QLINK *tmp;                     /* data from queue to return */

        if (q->Fitem != NULL)           /* queue empty ? */
                {
                tmp = q->Fitem;         /* this is what we return */
                q->Fitem = *tmp;        /* point over data returned */
                return (tmp);
                }
        return (NULL);
}

QAfter (q, predecessor, item)           /* insert after predecessor */
QUEUE *q;                               /* queue head */
QLINK *predecessor;
QLINK *item;                            /* data to insert */
{
        register QLINK *tmp;

        for (tmp = q; tmp != NULL; tmp = *tmp)
                {
                if (tmp == predecessor)
                        {
```

```
                    *item = *tmp;    /* new data points to next */
                    *tmp   item;     /* previous points to new data */
                    return;
                    }
                }
        return;
}
```

```
/*
        File:  bsp/read.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Read some bytes from a BSP connection.
        Returns:  PupType or TIMEOUT.

        MODIFIED BY     REASON
*/

BSPread (userbuffer, len, timeout, size)
char *userbuffer;
int len;                                /* maximum length of the buffer */
short timeout;                          /* clock ticks to wait */
int *size;                              /* gets the returned size */
{
        register int amount;
        register PBI *pbip;
        register unsigned char puptype;

        *size   0;                      /* return length */

        /*
        *  If no data is available from the BSP buffers do a timeout
        *  read for a packet, because we need one right now.  Be careful
        *  to assure we receive a packet for the user and not an ADATA
        *  with null length requesting allocation information.
        */
        while (unreadPups == 0)
                {
                pupSetTimeout (&UserChannel, timeout);
                if (BSPreceive () != OK)
                        return (TIMEOUT);
                }

        /*
        *  If there is data waiting, read all available packets or
        *  until BSP buffers full.
        */
        while (pupAvailable (&UserChannel) > 0 && unreadPups < READPBIS)
                BSPreceive ();

        /*
        *  Transfer any pending packets from BSP buffers to user buffers
        *  providing that the pending packet has been sequenced.... a
        *  packet could have gotten lost in transmission and we're waiting
        *  for the retransmit now.  Multiple pending DATA packets may be
        *  returned together while all others (MARK, RTP, ABORT) packets
        *  must be returned one at a time.
        */
        while (unreadPups > 0 && *size < len)
                {
                pbip = QGet (&rdq);             /* get packet */
                if (*size == 0)                 /* first time */
                        puptype   pbip->pbi_type;
                else if (puptype != pbip->pbi_type || puptype != BSP_DATA)
                        {
```

```
                QReverse (&rdq, pbip);
                break;
                }

        if ((puptype == BSP_DATA || puptype == BSP_MARK) &&
                                pbip->pbi_id > ReceiverByteID)
                {                       /* skip in sequence */
                QReverse (&rdq, pbip);
                break;
                }

        /*
        *  Compute amount to move to user's buffer and do move.
        */
        amount   min (pbip->pbi_cnt, len - *size);
        if (BSPdebug)
                printf ("BSP rd - ID: 0x%08x, cnt: %d (full %s)\n",
                        pbip->pbi_id, amount, pbip->pbi_rdnext ==
                        pbip->pbi_buf ? "TRUE" : "FALSE");
        movebytes (pbip->pbi_rdnext, userbuffer + *size, amount);

        /*
        *  Check if copied entire packet to user buffer.  If
        *  partial, then update pointers accordingly.
        */
        if (amount < pbip->pbi_cnt)
                {                                   /* partial used */
                pbip->pbi_cnt -= amount;
                pbip->pbi_rdnext += amount;
                QReverse (&rdq, pbip);
                }
        else
                {                                   /* entire used */
                unreadPups--;
                QForward (&freeq, pbip);
                }

        *size += amount;                /* update amount in buffer */
        }

    /*
    *  If previous allocation was zero, send a gratuity allocation
    *  since we should now have space available (the reader just
    *  took some).
    */
    if (sentzeroAlloc && unreadPups != READPBIS)
        BSPwtACK ();

    return (puptype);
}

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

One common place to get a packet.  Used by read, write, and watchdog.

*/
```

```
BSPreceive ()
{
        register PBI *pbip;

        if ((pbip = QGet (&freeq)) != NULL)
                {
                if (readMACRO (&UserChannel, pbip) == OK)
                        {
                        pbip->pbi_rdnext = &pbip->pbi_buf;
                        if (BSPprocessrd (pbip) == KEEP)
                                unreadPups++;
                        else
                                QForward (&freeq, pbip);

                        if (ackPending)
                                BSPwtACK ();

                        return (OK);
                        }

                QForward (&freeq, pbip);
                }

        return (TIMEOUT);
}

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

Process a packet just read based on pup type.

*/

BSPprocessrd (pbip)
register PBI *pbip;
{
        register PBI *qp;
        int result;

        switch (pbip->pbi_type)
                {
case BSP_ADATA:
case BSP_AMARK:
                ackPending = TRUE;
case BSP_DATA:
case BSP_MARK:
                /*
                *  Place DATA and MARK bytes in stream available for
                *  upperlevel program providing the PUP ID is in correct
                *  sequence. They increment the receiver byte ID sequence
                *  by the packet length.
                */
                if (result = BSPsequence (&rdq, pbip))
                        {
                        switch (pbip->pbi_type)
                                {
        case BSP_ADATA:         pbip->pbi_type = BSP_DATA;      break;
        case BSP_AMARK:         pbip->pbi_type   BSP_MARK;      break;
```

```
                        }

                    /*
                    *  Check if receiver byte ID can be incremented.
                    */
                    for (qp   (PBI *)&rdq; qp->Fitem != NULL; )
                            {
                            qp = qp->Fitem;
                            if (qp->pbi_id == ReceiverByteID)
                                    ReceiverByteID += qp->pbi_cnt;
                            }

                    BSPsettimer (ON);
                    }

                return (result !  FALSE ? KEEP : NOKEEP);
                break;

case BSP_ACK:
                BSPrdACK (pbip);
                break;

case RTP_END:
                /*
                *  Remote host desires close of connection.  Place at
                *  of end of BSP queue so any pending packets get
                *  processed first.
                */
                if (pbip->pbi_id == ConnectionID)
                        {
                        QForward (&rdq, pbip);
                        return (KEEP);
                        }
                break;

case RTP_ABORT:
                /*
                *  Remote host is aborting connection due to a
                *  catastrophic error.  Place a beginning of queue
                *  for next packet to get picked up.
                */
                if (pbip->pbi_id == ConnectionID)
                        {
                        pbip->pbi_buf[pbip->pbi_cnt++] = NULL;
                        QReverse (&rdq, pbip);
                        return (KEEP);
                        }
                break;

case BSP_INTR:
                /*
                *  NOTE:  No known software implements this !!
                *  Acknowledge with and InterruptReply PUP only if its
                *  ID is equal to or one less than the current Interrupt
                *  ID.  If ID match current Interrupt ID signal and advance
                *  interrupt ID.  If one less than the current Interrupt ID
                *  it is a duplicate and should only be acknowledged.
```

```
                */
                if (pbip->pbi_id < InterruptID - 1)
                        break;
                pupWrite (&UserChannel, BSP_INTRR, InterruptID, NULL, 0);
                InterruptID++;
                break;

case BSP_INTRR:
                if (BSPdebug)
                        printf ("[received an interrupt reply!]\n");
                break;
                }

        return (NOKEEP);
}

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

Insert packets in sequence providing:
        1. The received PUP ID is greater than the receiver byte ID.
        2. The received packet has data (nonzero length).
        3. The received PUP ID is not a duplicate.

*/

BSPsequence (q, pbip)
QUEUE *q;
register PBI *pbip;
{
        register PBI *qp;

        if (pbip->pbi_id >= ReceiverByteID && pbip->pbi_cnt > 0)
                {
                /*
                *  The highest probabilty is that the packet
                *  belongs on the end of queue.  Check there first.
                */
                if (QEmpty (q) || q->Litem->pbi_id < pbip->pbi_id)
                        {
                        QForward (q, pbip);
                        if (BSPdebug)
                                printf ("BSP rdQ, ID: 0x%08x, cnt: %d\n",
                                        pbip->pbi_id, pbip->pbi_cnt);
                        return (TRUE);
                        }

                for (qp = (PBI *)q; qp->qlink != NULL; qp = qp->qlink)
                        {
                        if (pbip->pbi_id < qp->qlink->pbi_id)
                                {
                                QAfter (q, qp, pbip);
                                if (BSPdebug)
                                        printf ("BSP rdQ, ID: %x, cnt: %d\n",
                                                pbip->pbi_id, pbip->pbi_cnt);
                                return (TRUE);
                                }
                        if (pbip->pbi_id == qp->qlink->pbi_id)
```

```
                                {
                                if (BSPdebug)
                                        printf ("BSP dup, ID: 0x%08x\n",
                                                pbip->pbi_id);
                                return (FALSE);
                                }
                        }
                }

        return (FALSE);
}
```

```
/*
        File:  bsp/watchdog.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Handle watchdog timer.  Watchdog functions in
                  retransmission algorithm.
        Returns:  Nothing.

        MODIFIED BY     REASON
*/

BSPsettimer (state)
int state;                      /* TRUE to enable timer */
{
        extern int BSPwatchdog ();

        /*
        *  Check if this is a disable timer command.  We have to
        *  do this when using any form of shell commands.
        */
        if (state == OFF)
                alarm (0);
        else
                {
                signal (SIGALRM, BSPwatchdog);
                alarm (CLKRESOLUTION);
                }
}

BSPwatchdog ()
{
        while (pupAvailable (&UserChannel) > 0 && unreadPups < READPBIS)
                BSPreceive ();

        if (wtblocked)
                {
                if (BSPdebug)
                        printf ("BSP watchdog requesting ACK, ID 0x%08x\n",
                                                        SenderByteID);
                pupWrite (&UserChannel, BSP_ADATA, SenderByteID, NULL, 0);
                }

        BSPsettimer (ON);
        return;
}
```

```
/*
        File:  bsp/write.c
        Date:  March 15 1984
        Author:  Mark Van Dellon
        Purpose:  Write some bytes to a PUP BSP stream.
        Returns:  Errors code or OK.

        MODIFIED BY     REASON
*/

BSPwrite (type, userbuffer, len)
int type;                               /* pup type */
char *userbuffer;                       /* buffer of data to send */
int len;                                /* length of buffer */
{
        register int i;
        register int needA;             /* need ACK flag */
        register PBI *pbip;             /* pointer to current PBI*/
        register int sofar;             /* number transmitted so far */

        /*
        *  Send requested data.  If necessary, split data apart into
        *  maximum packet size as specified by current allocation.
        */
        for (sofar   0; sofar < len; )
                {
                /*
                *  If blocked to send (reader has no space) then read to
                *  get an allocation packet until timeout.  The watchdog
                *  timer will retransmit a request for allocation if other
                *  host previously missed it.  We need to know the size of
                *  each PUP and the number of PUPs the reader can accept.
                */
                if (wtblocked)
                        {
                        for (i = 0; i < ALLOCWAIT && wtblocked; i++)
                                {
                                pupSetTimeout (&UserChannel, ALLOCTIME);
                                BSPreceive ();
                                }
                        if (wtblocked)
                                {
                                BSPsettimer (OFF);
                                return (TIMEOUT);
                                }
                        }

                pbip = QGet (&freeq);
                pbip->pbi_cnt = (len - sofar > pupsizeAlloc) ? pupsizeAlloc :
                                                        len - sofar;

                /*
                *  Save data for possible retransmit if receiving
                *  port never gets packet or packet gets garbled.
                */
                movebytes (userbuffer + sofar, &pbip->pbi_buf, pbip->pbi_cnt);
                sofar += pbip->pbi_cnt;
```

```
                /*
                *  Determine if acknowledge is needed after this
                *  transmission by checking current allocation or
                *  last write PBI.
                */
                needA   (pupnumAlloc <= 1 || unackedPups + 1 >= WRITEPBIS);
                switch (type)
                        {
case BSP_DATA:          pbip->pbi_type = needA ? BSP_ADATA : BSP_DATA; break;
case BSP_MARK:          pbip->pbi_type = needA ? BSP_AMARK : BSP_MARK; break;
case BSP_ADATA:
case BSP_AMARK:         pbip->pbi_type   type; needA   TRUE;           break;
default:                printf ("Bad Mark in BSPwrite: 0%o\n", type);
                        return (-100);
                        }

                pbip->pbi_id = SenderByteID;

                /*
                *  Update pointers for packet about to send.
                */
                SenderByteID += pbip->pbi_cnt;;
                pupnumAlloc--;                  /* reader alloc down */
                QForward (&wtq, pbip);          /* q for retransmit */
                unackedPups++;                  /* outstanding pup */
                unackedBytes += pbip->pbi_cnt;

                if (BSPdebug)
                        printf ("BSP wt - ID: 0x%08x, cnt: %d, need ACK %s\n",
                                        pbip->pbi_id, pbip->pbi_cnt,
                                        needA ? "TRUE" : "FALSE");
                writeMACRO (&UserChannel, pbip);

                /*
                *  Resync watchdog timer to current write.
                */
                BSPsettimer (ON);
                wtblocked   needA;              /* do we wait now ? */
                }

        return (OK);
}
```

```
/*
        File:  ftp/command.h
        Date:  April 15 1984
        Author:  Mark Van Dellon
        Purpose:  Information necessary to interface keyboard to
                  user task.

        MODIFIED BY     REASON
*/

#define Echo    1                       /* for the command interpreter */
#define Silent  0

typedef struct                          /* one for each command */
        {
        int cmd_num;                    /* unique number of chars needed */
        char *cmd_name;                 /* the actual string of characters */
        int (*cmd_proc) ();             /* procedure to be executed */
        int cmd_arg;                    /* argument for procdure */
        } CMDTABLE;

/*
*  tty input terminators.
*/
#define CR      0x0d
#define ESC     0x1b
#define LF      0x0a
```

```
/*
        File:  ftp/ftp.h
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  Definition of FTP mark bytes and error codes.

        MODIFIED BY     REASON
*/

#include "../include/bsp.h"

/*************************************************************************/
/*                                                                       */
/*              FTP mark bytes commands from <PUP>FTPSpecs.press         */

#define Timeout_mark            -1
#define Retrieve                1       /* Retrieve file from server */
#define Store                   2       /* Store file on server */
#define Yes                     3       /* Positive response to command */
#define No                      4       /* Negative response to command */
#define Here_is_the_file        5       /* File is now being transported */
#define EOC                     6       /* End of command */
#define Comment                 7       /* NOP - just display text */
#define Version                 8       /* Data is my version */
#define NewStore                9       /* NewStore file on server */
#define Directory               10
#define Here_is_property_list   11      /* File property list */
#define You_are_user            12
#define Delete                  14      /* Delete file on server */
#define Rename                  15      /* Rename file on server */

#define Largest_mark            15

/*************************************************************************/
/*                                                                       */
/*              FTP 'NO' code bytes from <PUP>FTPSpecs.press             */

#define Not_implemented         1       /* Last command unimplemented */
#define Illegal_user_name       16      /* Property list error */
#define Illegal_password        17      /* Property list error */
#define File_not_found          64      /* File access   not found */
#define Protection_violation    65      /* File access   denied */
#define Do_not_send             69      /* Dont send (user to server) */

/*************************************************************************/
/*                                                                       */
/*                              File data types                          */

#define Unknown         0               /* used by filetypes & EOL */
#define Binary          1
#define Text            2
#define Guess           3               /* let ftp figure type */

/*************************************************************************/
/*                                                                       */
/*  End of line convention codes.  Note these actually denote an action*/
/*  to be taken when we get a file of the given form.  The ethernet    */
```

```
/*  default if CR and the UNOS filesystem is LF.                        */

#define EOL_CR          1               /* change net CR to UNOS LF */
#define EOL_CRLF        2               /* delete CR */
#define EOL_Trans       3               /* Transmit as is, no change */
#define EOL_NL          4               /* change UNOS LF to net CR */
#define EOL_Add         5               /* add CRs before newlines */

/*********************************************************************/
/*                                                                   */
/*              Error Recovery flags for type of action to take      */

#define ABORT           1               /* send abort, close, exit */
#define CLOSE           2               /* close, exit */
#define NONE            3               /* exit */
#define PROTOCOL        4

/*********************************************************************/
/*                                                                   */
/*                      Results of a NetToDisk transfer              */

#define ABORTED         1               /* transfer was aborted */
#define MARKNO          2               /* completed but got a NO mark */
#define MARKYES         3               /* completed ok */

/*********************************************************************/
/*                                                                   */
/*                              Program exit codes                   */

#define GOODEXIT        0               /* normal exit */
#define BADEXIT         -1              /* error exit */
#define TIMEEXIT        -2              /* idle timeout exit */

/*********************************************************************/
/*                                                                   */
/*                              Miscellaneous                        */

#define FTPVersion      1               /* ftp protocol version */

#define Omit            400             /* Don't bother sending a code byte */

#define FAILURE         1
#define SUCCESS         0

#define NEWFILE         10
#define OLDFILE         20

#define BUFSIZE         MAXPUPDATALEN

#define BLANK   " "

extern int FTPdebug;                    /* true if in debug mode */

extern char *formatMark ();             /* format a mark code */
extern char *ScanPL ();                 /* Scan a property list */
```

```
/*
        File:  ftp/command.c
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  Routines to handle terminal for FTP user.

        MODIFIED BY     REASON
        05/31/84 mvd    Updated terminal modes for version 4 release of OS.
*/

#include <stdio.h>
#include <ctype.h>                      /* character type macros */
#include <ttymodes.h>                   /* tty modes */
#include <sys/stypes.h>                 /* types for file descriptor */
#include <sys/filedesc.h>               /* descriptor for filestatus call */
#include "command.h"                    /* our own definitions */

#define BACKSPACE       '\010'
#define MATCH           0

static TTYMODE savemodes = 0;           /* saved copy of ttymodes */

static unsigned char erase = 0;         /* tty erase char */
static unsigned char kill = 0;          /* tty kill line char */
static unsigned int term = 0;           /* terminator, stopped tty input */

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

Setup ttymodes for special input to allow noise text.

*/

SetTTY ()
{
        TTYMODE tty;                    /* working copy of ttymodes */
        erkl ttyek;                     /* working copy of erase/kill chars */

        spfun (fdown (stdin), GTTY, &savemodes);

        spfun (fdown (stdin), GTTY, &tty);
        tty.t_i8bit = FALSE;            /* dont allow 8th bit on input */
        tty.t_icrlf = FALSE;            /* dont map CR to LF on input */
        tty.t_irawedit = TRUE;          /* stop editting chars */
        tty.t_eof = FALSE;              /* disable cntl-D being EOF */
        tty.t_escape = FALSE;           /* disable ESC processing */
        tty.t_prctl = TRUE;             /* enable suspend and kill */
        tty.t_scrctl = TRUE;            /* enable screen cntl chars */
        tty.t_wakealpha = TRUE;         /* wakeup on alpha chars */
        tty.t_wakectl   TRUE;           /* wakeup on control chars */

        tty.t_ealpha = FALSE;           /* dont echo alpha chars */
        tty.t_ecrlf   FALSE;            /* dont echo LF as CR & LF */
        tty.t_ectl = FALSE;             /* dont echo cntl chars */
        tty.t_erawctl = FALSE;          /* dont echo ctnl literally */
        tty.t_etab - FALSE;             /* dont echo tab */

        tty.t_ocrlf - TRUE;             /* output NL as CR/LF */
```

```
        tty.t_octl   FALSE;             /* print ctnl as are */
        tty.t_oxtab = TRUE;             /* expand tabs on output */
        spfun (fdown (stdin), STTY, &tty);

        spfun (fdown (stdin), GERKL, &ttyek);
        erase = ttyek.erkl_erasech;     /* get erase editting char */
        kill   ttyek.erkl_linedelch;    /* get line delete editting char */

        return;
}

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

Restore ttymodes to original state.

*/

ResetTTY ()
{
        spfun (fdown (stdin), STTY, &savemodes);
        return;
}

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

Check and see if file is a tty.  Function performed is equivalent to
UNIX routine "isatty" but uses UNOS file descriptors.

*/

IsaTTY (file)
FILE file;                      /* tty device name */
{
        char *argp[4];
        FILEDESC fd;
        char ttydev[25];

        if (filestat (file, &fd) >= 0)
                {
                sprintf (ttydev, "%d/%d", fd.f_device.d_major,
                                          fd.f_device.d_minor);

                if (findline ("/etc/ttys", ':', ttydev, 3, argp, 4) >= 0)
                        {
                        free (argp[0]);         free (argp[1]);
                        free (argp[2]);         free (argp[3]);
                        return (TRUE);
                        }
                }
        return (FALSE);
}

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

Read characters from the terminal and do our own line editting.  This
routines allows us to output noise text since the ttymodes have been
doctored.
```

```
*/

ReadTTY (str, max, echo)
char *str;                                  /* will get string */
int max;                                    /* sizeof str string */
int echo;                                   /* true to echo */
{
        register char c;
        register int len;

        fflush (stdout);                    /* flush previous output */

        for (len = 0; len < max; )
                {
                c = getchar ();             /* get single character */

                if (c == erase || c == BACKSPACE)
                        {                   /* erase single char */
                        if (len)
                                {
                                if (echo)
                                        printf ("\010 \010");
                                len--;
                                str--;
                                }
                        continue;
                        }

                if (c == kill)
                        {                   /* erase line */
                        while (len)
                                {
                                if (echo)
                                        printf ("\010 \010");
                                str--;
                                len--;
                                }
                        continue;
                        }

                if (isspace (c) || iscntrl (c) || c == EOF)
                        {                   /* terminator detected */
                        term   c;           /* save it */
                        *str   NULL;
                        break;
                        }

                if (echo)
                        putchar (c);        /* echo a good character */

                *str++   c;                 /* put char in string */
                len++;                      /* only count valid chars */
                }

        return (len);
}
```

```
/*************************************************************************

Return character that terminated last tty input.

*/

TermTTY ()
{
        return (term);
}

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

Execute a command from the supplied command table after fulling expanding
the command line with noise text.

*/

DoCommand (table)
CMDTABLE *table;                        /* command table to search */
{
        int ambig;                      /* ambiguous or unique cmd flag */
        char cmd[128];                  /* string for user command input */
        register int len;               /* length user typed */
        register CMDTABLE *p;

        /*
        *  Get a command and assure that something valid is there.
        */
        if (TermTTY () == EOF)          /* EOF happens on script files */
                {
                printf ("----- EOF detected on input\n");
                return (EOF);
                }

        len = ReadTTY (cmd, sizeof (cmd), Echo);
        if (len < 0)
                return (NULL);

        if (matchn ("?", cmd, len))                     /* asking for help */
                {
                DoHelp (table);
                return (NULL);
                }

       for (p = table, ambig - FALSE; p->cmd_num; p++)
                {                                       /* scan the table */
                if (matchn (p->cmd_name, cmd, len) == FALSE)
                        continue;                       /* not a match */

                if (len >= p->cmd_num)
                        {                               /* found command */
                        printf ("%s", p->cmd_name+len); /* noise text */
                        (*(p->cmd_proc)) (p->cmd_arg);  /* execute the proc */
                        return (NULL);
                        }
```

```
                if (ambig++ == 0)                       /* print once */
                        printf ("\nNot unique   one of:\n");

                printf ("%s\n", p->cmd_name);
                }

        if (ambig == FALSE)
                printf ("   ??? WHAT ???\n");

        return (NULL);
}

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

Display the help for each command that exists in the command table.

*/

DoHelp (table)
CMDTABLE *table;                        /* command table */
{
        register CMDTABLE *p;

        printf (" - One of the following:\n");
        for (p = table; p->cmd_num; )
                {
                printf ("%-35s", p++->cmd_name);
                printf ("%-35s\n", p->cmd_num ? p++->cmd_name : "");
                }

        return;
}
```

```
/*
        File:  ftp/ftp.c
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  Common support routines for FTP server and user.

        MODIFIED BY     REASON

        DetermineEOL    - Returns the end of line convention specified
                          in a property list.
        DetermineType   - Returns the file type specified in a property list.
        expectMark        Expect the given mark.
        FixupEOL          Fixup end-of-line chars to match machine convention.
        formatMark        Return readable string name of a mark byte.
        FormatReply       Format a string for a reply.
        GuessType         Try to see if file type is binary (or text).
        match             Match two strings insensitive to case.
        matchn            Match "n" chars of two strings insensitive to case.
        readData        - Read data up to a mark byte.
        readMark        - Read up to a mark byte and return its value.
        readUntil       - Read a string until the given mark.
        ScanPL          - Scan a property list for a specified property
                          and return a pointer to its value.
        writeData       - Send a string.
        writeMark       - Send a mark byte.
*/

#include <sys/stypes.h>         /* types for file descriptor */
#include <sys/filedesc.h>       /* descriptor for file status command */
#include <ctype.h>
#include <stdio.h>
#include "ftp.h"

static char *MarkName[] =       /* Human readable names for mark bytes */
        {                       /* used primarily in debug */
        "Invalid Mark - 0",
        "Retrieve",
        "Store",
        "Yes",
        "No",
        "Here-is-File",
        "End-of-Command",
        "Comment",
        "Version",
        "NewStore",
        "Directory",
        "Here-is-PList",
        "You are user",
        "Invalid Mark   13",
        "Oelete",
        "Rename",
        };


int ServerFlag = FALSE;         /* true if we are a server */

#define WESENT (ServerFlag ? "S: " : "U: ")
```

```
#define WEGOT  (ServerFlag ? "U: " : "S: ")

/*****************************************************************************
Return the end of line convention specified in a property list.  The
valid properties are "CR" (CR is default) or "CRLF" or "Transparent".
*/

int DetermineEDL (propertyList)
char *propertyList;
{
        char *value;

        value = ScanPL (propertyList, "End-of-line-convention");

        if (match (value, "CRLF"))
                return (EOL_CRLF);

        if (match (value, "Transparent"))
                return (EOL_Trans);

        return (EOL_CR);                /* default */
}

/*****************************************************************************
Return the file type specified in a property list.  The valid properties
are "Text" or "Binary".  There is no default.
*/

int DetermineType (propertyList)
char *propertyList;
{
        char *value;

        value   ScanPL (propertyList, "Type");

        if (match (value, "Binary"))
                return (Binary);

        if (match (value, "Text"))
                return (Text);

        return (Unknown);
}

/*****************************************************************************
Return success or failure based upon receiving the mark byte expected.
*/

expectMark (markwant)
unsigned char markwant;
{

        if (readMark () == markwant)    /* desired mark read */
                return (SUCCESS);

         if (FTPdebug)
                printf ("But were expecting: [%s]\n", formatMark (markwant));
```

```
        return (FAILURE);               /* not what we wanted */
}

/*************************************************************************
Doctor end of line termination character as specified by the user
property list.  Return number of bytes in output buffer.
*/

int FixupEOL (from, to, convention, size)
register char *from;                            /* input buffer */
register char *to;                              /* output buffer */
int size;                                       /* # bytes in input buffer */
int convention;                                 /* EOL convention */
{
        char *savedto;                          /* for saving address */

        savedto   to;
        switch (convention)
                {
case EOL_CRLF:                                  /* CRLF to LF */
                for ( ; size > 0; size--)
                        {
                        if (*from != '\r' )
                                *to++ = *from++;
                        else
                                from++;
                        }
                break;

case EOL_Trans:                                 /* Transparent - no change */
                movebytes (from, to, size);
                to += size;
                break;

case EOL_NL:                                    /* LF to CR */
                for ( ; size > 0; size--, from++)
                        {
                        if (*from == '\n')
                                *to++ = '\r';
                        else
                                *to++ = *from;
                        }
                break;

case EOL_Add:                                   /* LF to CRLF */
                for ( ; size > 0; size--)
                        {
                        if (*from == '\n')
                                *to = '\r';
                        *to++   *from++;
                        }
                break;

case EOL_CR:                                    /* CR to LF */
                for ( ; size > 0; size--, from++)
                        {
```

```
                        if (*from == '\r')
                                *to++   '\n';
                        else
                                *to++ = *from;
                        }
                break;
                }

        return (to   savedto);                  /* return new size */
}

/*************************************************************************
Format and return a pointer to a text string for the given mark byte
code for primarily debug display.
*/

char *formatMark (markbyte)
unsigned char markbyte;
{
        static char stringName[50];

        if (markbyte <= Largest_mark)
                return (MarkName[markbyte]);

        if (markbyte == Timeout_mark)
                return ("Timeout");

        sprintf (stringName, "Unknown mark byte code: %d", markbyte);

        return (stringName);
}

/*************************************************************************
Format a string for a reply to a request and return a pointer to the buffer.
*/

char *FormatReply (reply, code)
char *reply;                            /* string with reply */
int code;                               /* text preceeded by a code byte */
{
        static unsigned char buf[MAXPUPDATALEN];

        if (code)
                {
                if (FTPdebug)
                        sprintf (buf, "<%d> %s", reply[0], reply+1);
                else
                        sprintf (buf, "%s", reply+1);
                }
        else
                sprintf (buf, "%s", reply);

        if (buf[strlen (buf)   1] != '\n')
                strcat (buf, "\n");

        return (buf);                   /* returns the string */
}
```

```
/*************************************************************************
Try to guess the type of the given file, if file is not an EVENTCOUNT,
DEVICE or DIRECTORY, by reading the first few blocks and if any of the
bytes have the high-order bit set the file is considered binary, otherwise
it is considered text.
*/

int GuessType (filename)
char *filename;                         /* pointer to sting with filename */
{
        unsigned char buffer[2048];
        register int bytesread;
        FILEDESC fd;
        FILE fp;
        register unsigned char *ptr;

        if ((fp = fileopen (filename, FORREAD)) != NULL)
                {
                if (filestat (fp, &fd) >= 0 && fd.f_type != ECFILE &&
                    fd.f_type != BUFDEV && fd.f_type != UNBUFDEV &&
                    (fd.f_flags & FISADIR) != FISADIR)
                        {
                        if ((bytesread = fileread (fp, buffer,
                                                   sizeof (buffer))) > 0)
                                {
                                fclose (fp);
                                for (ptr = buffer; ptr < buffer + bytesread; )
                                        if (*ptr++ & 0x80)      /* msb set */
                                                return (Binary);

                                return (Text);
                                }
                        }
                fclose (fp);
                }
        return (Unknown);
}

/*************************************************************************
Compare two NULL terminated strings for match insenstive to character case.
Return TRUE if match.  (Similar to string routine strcmp).
*/

match (s1, s2)
register char *s1;
register char *s2;
{
        while (tolower (*s1) == tolower (*s2++))
                {
                if (*s1++ == NULL)
                        return (TRUE);
                }
        return (FALSE);
}

/*************************************************************************
```

```
Compare 'n' characters from two NULL terminated strings for match insenstive
to character case.  Return TRUE if match.  (Similar to string routine strncmp)
*/

matchn (s1, s2, n)
register char *s1;
register char *s2;
register int n;
{
        while (--n >= 0 && tolower (*s1) == tolower (*s2++))
                {
                if (*s1++ == NULL)
                        return (TRUE);
                }
        return (n < 0 ? TRUE : FALSE);
}

/*****************************************************************************
Read the next data packet into the buffer supplied and return only if the
packet is DATA.  Otherwise, it is a protocol error and invoke recovery.
*/

readData (buf, bufsiz)
char *buf;                              /* buffer for data */
int bufsiz;                             /* length of buffer */
{
        int bytes;                      /* # of bytes read from BSP */
        int puptype;

        puptype = BSPread (buf, bufsiz, ONESEC * 30, &bytes);
        buf[bytes] = NULL;              /* terminate with a null */
        switch (puptype)
                {
case BSP_DATA:
                if (FTPdebug)
                        printf ("%s%s", WEGOT, FormatReply (buf, TRUE));
                return;

case TIMEOUT:
                if (FTPdebug)
                        printf ("%sreadData Timeout\n");
                ErrorRecovery (ABORT, TIMEOUT,
                        "Receiver timeout - aborting connection");


default:
                if (FTPdebug)
                        printf ("%s[%s] (but were expecting data)\n",
                                                WEGOT, formatMark (buf[0]));
                ErrorRecovery (ABORT, PROTOCOL, buf + 2);
                }

        /*  Normal return from case BSP_DATA (otherwise no return) */
}

/*****************************************************************************
Read up to a mark byte (skipping data if trying to resync) and return
its value.  If timeout occurrs invoke error recovery.
```

```
*/

readMark ()
{
        static char buf[MAXPUPDATALEN];
        int bytes;                          /* # of bytes read from BSP */
        int puptype;                        /* puptype read from BSP */

        while ()
                {
                puptype   BSPread (buf, sizeof (buf), ONESEC * 30, &bytes);
                switch (puptype)
                        {
case BSP_MARK:                              /* this the the return we want */
                        if (FTPdebug)
                                printf ("%s[%s]\n", WEGOT,
                                        formatMark (buf[0]));
                        return (buf[0]);

case BSP_DATA:
                        if (FTPdebug && bytes != 0)
                                {
                                buf[bytes] = NULL;
                                printf ("%s%s", WEGOT,
                                        FormatReply (buf, TRUE));
                                }
                        continue;

case TIMEOUT:
                        if (FTPdebug)
                                printf ("%sreadMark Timeout\n");
                        ErrorRecovery (ABORT, TIMEOUT,
                                "Receiver timeout - aborting connection");

default:
                        if (FTPdebug)
                                printf ("%sExpected MARK, got 0%o\n", WEGOT,
                                                                    puptype);
                        ErrorRecovery (ABORT, PROTOCOL, buf + 2);
                        }
                }
        /*  Normal return is case BSP_MARK */
}

/*************************************************************************
Read a string into the supplied buffer looking for the desired mark byte
and return the mark byte.  Invoke error recovery if buffer overflow or
timeout.
*/

readUntil (buf, bufsiz, markwant)
char *buf;                              /* buffer for data */
int bufsiz;                             /* size of buffer */
unsigned char markwant;                 /* read upto this mark pup */
{
        int bytes;                      /* # of bytes read from BSP */
        unsigned char markgot;
```

```
        int puptype;                    /* puptype read from BSP */
        int sofar;

        for (sofar = 0; ; )
                {
                puptype = BSPread (buf + sofar, bufsiz   sofar, ONESEC * 30,
                                                        &bytes);
                sofar += bytes;
                buf[sofar] = NULL;
                switch (puptype)
                        {
case BSP_DATA:
                        if (sofar >= bufsiz)    /* no mark and buffer full */
                                ErrorRecovery (ABORT, PROTOCOL,
                                "FTP buffer overflow - aborting connection");
                        continue;

case BSP_MARK:
                        markgot   buf[--sofar];
                        buf[sofar]   NULL;
                        if (FTPdebug)
                                {
                                printf ("%s%s", WEGOT,
                                        FormatReply (buf, TRUE));
                                printf ("%s[%s]\n", WEGOT,
                                        formatMark (markgot));
                                if (markgot != markwant)
                                        printf ("%s %s %s\n",
                                        "But we were expecting a",
                                        formatMark (markwant),
                                        "mark byte!");
                                }
                        if (markgot == markwant)
                                return (SUCCESS);       /* return OK */
                        ErrorRecovery (ABORT, PROTOCOL,
                                "FTP protocol error - aborting connection");

case TIMEOUT:
                        if (FTPdebug)
                                printf ("%sreadUntil Timeout\n");
                        ErrorRecovery (ABORT, TIMEOUT,
                                "Receiver timeout   aborting connection");

default:
                        if (FTPdebug)
                                printf ("%s%s\n", WEGOT, buf);
                        ErrorRecovery (ABORT, PROTOCOL, buf+sofar-bytes+2);
                        }
                }
        /*  Normal return is getting mark desired in case BSP_MARK  */
}

/************************************************************************
Scan a property list for a property returns the value of the property or
a space if not found.  Parse according to FTP documentation.
*/
```

```
char *ScanPL (propertyList, desiredProperty)
char *propertyList;                     /* input property list */
char *desiredProperty;                  /* desired property */
{
        register char *PLptr;           /* ptr to property list */
        static char value[100];         /* value of property */
        register char *Vptr;

        for (PLptr   propertyList; *PLptr !  NULL; )
                {
                while (*PLptr == '(')
                        PLptr++;

                for (Vptr   value; *PLptr != NULL && *PLptr !- ' '; )
                        *Vptr++ = *PLptr++;     /* copy property name */

                if (*PLptr == NULL)
                        return (BLANK);

                *Vptr = NULL;
                if (match (value, desiredProperty))
                        {               /* found property looking for */
                        if (*PLptr++ != ' ')
                                return (BLANK);

                        for (Vptr = value; *PLptr != ')'; )
                                *Vptr++ = *PLptr++;

                        *Vptr = NULL;

                        if (strlen (value))
                                return (value);
                        else
                                return (BLANK);
                        }
                else
                        {
                        while (*PLptr++ != ')')
                                ;
                        }
                }

        return (BLANK);         /* Return a space if nothing found */
}

/*************************************************************************
Format and write the supplied string and supplied code as DATA on BSP
string.  If timeout invoke error recovery.
*/

writeData (code, string)
int code;                               /* code byte */
char *string;                           /* text string */
{
        char buffer[MAXPUPDATALEN];
        int error;                      /* status of BSP write */
        char *sendptr;
```

```
        int sendsize;                   /* computed bytes for BSP write */

        sendsize   strlen (string);
        if (code == Omit)               /* means no code byte */
                {
                if (FTPdebug)
                        printf ("%s%s\n", WESENT, string);
                sendptr = string;
                }
        else
                {
                buffer[0] = code;
                strcpy (buffer + 1, string);
                if (FTPdebug)
                        printf ("%s%s", WESENT, FormatReply (buffer, TRUE));
                sendptr = &buffer;
                sendsize++;
                }

        if ((error = BSPwrite (BSP_DATA, sendptr, sendsize)) != OK)
                {
                if (FTPdebug)
                        printf ("writeData: error code %d\n", error);
                ErrorRecovery (ABORT, TIMEOUT,
                                "Transmitter timeout   connection aborted");
                }

        return;
}

/*****************************************************************************
Send a mark byte of the supplied type on the BSP stream.  If mark byte is
end of command, write requesting acknowledgment of command.  Invoke error
recovery if write timeout.
*/

writeMark (markbyte)
unsigned char markbyte;                 /* desired mark to write */
{
        int error;

        if (FTPdebug)
                printf ("%s[%s]\n", WESENT, formatMark (markbyte));

        if ((error = BSPwrite (markbyte == EOC ? BSP_AMARK : BSP_MARK,
                               &markbyte, sizeof (markbyte))) != OK)
                {
                if (FTPdebug)
                        printf ("writeMark: error code %d\n", error);
                ErrorRecovery (ABORT, TIMEOUT,
                                "Transmitter timeout   connection aborted");
                }

        return;
}
```

```
/*
        File:  ftp/server.c
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  Server part of FTP.

        MODIFIED BY     REASON
*/

#include <sys/stypes.h>         /* types for file descriptor */
#include <sys/filedesc.h>       /* descriptor for file status command */
#include <sys/psinfo.h>         /* os pointer table for system name */
#include <sys/conf.h>           /* os configuration table for system name */
#include <sys/access.h>         /* filemode access bits */
#include <error.h>              /* UNOS error numbers */
#include <signal.h>             /* signal package common names */
#include <stdio.h>
#include <pwd.h>                /* passwd entry structure */
#include "ftp.h"

#define IDLETIME        (5*60)  /* Idle timeout (connection inactive) */
#define PATH            "PATH=:/bin:/usr/bin:/bin/local"

#define INVALID         1       /* invalid user login */

int DefaultType   Unknown;      /* default of file (text/binary) */
int EndOfLine = EOL_CR;         /* end of line convention */
int FTPdebug;                   /* true if debugging (an argument) */
char **myname;                  /* what 'ps' knows me by */
int Type = Unknown;             /* type of transfering file (text/binary) */

extern char **environ;          /* ptr to command interpreter environment */
extern char *ServerDate;        /* version date created by Makefile */
extern ServerFlag;              /* true when server (that's this program) */

extern FILE *popen ();
extern char *rindex();

char Home[100]          = BLANK;        /* string HOME=homedirectory */
char HomeDirectory[100] = BLANK;        /* for above */
char Password[100]      = BLANK;        /* user password */
char ServingWho[100]    = "FTPserving ?????";           /* for ps */
char SystemName[50]     = "?Amnesia?";  /* determined system name */
char UserName[100]        BLANK;        /* user name */

char readbuf[2048];                     /* read (BSP or file) buffer */

char *MyEnviron[] =
        {
        Home,
        PATH,
        NULL
        };

#include "serverutils.c"

/*-----------------------------------------------------------------------*/
```

```
main (argc, argv)
int argc;
char *argv[];
{
        FILEDESC fd;
        FILE memptr;                            /* points into /dev/kmem */
        register int i;
        unsigned char osconf[sizeof (conftable)];
        PSINFO ps;                              /* has addr of config table */
        extern SignalHandler ();

        if (strncmp (argv[1], "-help", strlen (argv[1])) == 0)
                {
                printf ("%s: [-b -f -p]\n", argv[0]);
                printf ("\t-b for BSP debug\n");
                printf ("\t-f for FTP debug\n");
                printf ("\t-p for PUP debug (lots of data)\n");
                printf ("\tStart FTP server process - must be admin\n");
                exit (GOODEXIT);
                }

        if (geteuid() != 0)
                comerr ("This program must be started by the super user\n");

        if (status (argv[0], &fd) < 0 || fd.f_owner != 0)
                comerr ("This program must be owned by the administrator\n");

        for (i = 1; i < argc; i++)
                {
                if (*argv[i] == '-')
                        {
                        switch (argv[i][1])
                                {
case 'b':                       BSPdebug = TRUE;        break;
case 'f':                       FTPdebug = TRUE;        break;
case 'p':                       PUPdebug   TRUE;        break;
default:                        comerr ("Bad flag '%s'\n", argv[i]);
                                }
                        }
                else
                        comerr ("Unknown argument: '%s'\n", argv[i]);
                }

        /*
        *  Initialize ourself.
        */
        ServerFlag = TRUE;              /* tell internals I'm server */
        myname = &argv[0];              /* make global ptr to process name */
        *myname   "FTPserver";          /* tell 'ps' who I am */
        psinfo (&ps, sizeof (ps));
        if ((memptr   fileopen ("/dev/kmem", "ru")) != NULL)
                {
                fileseek (memptr, (long) ps.ps_config);
                fileread (memptr, osconf, sizeof (osconf));
                strcpy (SystemName, ((conftable *) &osconf)->cnf_system_id);
                fclose (memptr);
```

```
                }
        else
                if (FTPdebug)
                        printf ("Can't open /dev/kmem for system name\n");

        /*
        *  Ignore some signals that might hurt us and set up a signal
        *  handler for SIGTERM which is sent when system is shutting
        *  down.  This will allow orderly cleanup.
        */
        signal (SIGHUP, SIG_IGN);
        signal (SIGPIPE, SIG_IGN);
        signal (SIGTERM, SignalHandler);

        /*
        *  Open a connection (for the well know FTP socket) and
        *  wait in the BSP package for an RFC at which time we
        *  return and fork off the server task.
        */
        while ()
                {
                switch (BSPopen (Listen, FTPSocket))
                        {
case RFC:
                        if (fork () != 0)
                                Connection ();
                        continue;

case NOCHAN:
                        if (FTPdebug)
                                printf ("No ethernet ports available.\n");
                        sleep (4);              /* pause and try again */
                        continue;
                        }
                }

        ErrorRecovery (NONE);                   /* only if fatal BSP open */
}

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

The "main" code for the forked server process.  Handles BSP read command
from the users process and executes the corresponding task.

*/

Connection ()
{
        char buf[BUFSIZE];              /* BSP read buffer */
        int bytes;                      /* number of bytes read */
        char errorMessage[256];

        *myname = ServingWho;           /* tell 'ps' who I am */

        /*
        *  Finish establishing the server connection with
        *  user port.
```

```
        */
        switch (BSPopen (Server, NULL))
                {
case OK:
                break;

case NOCHAN:
                sleep (4);
                BSPabort (NOCHAN, "No ethernet ports available for server");
                ErrorRecovery (NONE);
                }

        /*
        *  Loop forever (or until were told to close up) and perform
        *  user requested function based on puptype.
        */
        while ()
                {
                switch (BSPread (buf, sizeof (buf), ONESEC*IDLETIME, &bytes))
                        {
case TIMEOUT:
                        if (FTPdebug)
                                printf ("[FTP connection time out: %s]\n",
                                        printPort (&RemotePort));

                        ErrorRecovery (ABORT, TIMEOUT,
                                        "FTP Connection time out");
                        break;

case RTP_END:                                   /* normal exit */
                        BSPclose (RTP_ENDR);
                        if (FTPdebug)
                                printf ("[FTP connection closed: %s]\n",
                                        printPort (&RemotePort));

                        exit (GOODEXIT);
                        break;

case BSP_DATA:                                  /* ignore data */
                        buf[bytes] = NULL;
                        if (FTPdebug)
                                printf ("U: (Got DATA not MARK)   %s\n", buf);
                        break;

case BSP_MARK:                                  /* process command */
                        if (FTPdebug)
                                printf ("U: [%s]\n", formatMark (buf[0]));
                        switch (buf[0])
                                {
        case Comment:           DoComment ();           break;
        case Delete:            DoDelete ();            break;
        case Directory:         DoDirectory ();         break;
        case NewStore:          DoStore (1);            break;
        case Store:             DoStore (0);            break;
        case Retrieve:          DoRetrieve ();          break;
        case Version:           DoVersion ();           break;
        case EOC:               /* ignore */            break;
```

```
        case Yes:
        case No:
        case Rename:
        default:
                                if (FTPdebug)
                                        printf ("Unexpected mark!\n");
                                writeMark (No);
                                sprintf (errorMessage, "%s Not implemented",
                                                formatMark (buf[0]));
                                writeData (Not_implemented, errorMessage);
                                writeMark (EOC);
                                }       /* of switch (buf[0]) */
                        break;          /* case MARK */

case BSP_INTR:                                  /* ignore interrupt */
                        if (FTPdebug)
                                printf ("U: interrupt\n");
                        break;

default:                                        /* wierd pup type ?? */
                        if (FTPdebug)
                                printf ("U: strange pup type!\n");
                        }
                }
        /* never return - normal exit is case RTP_END */
}

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

Except user supplied commentary and ignore it.  The server has no device
to display it upon,

*/

DoComment ()
{
        char buffer[BUFSIZE];

        readData (buffer, sizeof (buffer));
        return;
}

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

Delete a file from file server - command / response sequence;
1. Terminate command if invalid login.
2. Send back property list and wait for a YES or NO mark byte.
3. If YES mark byte, attempt deletion and report YES or NO mark byte
   based on successful completion,

*/

DoDelete ()
{
        extern DeleteFile ();
        char *propertyList;
```

```
        propertyList = GetPropertyList ();
        if (CheckLogin (propertyList) == OK)
                expand (propertyList, &DeleteFile);

        writeMark (EOC);
        return;
}

/*-------------------------------------------------------------------------*/

DeleteFile (filename)                   /* delete a single file */
char *filename;
{
        int code;
        FILEDESC fd;                    /* file descriptor */

        SendPropertyList (filename);
        writeMark (EOC);

        switch (readMark ())            /* make sure he wants to delete it */
                {
case Yes:
                /*
                *  Allow delete if file is owned by current user or
                *  current user is administrator.
                */
                if (expectMark (EOC) != SUCCESS ||
                    (code = access (filename, AWRITE)) < 0)
                        {
                        writeMark (No);
                        writeData (Protection_violation, ErrToText (code));
                        }
                else if (fdelete (filename) < 0)
                        {
                        writeMark (No);
                        writeData (Protection_violation,
                                                "Unable to delete file");
                        }
                else
                        {
                        writeMark (Yes);
                        writeData (OK, "File deleted");
                        }

                break;

                                        /* Do NOT delete it! */
default:
                if (expectMark (EOC) != SUCCESS)
                        ErrorRecovery (ABORT, TIMEOUT, "Timeout");

                break;
                }

        return;
}

/*************************************************************************
```

```
List file(s) on file server - command / response sequence:
1. Terminate command if invalid login.
2. Send back property list.

*/

DoDirectory ()
{
        char *propertyList;
        extern SendPropertyList ();

        propertyList = GetPropertyList ();
        if (CheckLogin (propertyList) == OK)
                expand (propertyList, &SendPropertyList);

        writeMark (EOC);
        return;
}

/*-------------------------------------------------------------------------*/

SendPropertyList (filename)             /* send property list for a file */
char *filename;
{
        char directory[sizeof (HomeDirectory)];
        FILEDESC fildes;                /* file descriptor */
        char *nameBody;
        char ourPropertyList[BUFSIZE];
        struct passwd *pw;
        struct passwd *getpwuid ();

        /*
        *  Break the file name up into a directory and a name body.
        */
        if (nameBody   rindex (filename, '/'))
                {
                strcpy (directory, filename);
                directory [nameBody - filename] = NULL;
                nameBody++;
                }
        else
                {
                nameBody   filename;
                strcpy (directory, HomeDirectory);
                }

        if (DefaultType == Unknown)
                Type = GuessType (filename);
        else
                Type - DefaultType;

        if (status (filename, &fildes) < 0 ||
           (pw = getpwuid (fildes.f_owner)) == NULL)
                {
                sprintf (ourPropertyList, "((%s %s)(%s %s)(%s %s)%s)",
                                "Server-Filename", filename,
```

```
                                "Name-body", nameBody,
                                "Directory", directory,
                                Type==Text ? "(Type Text)" : Type==Binary ?
                                "(Type Binary)(Byte-size 8)" : "");
                }
        else
                {
                sprintf (ourPropertyList,
                        "((%s %s)(%s %s)(%s %s)(%s %s)(%s %d)(%s %s)%s)",
                                "Server-Filename", filename,
                                "Name-body", nameBody,
                                "Directory", directory,
                                "Author", pw->pw_name,
                                "Size", fildes.f_size,
                                "Write-Date", maketime (fildes.f_tmodified),
                                Type==Text ? "(Type Text)" : Type==Binary ?
                                "(Type Binary)(Byte-size 8)" : "");
                }

        writeMark (Here_is_property_list);
        writeData (Omit, ourPropertyList);

        return;
}

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

Retrieve a file from file server - command / response sequence:
1. Terminate command if invalid login.
2. Send back property list and wait for a YES or NO mark byte.
3. If YES mark byte, server now sends the file followed by a YES
   mark byte to signify end of data.

*/

DoRetrieve ()
{
        char *propertyList;
        extern RetrieveFile ();

        propertyList = GetPropertyList ();
        if (CheckLogin (propertyList) == OK)
                {
                switch (EndOfLine)      /* EOL conversion necessary ? */

                        {
case EOL_CR:                            /* UNOS LF to CR */
                        EndOfLine = EOL_NL;
                        break;

case EOL_CRLF:                          /* UNOS LF to CRLF */
                        EndOfLine = EOL_Add;
                        break;
                        }

                expand (propertyList, &RetrieveFile);
                }
```

```
        writeMark (EOC);
        return;
}

/*-------------------------------------------------------------------------*/

RetrieveFile (filename)                 /* retrieve one file */
char *filename;
{
        int code;
        FILEDESC fd;                    /* file descriptor */
        FILE fp;

        SendPropertyList (filename);
        writeMark (EOC);

        switch (readMark ())            /* wait for user */
                {
case No:        if (expectMark (EOC) != SUCCESS)
                        ErrorRecovery (ABORT, TIMEOUT, "Timeout");
                return;                 /* user is skipping it */

case Yes:       if (expectMark (EOC) != SUCCESS)
                        ErrorRecovery (ABORT, TIMEOUT, "Timeout");
                }

        /*
        *  Allow read only if file exists by current user or the
        *  current user is administrator or filemodes give access
        *  priviledge and the file can be opened properly.
        */
        if ((code = access (filename, AREAD)) < 0)
                {
                writeMark (No);
                writeData (Protection_violation, ErrToText (code));
                return;
                }
        if ((fp = fileopen (filename, FORREAD)) == NULL)
                {
                writeMark (No);
                writeData (Protection_violation,
                                        "Can't open remote file for read");
                return;
                }

        DiskToNet (fp, readbuf, sizeof (readbuf),
                        Type == Binary ? EOL_Trans : EndOfLine, NULL);

        return;
}

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

Store file on file server - command / response sequence:
1. Terminate command if invalid login, no disk space, etc.
2. If NewStore, send back property list and wait for a YES or NO
```

```
   mark byte.
3. The user now sends the file followed by a YES mark byte to signify
   end of data.
4. Server reports YES or NO mark byte based on successful completion.

*/

DoStore (new)                    /* "new" is true for a New store */
int new;
{
        int code;
        FILEDESC fd;                        /* file descriptor */
        char filename[256];                 /* the file name to store */
        FILE fp;                            /* file writing into */
        char *propertyList;
        int result;                         /* the result of transfer */
        char *temp;

        propertyList   GetPropertyList ();

        if (CheckLogin (propertyList) == INVALID)
                {
                writeMark (EOC);
                return;
                }

        if (match (temp = ScanPL (propertyList, "Server-Filename"), BLANK) &&
            match (temp   ScanPL (propertyList, "Name-Body"), BLANK))
                {
                writeMark (No);
                writeData (Do_not_send,
                                "Can't find file name in property list");
                writeMark (EOC);
                return;
                }
        else
                {
                strcpy (filename, temp);
                AddDirectory (filename, propertyList);
                }

        /*
        *  Allow store (newstore or store) only if file previously existed
        *  by current user or the filemodes give access priviledge or the
        *  current user is administrator.  If file did not previously exist,
        *  it must be able to be opened.
        */
        if ((code = access (filename, AWRITE)) < 0 && code != ENOFILE)
                {
                writeMark (No);
                writeData (Protection_violation, ErrToText (code));
                writeMark (EOC);
                return;
                }
        if ((fp = fileopen (filename, FORWRITE)) == NULL)
                {
                writeMark (No);
```

```
                writeData (Protection_violation,
                                        "Can't open remote file to write");
                writeMark (EOC);
                return;
                }

        if (new)
                {
                /*
                *  If this is a newstore operation, we send back the
                *  property list and let the user decide again if he
                *  really wants to do the store.
                */
                SendPropertyList (filename);
                writeMark (EOC);
                if (expectMark (Here_is_the_file) != SUCCESS)
                        {       /*  user decided against storing the file */
                        expectMark (EOC);
                        writeMark (No);
                        writeData (Do_not_send, "Not stored");
                        writeMark (EOC);
                        return;
                        }
                }
        else
                {
                /*
                *  If store (not new store) indicate ready for file.
                */
                writeMark (Yes);
                writeData (OK, "File open, ready for data");
                writeMark (EOC);
                expectMark (Here_is_the_file);
                }

        result - NetToDisk (fp, readbuf, sizeof (readbuf),
                 DefaultType == Binary ? EOL_Trans : EndOfLine, NULL);

        if (expectMark (EOC) != SUCCESS)        /* swallow his message */
                ErrorRecovery (ABORT, TIMEOUT, "Timeout");

        switch (result)
                {
case MARKYES:   writeMark (Yes);
                writeData (OK, "Store complete");
                break;

case MARKNO:    writeMark (No);
                writeData (OK, "Store unsuccessful");
                break;

case ABORTED:   ErrorRecovery (NONE);
                }

        writeMark (EOC);
        return;
}
```

```
/*****************************************************************************

Send back server version in reponse to users request.

*/

DoVersion ()
{
        char buffer[BUFSIZE];

        readUntil (buffer, sizeof (buffer), EOC);

        writeMark (Version);            /* send back our response */
        sprintf (buffer, "%s - 68000/UNOS FTP Server 1.00 of %s",
                                                SystemName, ServerDate);
        writeData (FTPVersion, buffer);
        writeMark (EOC);

        return;
}

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

Signal handler to allow orderly clean of server.

*/

SignalHandler (sig)
int sig;
{

        if (sig == SIGTERM)
                {
                printf ("%s: cleaning up - received SIGTERM\n", *myname);
                BSPclose (OK);
                exit (0);
                }
}

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

Called on communication errors (or any others) to cleanup communications,
and exit.  There is a routine by the same name in user task.

*/

ErrorRecovery (action, abortcode, abortstring)
int action;
int abortcode;
char *abortstring;
{

        switch (action)
                {
case ABORT:     BSPabort (abortcode, abortstring);
                break;
```

```
case CLOSE:     BSPclose (RTP_END);
                break;
                }

        exit (BADEXIT);
}
```

```
/*
        File:  ftp/serverutils.c
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  Utilities for server part of FTP.

        MODIFIED BY     REASON

        access              Check if user can access file.
        AddDirectory        Prepend directory to make absolute pathname.
        CheckLogin          Assure valid user on this system.
        ErrToText           Convert error numbers to messages.
        expand              Expand wildcards for filename.
        GetPropertyList     Read property list from net.
        Lower               Lower casify string.
        maketime            Convert UNOS clock ticks to date and time.
*/

#include <sys/stypes.h>
#include <sys/filedesc.h>
#include <error.h>

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

Check if user has the priviledge to access the file.  Return OK or
system error code.

*/

access (name, mode)
char *name;                                 /* absolute pathname of file */
int mode;                                   /* access mode */
{
        int code;
        FILEDESC stat;
        unsigned short u;

        typedef struct
                {
                unsigned char owner;
                unsigned char world;
                } filemodes;

        if ((code   status (name, &stat)) < 0)
                return (code);

        u = stat.f_modes;

        /*
        *  If owner and owner modes allow access.
        */
        if ((u.owner & mode) == mode && getuid () == stat.f_owner)
                return (0);

        /*
        *  If world modes allow access or administrator.
        */
```

```
        if ((u.world & mode) == mode || getuid () == 0)
                return (0);

        return (EACCESS);
}

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

Prepend directory onto users filename in property list provided the
filename was not specified with absolute path name and a directory
exists to prepend.  Return the absolute path.

*/

AddDirectory (PLfilename, propertyList)
char *PLfilename;
char *propertyList;                     /* the property list */
{
        char *directory;
        char tempbuf[256];

        directory   ScanPL (propertyList, "Directory");

        if (match (directory, BLANK) == FALSE && (PLfilename[0] != '/'))
                {
                strcatl (tempbuf, directory, "/", PLfilename, NULL);
                strcpy (PLfilename, tempbuf);
                }

        return (PLfilename);
}

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

Verifies the remote users login and returns INVALID or OK.

*/

char *CheckLogin (propertyList)
char *propertyList;
{
        char new_user_name[sizeof (UserName)];
        struct passwd *getpwnam ();
        struct passwd *pw;
        register int saveuid;

        strcpy (new_user_name, ScanPL (propertyList, "User-name"));
        if (match (new_user_name, BLANK) || strlen (new_user_name) < 1)
                {
                writeMark (No);
                writeData (Illegal_user_name, "Illegal user-name");
                return (INVALID);
                }

        if (match (new_user_name, UserName) == FALSE)
                {
                if ((pw   getpwnam (new_user_name)) == NULL &&
```

```
            (pw   getpwnam (Lower (new_user_name))) == NULL)
                {
                writeMark (No);
                writeData (Illegal_user_name, "Not a valid user");
                return (INVALID);
                }

            /*
            *  Check the password if there is one.
            */
            if (strlen (pw->pw_passwd))
                {
                strcpy (Password, ScanPL (propertyList,
                                          "User-password"));
                if (strcmp (pw->pw_passwd, crypt (Password, "")) &&
                    strcmp (pw->pw_passwd, crypt (Lower (Password),
                                                  "")))
                    {
                    writeMark (No);
                    writeData (Illegal_password,
                                "Incorrect user-password");
                    return (INVALID);
                    }
                }

            saveuid = chdomain ();
            if (setuid (pw->pw_uid) < 0 || setwd (pw->pw_dir) < 0)
                {
                writeMark (No);
                writeData (Illegal_password,
                            "Unable to setup user's environment");
                setuid (saveuid);
                return (INVALID);
                }
            setuid (saveuid);

            /*
            *  Set up the current, home directories, and search paths.
            */
            strcpy (UserName, new_user_name);
            strcpy (HomeDirectory, pw->pw_dir);
            sprintf (Home, "HOME=%s", HomeDirectory);
            environ = MyEnviron;
            sprintf (ServingWho, "FTPserving %s", UserName);
            }

        return (OK);
}

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

Convert access error numbers into messages.  Return pointer to string.

*/

ErrToText (number)
int number;
```

```
{
        static char buffer[128];

        switch (number)
                {
case ENDFILE:   return ("Remote file(s) does not exist.");

case EBADNAME:  return ("A bad remote filename has been given.");

case EMISSDIR:  return ("Pathname for remote file has bad directory.");

case EACCESS:   return ("Access permission denied to remote file.");

default:        sprintf (buffer, "UNOS error code: %d", number);
                return (buffer);
                }
}

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

Use the system routine 'list' to expand wildcards and to verify the
existence of each file.  Call the passed procedure (either Delete,
List, or Retrieve) for each expanded filename.

*/

expand (propertyList, proc)
char *propertyList;                     /* a property list */
int (* proc) ();                        /* procedure to be called */
{
        char filename[256];             /* expanded filename */
        FILE *ls;                       /* the listing stream */
        char lsCommand[256];            /* command to run */
        char PLfilename[256];           /* extracted PL filename */
        register char *tp;

        /*
        *  Extract filename, prepend directory (if necessary), and
        *  execute directory list command to give us fully expanded
        *  filenames, one per line.
        */
        strcpy (PLfilename, ScanPL (propertyList, "Server-Filename"));
        AddDirectory (PLfilename, propertyList);
        sprintf (lsCommand, "/bin/list -all -dir %s", PLfilename);
        BSPsettimer (OFF);
        ls = popen (lsCommand, "r");
        BSPsettimer (ON);
        if (ls == NULL)
                {
                writeMark (No);
                writeData (OK, "Can't expand filename with 'popen'");
                return;
                }

        /*
        *  Read in expanded filenames.  If there are blanks on the
        *  line then "File(s) not found" is there, indicating error.
```

```
        */
        while (fgetline (ls. filename. sizeof (filename)) > 0)
                {
                tp   rindex (filename, '\n'):   /* remove the newline */
                if (tp) *tp   NULL;

                if (rindex (filename, ' '))
                        {
                        writeMark (No);
                        writeData (File_not_found,
                                ErrToText (access (PLfilename, AREAD))):
                        pclose (ls);
                        return;
                        }
                (* proc) (filename);            /* execute command */
                }

        pclose (ls);
        return;
}

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

Read the freshly received property list and scan setting a few global
variables.  Return pointer to property list.

*/

char *GetPropertyList ()
{
        static char propertyList[BUFSIZE];

        readUntil (propertyList, sizeof (propertyList), EOC);
        DefaultType = DetermineType (propertyList);
        EndOfLine = DetermineEOL (propertyList);

        return (propertyList);
}

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

Lower casify a string.  Return a pointer to string.

*/

Lower (s)
register char *s;
{
        register char *save;

        for (save = s; *s; )
                *s++ = tolower(*s);

        return (save);
}

/*************************************************************************
```

```
Convert UNOS clock ticks to text data and time and return a pointer to
text string.  Format of string "day-month-year hour:minute:second".

*/

maketime (ticks)
{
        static char *month[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
        int tvec[7];
        static char time[50];

        cvtime (ticks, tvec);
        sprintf (time, "%.2d-%.3s-%.2d %02d:%02d:%02d", tvec[2],
                        month[tvec[1]], tvec[3] - 1900, tvec[4],
                        tvec[5], tvec[6]);

        return (time);
}
```

```
/*
        File:  ftp/user.c
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  User part of FTP.

        MODIFIED BY     REASON
*/

#include <sys/stypes.h>         /* types for file descriptor */
#include <sys/filedesc.h>       /* descriptor for file status command */
#include <signal.h>             /* signal package common names */
#include <stdio.h>
#include "ftp.h"
#include "command.h"            /* some type definitions */

int FTPdebug = FALSE;           /* true if we are in debug mode */
int HashMode = TRUE;            /* hash marks on buffer loads */
int HashCount = 0;
int ScriptInput = FALSE;        /* don't ask for confirmation if not tty */

int DefaultType   Guess;        /* file type default */
int EOLconvention = EOL_CR;     /* net default */
int ServerDoesNewStore = TRUE;  /* Assume server can do NewStore */

char DirectoryName[256];        /* current remote fileserver directory */
char FileName[256];             /* server filename of transaction */
char HostName[100];             /* name of the remote fileserver */
char InputBuffer[2048];         /* input data buffer */
char LoginDirectory[256];       /* user's login directory */
char Password[50];              /* user's password */
char UserName[50];              /* user's name */


CMDTABLE cmd_table[] =          /* table defined in command.h */
        {
        { 1, "cd (local directory to) ",        DoChange,       NULL},
        { 2, "delete remote file ",             DoDelete,       NULL},
        { 2, "directory (remote default) ",     DoDirectory,    NULL},
        { 1, "EOL convention ",                 DoEOL,          NULL},
        { 2, "list remote files matching ",     DoList,         NULL},
        { 2, "login as remote user ",           DoLogin,        NULL},
        { 1, "quit ",                           DoQuit,         NULL},
        { 1, "retrieve remote file ",           DoRetrieve,     NULL},
        { 2, "store local file ",               DoStore,        NULL},
        { 2, "show (progress of transfer) ",    DoToggle,       &HashMode},
        { 1, "type (default for transfer) ",    DoType,         NULL},
        { 1, "verbose (mode) ",                 OoToggle,       &FTPdebug},
        { 1, "! (shell command) ",              OoShell,        NULL},
        { 0, "", NULL, NULL}
        };

CMOTABLE EOLTable[] -
        {
        { 2, "CR",              SetEOL,         EOL_CR},
        { 3, "CRLF",            SetEOL,         EOL_CRLF},
        { 1, "Transparent",     SetEOL,         EOL_Trans},
```

```
        { 0, "", NULL, NULL},
        };

CMDTABLE typeTable[]
        {
        { 1, "binary",          SetType,        Binary},
        { 1, "text",            SetType,        Text},
        { 1, "unspecified",     SetType,        Guess},
        { 0, "", NULL, NULL},
        };

#include "userutils.c"

/*------------------------------------------------------------------------*/

main (argc, argv)
int argc;
char *argv[];
{
        register int i;
        int input;
        extern SignalHandler ();

        if (argc < 2 || strncmp (argv[1], "-help", strlen (argv[1])) == 0)
                {
                printf ("%s: [-b -f -p] server_hostname\n", argv[0]);
                printf ("\t-b for BSP debug\n");
                printf ("\t-f for FTP debug\n");
                printf ("\t-p for PUP debug (lots of data)\n");
                printf ("\tserver_hostname to whom you wish connection\n");
                exit (GOODEXIT);
                }

        for (i   1; i < argc - 1; i++)
                {
                if (*argv[i] == '-')
                        {
                        switch (argv[i][1])
                                {
case 'b':                       BSPdebug   TRUE;        break;
case 'f':                       FTPdebug   TRUE;        break;
case 'p':                       PUPdebug = TRUE;        break;
default:                        comerr ("Bad flag '%s'\n", argv[i]);
                                }
                        }
                else
                        comerr ("Unknown argument: '%s'\n", argv[i]);
                }

        strcpy (HostName, argv[argc - 1]);

        /*
        *  See if were running off a terminal or being piped into.
        *  If were being piped into, we can't ask any questions.
        */
        ScriptInput = IsaTTY (stdin) == FALSE;
        if (ScriptInput == FALSE)
```

```
        SetTTY ();              /* set up terminal modes */

    /*
    *  Set up signals with appropriate signal handlers for
    *  proper cleanup and termination only if not in debug mode.
    */
    for (i   SIGHUP; i <= SIGTERM; i++)
            {
            if ((i != SIGQUIT && i != SIGTRAP && FTPdebug == FALSE) ||
               (i == SIGINT))
                    signal (i, &SignalHandler);
            }
    signal (SIGPIPE, SIG_IGN);

    /*
    *  Find network address of server hostname desiring to
    *  connect with.
    */
    switch (BSPlookup (HostName))
            {
case OK:
            break;

case NOTFOUND:
            printf ("[%s]\n", PUPErrorMsg);
            ErrorRecovery (NONE);

case TIMEOUT:
            printf ("[name server didn't respond]\n");
            ErrorRecovery (NONE);

case NOCHAN:
            printf ("[Sorry, no ethernet ports available]\n");
            ErrorRecovery (NONE);

default:
            if (FTPdebug)
                    printf ("Unexpected value returned from BSPlookup\n");
            }

    /*
    *  Try to open connection with server.
    */
    switch (BSPopen (Initiate, FTPSocket))
            {
case OK:
            if (FTPdebug)
                    printf ("[Connected to: %s]\n", HostName);
            break;

case TIMEOUT:
            printf ("[%s file server failed to respond]\n", HostName);
            ErrorRecovery (NONE);

case RTP_ABORT:
            printf ("[Abort: %s]\n", PUPErrorMsg);
            ErrorRecovery (NONE);
```

```
case NOCHAN:
                printf ("[Sorry, no ethernet ports available]\n");
                ErrorRecovery (NONE);

case NOROUTE:
                printf ("[Can't get there from here]\n");
                ErrorRecovery (NONE);

default:
                printf ("[Internal system error in BSPopen ???]\n");
                ErrorRecovery (NONE);
                }

        strcpy (DirectoryName, gwd (LoginDirectory));
        strcpy (Password, "");
        strcpy (UserName, getuname (getuid ()));

        /*
        *  Exchange protocol versions and verify that they match.
        *  If not, we won't understand the servers command / response
        *  sequence and will have to quit.
        */
        writeMark (Version);
        writeData (FTPVersion, "68000/UNOS User FTP 1.00");
        writeMark (EOC);

        if (expectMark (Version) != SUCCESS)
                {
                printf ("ERROR: We did not get any version message!\n");
                ErrorRecovery (CLOSE);
                }

        readUntil (InputBuffer, sizeof (InputBuffer), EOC);
        if (InputBuffer[0] != FTPVersion)
                {
                printf ("Protocol version mismatch %s %d, %s %d\n",
                                "user is:", FTPVersion,
                                "server is:", InputBuffer[0]);
                printf ("I have to quit, dont understand that protocol\n");
                printServerMsg (InputBuffer, TRUE);
                ErrorRecovery (CLOSE);
                }

        printServerMsg (InputBuffer, TRUE);     /* Print server herald */

        /*
        *  Loop forever waiting for input and executing commands.
        *  It is possible for the server to timeout if we wait too long.
        */
        for (input = ~EOF; input != EOF; )
                {
                printf ("FTP-> ");
                input = DoCommand (cmd_table);
                }

        ErrorRecovery (CLOSE);
```

```
}

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

Change the current local host directory.  This does not effect remote
server directory.

*/

DoChange ()
{
        char directory[128];

        if (ReadTTY (directory, sizeof (directory), Echo) > 0)
                {
                if (setwd (directory) < 0)
                        printf (" ... failed");
                }
        putchar ('\n');

        return;
}

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

Delete a file from file server - command / response sequence:
1. Send delete mark byte and property list to server.
2. Wait for server to respond with property list.
3. Query user for confirmation of delete and send back appropriate
   YES or NO mark byte based on query.

*/

DoDelete ()
{
        int len;                /* # of chars typed on TTY */
        int markbyte;           /* returned remote servers mark */

        len   ReadTTY (FileName, sizeof (FileName), Echo);
        putchar ('\n');
        if (len == 0)
                return;

        writeMark (Delete);
        writeData (Omit, MakePL (DirectoryName, FileName, UserName, Password,
                                 Unknown, Unknown));
        writeMark (EOC);

        while ()                        /* loop to handle wildcards right */
                {
                switch (markbyte - readMark ())
                        {
default:                                        /* something went wrong */
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, TRUE);
                        expectMark (EOC);
                        return;
```

```
case Comment:                                   /* got a chatty server */
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, FALSE);
                        break;

case EOC:                                       /* normal return */
                        return;

case Here_is_property_list:                     /* this is what we want */
                        readUntil (InputBuffer, sizeof (InputBuffer), EOC);

                        printf ("Delete %s ", ScanPL (InputBuffer,
                                                "Server-filename"));
                        printf ("of %s ", ScanPL (InputBuffer,
                                                "Write-date"));

                        if (YesNo ("? "))
                                {
                                writeMark (Yes);
                                writeData (OK, "Delete that file");
                                writeMark(EOC);

                                switch (markbyte = readMark())
                                        {
        case Yes:                               break;

        case No:                                readData (InputBuffer,
                                                        sizeof (InputBuffer));
                                                printServerMsg (InputBuffer, TRUE);
                                                break;
                                                }
                                }
                        else
                                {
                                writeMark (No);
                                writeData (OK, "Dont delete that file");
                                writeMark (EOC);
                                }
                        }
                }
        /* normal return is from case EOC */
}

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

Change the default directory on the remote file server. ('.' changes it
to the current local host directory.)  This information is saved for
the next transmission of property list.

*/

DoDirectory ()
{
        char directory[sizeof (DirectoryName)];

        if (ReadTTY (directory, sizeof (directory), Echo) > 0)
```

```
                {
                strcpy (DirectoryName, match (directory, ".") ?
                                        gwd (directory) : directory);
                }

        putchar ('\n');
        return;
}

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

Allows the user to specify a default end of line convention for text files.

*/

DoEOL ()
{

        DoCommand (EOLTable);
        putchar ('\n');
}

/*-----------------------------------------------------------------------*/

SetEOL (arg)                    /* called by DoCommand via DoEOL */
int arg;
{
        EOLconvention = arg;
        return;
}

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

List file(s) on file server - command / response sequence:
1. After getting necessary inforamtion from terminal, send directory mark
   byte and property list to server.
2. Wait for here-is-property-list mark byte and display on terminal.

*/

DoList ()
{
        char answer;
        int done;               /* true to stop list to terminal */
        register int i;
        int markbyte;           /* returned remote servers mark */
        int len;                /* # of chars typed on TTY */

        len = ReadTTY (FileName, sizeof (FileName), Echo);
        putchar ('\n');
        if (len == 0)
                return;

        writeMark (Directory);
        writeData (Omit, MakePL (DirectoryName, FileName, UserName, Password,
                                        Unknown, Unknown));
        writeMark (EOC);
```

```
        switch (markbyte   readMark ())
                {
case No:                                        /* server cant list - error */
                readUntil (InputBuffer, sizeof (InputBuffer), EOC);
                printServerMsg (InputBuffer, TRUE);
                break;

case Comment:                                   /* got a chatty server */
                readData (InputBuffer, sizeof (InputBuffer));
                printServerMsg (InputBuffer, FALSE);
                break;

case Here_is_property_list:                     /* this is what we want */
                DirHeader ();

                /*
                *  Loop getting servers data (property list) and
                *  display on terminal file information.
                */
                for (done   FALSE; done == FALSE && markbyte != EOC; )
                        {
                        for (i   1; (markbyte == Here_is_property_list) &&
                                                (i < 22); i++)
                                {
                                while (readData (InputBuffer,
                                        sizeof (InputBuffer)) == BSP_DATA &&
                                        strlen (InputBuffer) == 0)
                                        ;

                                DirEntry ();
                                markbyte = readMark ();
                                }

                        /*
                        *  More to display but terminal screen full -
                        *  ask for continuation (if stdin is tty).
                        */
                        if (markbyte != EOC)
                                {
                                if (ScriptInput == FALSE)
                                        {
                                        printf ("- MORE? ");
                                        answer = tolower (getchar ());
                                        done = (answer == 'q' || answer == 'n'
                                                || answer == ESC);
                                        for (i   strlen("- MORE? "); i-- > 0;)
                                                printf ("\010 \010");
                                        }
                                }
                        else
                                break;
                        }

                /*
                *  Loop here to suck servers data and discard if user
                *  has answered no to display more on terminal.
```

```
                */
                while (markbyte != EOC)
                        {
                        readData (InputBuffer, sizeof (InputBuffer));
                        markbyte = readMark ();
                        }
                break;

default:
                printf ("Protocol error in DoList [%s]\n",
                                formatMark (markbyte));
                }

        return;
}

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

Print out heading for directory list command.

*/

DirHeader ()
{
        printf ("\n Name                        Author          %s",
                        "    Write-date       Size     Type\n");
}

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

Print out a directory listing line and the directory name if changed for
list command.

*/

DirEntry ()
{

        printf (" %-23s", ScanPL (InputBuffer, "Name-body"));
        printf (" %-13s ", ScanPL (InputBuffer, "Author"));
        printf ("%20s   ", ScanPL (InputBuffer, "Write-date"));
        printf (" %-6s   ", ScanPL (InputBuffer, "Size"));
        printf ("%s\n", ScanPL (InputBuffer, "Type"));

        return;
}

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

Login as remote user.  This information will be used with the next
property list sent to server.

*/

DoLogin ()
{
        char newUserName[sizeof (UserName)];
```

```
        if (ReadTTY (newUserName, sizeof (UserName), Echo) > 0)
                {
                strcpy (UserName, newUserName);
                if (TermTTY () != CR)
                        {
                        printf (" password ");
                        ReadTTY (Password, sizeof (Password), Silent);
                        }
                else
                        strcpy (Password, "");
                }

        putchar ('\n');
        return;
}

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

The user desires to quit FTP.  This is the normal way of exiting.

*/

DoQuit ()
{

        if (ScriptInput == FALSE) ResetTTY ();
        BSPclose (RTP_END);
        printf ("... connection closed\n");
        setwd (LoginDirectory);

        exit (GOODEXIT);
}

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

Retreive a file from file server - command / response sequence:
1. After getting necessary information from terminal, send retreive mark
   byte and property list to server.
2. Wait for server to respond with property list.
3. Send back YES mark byte when were ready.
4. If YES mark byte, server now sends the file followed by a YES
   mark byte to signify end of data or NO mark byte if error.

*/

DoRetrieve ()
{
        FILE fp;                        /* file ptr writing with */
        int len;                        /* # of chars typed on TTY */
        register int markbyte;          /* read mark byte */
        char *nameBody;
        extern PrintHash ();
        int result;                     /* result to transfer */
        register int ThisConvention;    /* this file end of line convention */
        register int ThisType;          /* this file type */
```

```
        len   ReadTTY (FileName, sizeof (FileName), Echo);
        putchar ('\n');
        if (len == 0)
                return;

        /*
        *  Tell server you desire retrieve and send property list.
        */
        writeMark (Retrieve);
        writeData (Dmit, MakePL (DirectoryName, FileName, UserName, Password,
                                        DefaultType, Unknown));
        writeMark (EOC);

        /*
        *  Wait for servers response to your request (and property list).
        */
        while ()                        /* loop to handle wild cards */
                {
                switch (markbyte = readMark ())
                        {
case No:                                        /* something was wrong */
                        readUntil (InputBuffer, sizeof (InputBuffer), EOC);
                        printServerMsg (InputBuffer, TRUE);
                        return;

default:                                        /* some other error */
                        printf ("Protocol error in Retreive [%s]\n",
                                        formatMark (markbyte));
                        return;

case Comment:                                   /* got a chatty server */
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, FALSE);
                        break;

case EOC:                                       /* normal return */
                        return;

case Here_is_property_list:                     /* this is what we want */
                        /*
                        *  Determine file type and end of line convention
                        *  from servers returned property list.
                        */
                        readUntil (InputBuffer, sizeof (InputBuffer), EOC);

                        ThisType = DetermineType (InputBuffer);
                        if (ThisType == Unknown)
                                ThisType = DefaultType;
                        if (ThisType != Binary)
                                printf ("Text file");
                        else
                                printf ("Binary file");

                        ThisConvention   DetermineEOL (InputBuffer);

                        strcpy (FileName,
                                ScanPL (InputBuffer, "Server-filename"));
```

```
                printf (" %s", FileName);

                nameBody   ScanPL (InputBuffer, "Name-body");
                if (match (nameBody. BLANK) == FALSE)
                        strcpy (FileName, nameBody);

                /*
                *  What name does user want to call file on
                *  local host.
                */
                printf (" to local file " );
                if ((fp = GetLocalFileName (FileName)) == NULL)
                        {
                        writeMark (No);
                        writeData (Do_not_send, "No thanks");
                        writeMark (EOC);
                        continue;
                        }

                /*
                *  Were already, tell server to send the file.
                */
                writeMark (Yes);
                writeData (OK, "File open, ready for data");
                writeMark (EOC);
                switch (readMark ())
                        {
default:                                /* something wrong */
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, TRUE);
                        fclose (fp);
                        break;

case Comment:                           /* got a chatty server */
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, FALSE);
                        readMark ();
                        continue;

case Here_is_the_file:                  /* this is what we want */
                        HashCount   0;
                        result = NetToDisk (fp, InputBuffer,
                                sizeof (InputBuffer),
                                ThisType == Binary ? EOL_Trans :
                                ThisConvention, PrintHash);
                        switch (result)
                                {
        case MARKYES:                   readData (InputBuffer,
                                                sizeof (InputBuffer));
                                        break;
        case MARKNO:                    readData (InputBuffer,
                                                sizeof (InputBuffer));
                                        printServerMsg (InputBuffer, TRUE);
                                        break;
        case ABORTED:                   printf ("%s\n", InputBuffer + 2);
                                        ErrorRecovery (NONE);
                                        }
```

```
                                }                       /* switch (readMark ()) */
                        }                               /* case Here is PL */
                }                                       /* while loop */

        /* normal return is from case EOC */
}

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

Allow the user to temporarily escape FTP to execute a shell command.

*/

DoShell ()
{
        char command[128];

        if (ScriptInput == FALSE)
                ResetTTY ();    /* put TTY back into resaonable mode */
        if (fgetline (stdin, command, sizeof (command)) > 8)
                {
                BSPsettimer (OFF);
                system (command);
                BSPsettimer (ON);
                }
        if (ScriptInput == FALSE)
                SetTTY ();

        return;
}

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

Store file on file server   command / response sequence:
1. After getting necessary information from terminal, send newstore or
   store mark byte to server.
2. Wait for server to respond with PropertyList mark byte for newstore or
   YES mark byte for store.
3. The user now sends the file followed by a YES mark byte to signify
   end of data.
4. Server reports YES or NO mark byte based on successful completion.

*/

DoStore ()
{

        FILE fp;                        /* file reading from */
        register int markbyte;          /* returned remote servers mark */
        register int localEOL;          /* convertion for outgoing file */
        char Name[sizeof (FileName)];
        register int ThisType;

        if (ReadTTY (FileName, sizeof (FileName), Echo) == 0)
                {
                putchar ('\n');
                return;
```

```
                }

        if ((fp = fileopen (FileName, FORREAD)) == NULL)
                {
                printf ("\nCannot open: %s\n", FileName);
                return;
                }

        /*
        *  Ask user for remote filename.
        */
        printf (" as remote file ");
        if (ReadTTY (Name, sizeof (Name), Echo) == 0)
                {
                if (TermTTY () == ESC)
                        {
                        printf (" -- skipped\n");
                        return;
                        }
                strcpy (Name, FileName);
                printf ("%s\n", Name);
                }
        else
                putchar ('\n');

        /*
        *  Check if the file exists on the remote file server.
        */
        switch (RemoteFileLookup (Name))
                {
case OLDFILE:   printf ("Remote file %s already exists - overwrite", Name);
                if (YesNo ("? ") == FALSE)
                        return;
case NEWFILE:   break;
case FAILURE:   return;
                }

        /*
        *  Determine file type and end of line conversion that
        *  may be required.
        */
        ThisType = AskType (FileName);
        switch (EOLconvention)
                {
case EOL_CR:    localEOL = EOL_NL;              /* UNOS LF to CR */
                break;

case EOL_CRLF:  localEOL = EOL_Add;             /* UNOS LF to CRLF */
                break;

default:        localEOL   EOL_Trans;           /* no change */
                break;
                }

        /*
        *  We initially assume the server can do a NewStore.  Give
        *  that a try and if it fails then mark flag - ServerDoesNewStore
```

```
        *  to indicate server cannot do NewStore and then issue Store.
        *  Subsequent store attempts will issue NewStore based on
        *  the state of the flag - ServerDoesNewStore.
        */
        if (ServerDoesNewStore)
                {
                writeMark (NewStore);
                writeData (Omit, MakePL (DirectoryName, Name, UserName,
                                        Password, ThisType, EOLconvention));
                writeMark (EOC);

                if ((markbyte   readMark ()) == No)
                        {
                        readUntil (InputBuffer, sizeof (InputBuffer), EOC);
                        if (InputBuffer[0] == Not_implemented)
                                {
                                ServerDoesNewStore   FALSE;
                                writeMark (Store);
                                writeData (Omit, MakePL (DirectoryName, Name,
                                        UserName, Password,
                                        ThisType, EOLconvention));
                                writeMark (EOC);
                                }
                        else
                                {
                                printServerMsg (InputBuffer, TRUE);
                                return;
                                }
                        }
                }
        else
                {
                writeMark (Store);
                writeData (Omit, MakePL (DirectoryName, Name, UserName,
                                Password, ThisType, EOLconvention));
                writeMark (EOC);
                markbyte = readMark ();
                }

        /*
        *  Check whether server says we can store.
        */
        switch (markbyte)
                {
case No:                                        /* something was wrong */
                readUntil (InputBuffer, sizeof (InputBuffer), EOC);
                printServerMsg (InputBuffer, TRUE);
                fclose (fp);
                return;

default:                                        /* some other error */
                printf ("Internal error: [%s]\n", formatMark (markbyte));
                fclose (fp);
                break;

case Comment:                                   /* got a chatty server */
                readData (InputBuffer, sizeof (InputBuffer));
```

```
                printServerMsg (InputBuffer, FALSE);
                break;

case Here_is_property_list:                     /* this is for NewStore */
                readData (InputBuffer, sizeof (InputBuffer));
case Yes:                                       /* this is for Store mark */
                HashCount   0;
                markbyte = readMark ();         /* get EOC */
                DiskToNet (fp, InputBuffer, sizeof (InputBuffer),
                          ThisType == Binary ? EOL_Trans : localEOL,
                          PrintHash);
                writeMark (EOC);                /* indicate user done */

                /*
                *  Check if server indicated transfer ok.
                */
                switch (readMark ())
                        {
        case Yes:                               /* this is what we want */
                        break;

        case No:
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, TRUE);
                        break;

        default:
                        ErrorRecovery (ABORT, PROTOCOL,
                                "FTP protocol error - aborting connection");
                        return;
                        }

                if (expectMark (EOC) != SUCCESS)
                        ErrorRecovery (ABORT, PROTOCOL,
                                "FTP protocol error - aborting connection");
                }

        return;
}

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

Toggle flags to complement state.

*/

DoToggle (flag)
int *flag;
{
        if (*flag)
                {
                printf ("OFF\n");
                *flag   FALSE;
                }
        else
                {
                printf ("ON\n");
```

```
                *flag   TRUE;
                }
        return;
}

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

Allows the user to specify a default transfer type for each file.
Normally FTP figures the type automatically.

*/

DoType ()
{
        DoCommand (typeTable);
        putchar ('\n');
}

/*-----------------------------------------------------------------------*/

SetType (arg)                   /* called by DoCommand via DoType */
int arg;
{
        DefaultType = arg;
        return;
}

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

Any signals (previously setup by main) cause us to close our connection
with remote server and hopefully abort gracefully.

*/

SignalHandler (signum)
int signum;
{
        static char *signam[]   { "#0 ??", "SIGHUP", "SIGINT", "SIGQUIT",
                                  "SIGILL", "SIGTRAP", "SIGIOT", "SIGEMT",
                                  "SIGFPE", "SIGKILL", "SIGBUS", "SIGSEGV",
                                  "SIGSYS", "SIGPIPE", "SIGALRM", "SIGTERM" };

        putchar ('\n');

        if (FTPdebug)
                {
                if (signum <= 0 || signum > 15)
                        printf ("Signal number was: %d\n", signum);
                else
                        printf ("Signal was %s\n", signam[signum]);
                }

        printf ("Ftp program interrupted ..,");

        setwd (LoginDirectory);
        signal (signum, SIG_DFL);
        if (ScriptInput == FALSE)
```

```
            ResetTTY ();                /* reset to normal terminal modes */
        BSPclose (RTP_END);             /* close connection */
        printf (" connection closed\n");

        exit (BADEXIT);                 /* exit program */
}

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

Called on communication errors (or any others) to cleanup communications,
terminal modes, and exit.  There is a routine by the same name in server
task.

*/

ErrorRecovery (action, abortcode, abortstring)
int action;
int abortcode;
char *abortstring;
{

        switch (action)
                {
case ABORT:     printf ("\n%s\n", abortstring);
                BSPabort (abortcode, abortstring);
                break;

case CLOSE:     BSPclose (RTP_END);
                break;
                }

        if (ScriptInput == FALSE) ResetTTY ();
        setwd (LoginDirectory);
        exit (BADEXIT);
}
```

```
/*
        File:  ftp/userutils.c
        Date:  April 30 1984
        Author:  Mark Van Dellon
        Purpose:  Utilities for user part of FTP.

        MODIFIED BY     REASON

        AskType         - Get the type of file about to be transfered.
        GetLocalFileName- Get local file name from tty.
        MakePL          - Make a property list with filename and other
                          parameters given.
        PrintHash       - Print ! to indicate file transfer progess.
        printServerMsg  - Print a server reply to terminal.
        RemoteFileLookup- Determine if file already exists on remote host.
        YesNo           - Get a yes/no response from tty.
*/

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

Returns the type of the given file when we don't already know it.  We
first use the default, which might involve guessing the type from the
file itself.

*/

int AskType (filename)
char *filename;
{

        switch (DefaultType)
                {
case Guess:
                switch (GuessType (filename))
                        {
        case Binary:
                        printf ("%s, Type Binary\n", filename);
                        return (Binary);

        case Text:
                        printf ("%s, Type Text\n", filename);
                        return (Text);

        case Unknown:
                        printf ("%s, Type undetermined - use binary\n",
                                                        filename);
                        return (Binary);
                        }

case Text:
                if (GuessType (filename) != Text)
                        {
                        printf ("File is Binary, but you have made ");
                        printf ("the default Text.\n");
                        printf ("Information will be lost in this ");
                        printf ("transfer.\n");
                        }
```

```
                printf ("%s, Type Text\n", filename);
                break;

case Binary:
                printf ("%s, Type Binary\n", filename);
                break;
                }

        return (DefaultType);
}

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

Before retrieving a file to local host, check that it doesnot already
exist. If it does ask for overwrite confirmation and check access rights.
Return open file pointer or NULL.

*/

GetLocalFileName (defaultName)
char *defaultName;
{
        FILEDESC fd;
        char filename[sizeof (FileName)];
        FILE fp;

        /*
        *  If the user just hit a CR only use the default name.
        */
        if (ReadTTY (filename, sizeof (filename), Echo) == 0)
                {
                if (TermTTY () == ESC)          /* skip file */
                        {
                        printf (" -- skipped\n");
                        return (NULL);
                        }
                printf ("%s", defaultName);
                strcpy (filename, defaultName);
                }
        putchar ('\n');

        /*
        *  If file exists, ask for overwrite confirmation.
        */
        if ((status (filename, &fd) >= 0) && fd.f_type != UNUSED)
                {
                printf ("Local file %s already exists - overwrite", filename);
                if (YesNo ("? ") == FALSE)
                        return (NULL);

                /*
                *  Allow store only if file previously existed by current
                *  user or the filemodes give access priviledge or the
                *  current user is administrator.  If file did not previously
                *  exist, it must be able to be opened.
                */
                if (getuid () != 0 && fd.f_owner != getuid () &&
```

```
                                        (fd.f_modes & 0x200) == 0)
                        {
                        printf ("Access violation - can't open file\n");
                        return (NULL);
                        }
                }

        if ((fp = fileopen (filename, FORWRITE)) == NULL)
                {
                printf ("File open error, internal code %d\n", geterrno ());
                return (NULL);
                }

        return (fp);
}

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

Return a pointer to a property list made with filename and other
parameters given.

*/

char *MakePL (directory, filename, username, password, type, EOL)
char *directory;
char *filename;
char *username;
char *password;
int type;
int EOL;
{
        char *eols;
        static char propertyList[BUFSIZE];

        switch (EOL)
                {
case EOL_CR:    eols = "(End-of-Line-Convention CR)";          break;
case EOL_CRLF:  eols = "(End-of-Line-Convention CRLF)";        break;
case EOL_Trans: eols   "(End-of-Line-Convention Transparent)"; break;
default:        eols = "";                                     break;
                }

        sprintf (propertyList, "((%s %s)(%s %s)(%s %s)(%s %s)%s%s)",
                                "Server-Filename", filename,
                                "Directory", directory,
                                "User-Name", username,
                                "User-Password", password,
                                type==Text ? "(Type Text)" : type==Binary ?
                                "(Type Binary)(Byte-size 8)" : "", eols);

        return (propertyList);
}

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

Print a hash mark (if we should) to show progress of transfer.
```

```
*/

PrintHash ()
{
        if (HashMode)
                {
                putchar ('!');
                if (++HashCount % 79 == 0)
                        putchar ('\n');
                }
        return;
}

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

Print reply server gave (usually why a command cannot be done).

*/

printServerMsg (buf, CodeFlag)          /* print a reply */
char *buf;                              /* string with reply */
int CodeFlag;                           /* true if code byte is first */
{
        printf ("< %s", FormatReply (buf, CodeFlag));
        return;
}

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

Do a remote directory look-up to determine if file exists.
Returns NEWFILE, OLDFILE or FAILURE.

*/

RemoteFileLookup (FileName)
char *FileName;
{

        int markbyte;

        writeMark (Directory);
        writeData (Omit, MakePL (DirectoryName, FileName, UserName, Password,
                                        Unknown, Unknown));
        writeMark (EOC);

        while ()
                {
                switch (markbyte - readMark ())
                        {
case No:
                        readUntil (InputBuffer, sizeof (InputBuffer), EOC);
                        if (InputBuffer[0] == File_not_found)
                                return (NEWFILE);
                        /* bad error - print it */
                        printServerMsg (InputBuffer, TRUE);
                        return (FAILURE);
                        break;
```

```
case Comment:
                        readData (InputBuffer, sizeof (InputBuffer));
                        printServerMsg (InputBuffer, FALSE);
                        continue;

case Here_is_property_list:
                        readData (InputBuffer, sizeof (InputBuffer));
                        while (readMark () == Here_is_property_list)
                                readData (InputBuffer, sizeof (InputBuffer));
                        return (OLDFILE);
                        break;

default:
                        printf ("Internal error in RemoteFileLookup [%s]",
                                                formatMark (markbyte));
                        return (FAILURE);
                        }
                }
}

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

Ask a yes/no question until we get a response.
Returns true if the response was positive.

*/

YesNo (prompt)
char *prompt;
{

        while ()
                {
                printf (prompt);
                if (ScriptInput)        /* Always positive in script input */
                        {
                        printf ("Yes - script input\n");
                        return (TRUE);
                        }

                switch (tolower (getchar ()))
                        {
case 'y':
                        printf ("Yes\n");
                        return (TRUE);

case 'n':
case ESC:
                        printf ("No\n");
                        return (FALSE);

default:
                        printf ("'Y' or 'N' only");
                        }
                }
        /*  Normal return is based upon tty input  */
```

```
}
```

```
/*
        Date:  April 30 1984
        Author:  Mark Van Oellon
        Purpose:  Common transfer routines for server and user.

        MODIFIED BY     REASON
*/

#include <stdio.h>
#include "ftp.h"

#define WEGOT (ServerFlag ? "U: " : "S: ")
extern int ServerFlag;

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

Transfer data from disk to ethernet.

*/

DiskToNet (fp, rdbuf, rdbufsiz, EOL, ShowProgress)
FILE fp;                                        /* file to read from */
char *rdbuf;                                    /* read buffer */
unsigned long rdbufsiz;                         /* read buffer size */
int EOL;                                        /* text EOL convention */
int (*ShowProgress) ();                         /* FTP user show progress */
{
        unsigned long bytesread;                /* bytes read from net */
        register unsigned long bytetotal;       /* total bytes transmitted */
        char *convertbuf;                       /* ptr to eol cpnvert buf */
        unsigned long elapsedtime;              /* time required for Xmit */
        register int error;                     /* BSPwrite error */
        register int markbyte;                  /* read mark byte */
        register int puptype;                   /* BSP read type */
        unsigned long starttime;                /* time transfer started */

        /*
        *  If a convertion is going to be necessary than allocate
        *  some memory to do it in.
        */
        convertbuf   alloc (rdbufsiz << 1);

        writeMark (Here_is_the_file);

        /*
        *  This loop actually reads the file from the disk, determines
        *  if any conversion is necessary, and writes it to the BSP
        *  connection.
        */
        starttime - gettime ();
        for (bytetotal - 0; ; bytetotal += bytesread)
                {
                if ((bytesread = fileread (fp, rdbuf, rdbufsiz)) > 0)
                        {
                        if (EOL == EOL_Trans)
                                error = BSPwrite (BSP_DATA, rdbuf, bytesread);
                        else
```

```
                        {
                        bytesread = FixupEOL (rdbuf, convertbuf, EOL,
                                                bytesread);
                        error = BSPwrite (BSP_DATA, convertbuf,
                                                bytesread);
                        }

                if (error != OK)
                        {
                        if (ShowProgress) putchar ('\n');
                        if (FTPdebug)
                                printf ("Transmit timeout - %s\n",
                                                "connection aborted");
                        ErrorRecovery (ABORT, TIMEOUT,
                                "Transmit timeout, connection abort");
                        }

                if (ShowProgress) (* ShowProgress) ();
                }
        else
                break;
        }

    fclose (fp);

    if (ShowProgress)
        {
        elapsedtime = gettime () - starttime;
        if (elapsedtime <= 0) elapsedtime = 1;
        printf ("\n%d bytes, %d seconds, %d bits/sec\n", bytetotal,
                        elapsedtime, bytetotal * 8 / elapsedtime);
        }

    writeMark (Yes);
    writeData (OK, "Transfer complete");

    return;
}

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

Transfer data from ethernet to disk.

*/

NetToDisk (fp, rdbuf, rdbufsiz, EOL, ShowProgress)
FILE fp;                                    /* file to write to */
char *rdbuf;                                /* write buffer */
unsigned long rdbufsiz;                     /* write buffer size */
int EOL;                                    /* text EOL convention */
int (*ShowProgress) ();                     /* FTP user show progress */
{
    unsigned long bytesread;                /* bytes read from net */
    register unsigned long bytetotal;       /* total bytes transmitted */
    char *convertbuf;                       /* ptr to eol cpnvert buf */
    unsigned long elapsedtime;              /* time required for Xmit */
    register unsigned long pending;         /* data pending disk write */
```

```
        register int puptype;                   /* BSP read type */
        unsigned long starttime;                /* time transfer started */

        /*
        *  If a convertion is going to be necessary than allocate
        *  some memory to do it in.
        */
        convertbuf = alloc (rdbufsiz << 1);

        /*
        *  This loop actually reads the file from the BSP connection,
        *  determines if any conversion is necessary, and writes it
        *  to disk.
        */
        starttime = gettime ();
        for (bytetotal   0, pending   0; ; bytetotal += bytesread)
                {
                puptype = BSPread (rdbuf + pending, rdbufsiz   pending,
                                ONESEC * 45, &bytesread);
                pending += bytesread;
                switch (puptype)
                        {
case BSP_DATA:
                        if (pending >= (rdbufsiz / 3))
                                {
                                if (ShowProgress) (* ShowProgress) ();
                                diskwrite (fp, rdbuf, pending, EOL,
                                                convertbuf);
                                pending   0;
                                }
                        break;

case BSP_MARK:
                        /*
                        *  This should be end of transmission.  Check if
                        *  everbody was successful.
                        */
                        if (pending)
                                {
                                diskwrite (fp, rdbuf, pending   bytesread,
                                                EOL, convertbuf);
                                movebytes (rdbuf + pending   bytesread, rdbuf,
                                                bytesread);
                                }
                        free (convertbuf);
                        fclose (fp);
                        if (FTPdebug)
                                printf ("\n%s[%s]\n", WEGOT,
                                                formatMark (rdbuf[0]));
                        switch (rdbuf[0])
                                {
        case Yes:
                                if (ShowProgress)
                                        {
                                        elapsedtime = gettime () - starttime;
                                        if (elapsedtime <= 0) elapsedtime = 1;
                                        if (FTPdebug == FALSE) putchar ('\n');
```

```
                                printf ("%d %s, %d %s, %d %s\n",
                                        bytetotal, "bytes",
                                        elapsedtime, "seconds",
                                        (bytetotal * 8) / elapsedtime,
                                        "bits/sec");
                                }
                        return (MARKYES);

        default:
                        if (FTPdebug)
                                {
                                printf ("Unsuccesful transfer: ");
                                printf ("[%s]\n",
                                        formatMark (rdbuf[0]));
                                }
                        return (MARKNO);
                        }

case RTP_ABORT:
                if (pending)
                        {
                        diskwrite (fp, rdbuf, pending - bytesread,
                                                EOL, convertbuf);
                        movebytes (rdbuf + pending - bytesread, rdbuf,
                                                bytesread);
                        }

                free (convertbuf);
                fclose (fp);
                if (ShowProgress) putchar('\n');
                return (ABORTED);
                }                               /* switch */
        }                                       /* for loop */

        /*  Normal return is case BSP_MARK  */
}

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

Do conversion, if necessary, and write to disk.

*/

static diskwrite (fp, rdbuf, pending, EOL, convertbuf)
FILE fp;
char *rdbuf;
unsigned long pending;
int EOL;
char *convertbuf;
{

        if (EOL == EOL_Trans)
                filewrite (fp, rdbuf, pending);
        else
                {
                pending - FixupEOL (rdbuf, convertbuf, EOL, pending);
                filewrite (fp, convertbuf, pending);
```

```
                }

        return;
}
```

Appendix D: User Manual

FTP is a Pup-based File Transfer Program for moving files to and from a file system. The program comes in two parts:

1) An FTP Server, which listens for file transfer requests from other hosts and,
2) An FTP User, which initiates file transfers under control of either the keyboard or a typescript.

1. Concepts and Terminology

Transferring a file from one machine (or "host") to another over a network requires the active cooperation of programs on both machines. In a typical scenario for file transfer, a human user (or a program acting on his behalf) invokes a program called an "FTP User" and directs it to establish contact with an "FTP Server" program on another machine. Once contact has been established, the FTP User initiates requests and supplies parameters for the actual transfer of files, which the User and Server proceed to carry out cooperatively. The FTP User and FTP Server roles differ in that the FTP User interacts with the human user (usually through some sort of keyboard interpreter) and takes the initiative in user/server interactions, whereas the FTP Server plays a comparatively passive role.

The question of which machine is the FTP User and which is the FTP Server is completely independent of the direction of file transfer. The two basic file transfer operations are called "Retrieve" and "Store"; the Retrieve operation causes a file to move from Server to User, whereas Store causes a file to move from User to Server.

Transferring files to or from a file server involves establishing contact with FTP Server processes that run all the time on those machines. Hence, one may simply invoke the FTP subsystem and direct its FTP User process to connect to the machine.

In the descriptions that follow, the terms "local" and "remote" are relative to the machine on which the FTP User program is active. That is, we speak of typing commands to our "local" FTP User program and directing it to establish contact with an FTP Server on some "remote" machine. A Retrieve command then copies a file from the "remote" file system to the "local" file system, whereas a Store command copies a file from the "local" file system to the "remote" file system.

Furthermore, we refer to "local" and "remote" filenames. These must conform to the conventions used by the "local" and "remote" host computers, which may be dissimilar.

2. Calling the FTP Subsystem

A number of debug options are available when running FTP. These are probably not of any interest to the normal user. The general form of the command line to invoke FTP looks like:

FTP [-ftpdebug -bspdebug -pupdebug] <host-name>

The square brackets denote portions of the command line that are optional and may be omitted. The first token after the options is assumed to be a <host-name>. The User FTP will attempt to connect to the FTP Server in that host. After connecting to the server, an interactive keyboard command interpreter is started.

FTP permits only one user connection at a time. Ordinarily, host names should be the name of the machine you wish to connect to. Most machines have names which are registered in Name Lookup Servers. So long as a name lookup server is available, FTP is able to obtain the information necessary to translate a known host name to an inter-network address.

If the host name of the server machine is not known or if no lookup servers are available, you may specify an inter-network address in place of the <host-name>. The general form of an inter-network address is:

<network> # <host> # <socket>

where each of the three fields is a number (with leading 0 octal, with leading 0x hex, else decimal). The <network> number designates the network to which the Server host is connected. This may be omitted if the Server and User are known to be connected to the same network. The <host> number designates the Server host's address on that network. The <socket> number designates the actual Server process on that host; ordinarily it should be omitted, since the default is the regular FTP server socket. Hence, to connect to the FTP server running on host number 123 on the directly-connected Ethernet, you should say #123# (the "#" is required).

3. Keyboard Command Syntax

FTP's interactive command interpreter presents a friendly user interface. The user's standard editting characters (<LINEKILL> and <ERASE>), command termination (<ESCAPE>), and help (via "?") are available.

3.1. Keyboard Commands

The keyboard commands are insensitive to character case (i.e. QUIT or quit) and unambiguous abbreviations of command keywords (which in most cases amount to the first letter) are legal. However, when constructing typescript files, you should always spell commands in full, since the uniqueness of abbreviations in the present version of FTP is not guaranteed in future versions.

CD <directory name>

Causes <directory name> to be used as the default local directory in data transfer commands (actually it changes to that directory for the remainder of program execution or until another CD command is issued). Explicitly mentioning a directory in a file name overrides the default directory. Issue of a CD command in no way effects the DIRECTORY command.

DELETE <remote filename>

Deletes <remote filename> from the remote filesystem. The syntax of the remote filename must conform to the remote host's file system name conventions. After determining that the remote file exists, FTP asks you to confirm your intention to delete it. If the remote filename designates multiple files (the remote host permits "*" or some equivalent in file names), FTP asks you to confirm the deletion of each file.

DIRECTORY <directory name>

Causes <directory name> to be used as the default remote directory in data transfer commands (essentially it causes <directory-name> to be attached to all remote filenames that do not explicitly mention a directory). Specifying a default directory in no way modifies your access privileges. Explicitly mentioning a directory in a file name overrides the default directory. When first initiated, FTP User sets the directory to the current working directory.

EOL <convention>

Applicable only to files of type Text, EOL specifies the End-of-Line Convention to be used for transferring text files. The values for <convention> are CR, CRLF, and TRANSPARENT. The default is CR.

LIST <remote file designator>

Lists all files in the remote file system which correspond to <remote file designator>. The remote file designator must conform to file naming conventions on the remote host, and may designate multiple files if "*" expansion or some equivalent is supported there.

This information is only as reliable as the Server that provided it, and not all Servers provide all file properties. Much of this information is derived from hints, so do not be alarmed if it is sometimes wrong.

LOGIN <user-name> <user-password>

Supplies any login parameters required by the remote server before it will permit file transfers. FTP will default to use the user-name from the local host, without password.

When you issue the "Login" command, you should first type a valid remote host user-name. The command may be terminated by carriage return after entering the user-name to omit entering the user-password or a space which will prompt for the user-password.

The parameters are not immediately checked for legality, but rather are sent to the server for checking when the next file transfer command is issued.

QUIT

Returns control to the UNOS command interpreter after closing open connection.

RETRIEVE <remote filename>

Initiates transfer of the specified remote file to the local host. The syntax of <remote filename> must conform to the remote host's file system name conventions. Before transferring a file, FTP will prompt for a local-filename. At this point you may make one of three choices:

1. Type Carriage Return to cause the local filename to be the same as the remote-filename without directory or version.
2. Type Escape to indicate that the file is not to be transferred.
3. Type any desired local filename followed by Carriage Return. This filename must conform to local conventions.

Once the local-filename has been entered, FTP will tell you whether a file exists with that name. You then have the option to overwrite.

If the remote-filename designates multiple files (the remote host permits "*" or some equivalent in file names), each file will be transferred separately and FTP will ask you to make one of the above three choices for each file.

SHOW

Allows you to toggle a switch that shows the transmission progress being made in file transfer. The default is ON.

STORE <local filename>

Initiates transfer of the specified local file to the remote host. FTP will prompt for a remote-filename to which you should respond in a manner similar to that described under RETRIEVE except that if you supply a different filename, it must conform to the remote file system's conventions. The default remote filename is one with the same name and extension as the local file; the remote server defaults other fields as necessary. The directory is that most recently supplied in LOGIN or DIRECTORY commands.

TYPE <data type>

Forces the data to be interpreted according to the specified <data type>, which may be TEXT or BINARY. Initially the type is UNSPECIFIED, meaning that the source process should, if possible, decide on the appropriate type based on local information.

VERBOSE

Allows you to toggle switches which control operation of the FTP User. There is currently only one: DEBUG, which controls display of protocol interactions. Warning: this printout sometimes includes passwords.

! <system command>

Allows the user to temporarily escape the FTP user program and execute a system command. Once the command is completed, the FTP prompt will return. This feature avoids having to establish the connection with the remote host again.

4. Script Input

With the use of redirected I/O, the user may use typescript input to replace the interactive keyboard command interpreter. The command syntax is the same as the keyboard commands except all confirmation (file overwrite, delete, etc.) default to YES. If abnormal typescript termination is detected, the connection is closed and FTP User terminated.

5. File Property Defaulting

Without explicit information from the file system, it is often difficult to determine whether a file is Binary or Text, if Binary, what its byte-size is, and if Text, what End-Of-Line convention is used. The User and Server FTPs use some simple heuristics to determine the correct manner in which to transfer a file. The heuristics generally do the right thing in the face of incomplete information, and can be overridden by explicit commands from a human user who knows better.

The FTP protocol specifies a standard representation for a file while in transit over a network. If the file is of type Binary, each logical byte is packed right-justified in an integral number of 8-bit bytes. The bytesize is sent as a property along with the file. If the file is of type Text, each character is sent right-justified in an 8-bit byte. An EOL convention may be sent as a file property. The default is that <return> marks the end of a line.

5.1. File Types

FTP determines the type of a local file by reading it and looking for bytes with the high-order bit on. If any byte in the file has a high-order bit on, the file is assumed to be Type Binary, otherwise it is assumed to be Type Text. FTP will generate a warning, but allow you to send what it thinks to be a text file as type Binary, since no information is lost. It will refuse to send a binary file as type text.

Don't specify a Type unless you know what you are doing. The heuristic will not lose information.

5.2. End-of-Line Conventions

FTPs are expected to be able to convert text files between the local file system End-Of-Line (EOL) convention and the network convention.

As an escape to bypass conversion and checking, EOL convention 'transparent' tells both ends NOT to convert to network standard, but rather send a file 'as is'. This is included for Lisp files which contain internal character pointers that are messed up by removing line feed characters.

Don't specify an EOL convention unless you know what you are doing. If your text file is a Lisp source file, specify EOL convention 'Transparent'.

6. Abort Messages

Abort packets are fatal and cause the program to terminate after displaying the message.

The most common Abort message is "FTP connection timeout", generated when a server process has not received any commands for a long period of time (typically 5 minutes).

Two other Abort messages are "Receiver timeout" and "Transmitter timeout", both generated by failure to receive an anticipated packet within a specified period of time.