HP3000 TCP DESIGN DOCUMENT

Jack Sax and Winston Edmond

Bolt Beranek and Newman Inc.

July 1980

Sax and Edmond
Bolt Beranek and Newman Inc.
July 1980

Table of Contents

1 Preface..... 3
2 Introduction..... 4
3 Current HP3000 Structure..... 7
3.1 Processor Features..... 7
3.2 Network Interface Hardware..... 8
3.3 Operating System Software..... 8
3.4 Input/Output..... 10
3.5 Interprocess Communication..... 12
3.6 Existing INP Software..... 14
4 Protocol Software Architecture..... 16
5 System Protocol Software..... 20
5.1 Implemented Features..... 20
5.2 Software Architecture Overview..... 21
5.3 Control Structures..... 23
5.3.1 Network Resources Control Block..... 24
5.3.2 Foreign Host Control Blocks..... 25
5.3.3 Connection Control Block..... 26
5.3.4 Network Buffer Resources List Structures..... 26
6 User Process/TCP Interface..... 29
6.1 Interface Intrinsics..... 30
6.2 Flow Control Across the Interface..... 36
6.3 Interface Control Structures..... 36
6.4 Interface Control Algorithms..... 37
6.5 Windowing, Acknowledgment, and Retransmission 48
7 1822 Layer/INP Driver Communication..... 50
8 Protocol Software Buffering Scheme..... 52
8.1 Network Buffer Pool..... 54
8.1.1 Packet Compaction..... 55
8.1.2 Buffer Recycling..... 56
8.2 User Process Buffer Pool..... 58
9 Data Flow Through the Protocol Software..... 60
9.1 ARPANET to the User Level Data Flow..... 61
9.2 User Level to the ARPANET Data Flow..... 64

Sax and Edmond
Bolt Beranek and Newman Inc.
July 1980

1 Preface

This report describes a design implementation of ARPANET protocols on a Hewlett Packard HP3000 Series III computer system. Specific protocols to be implemented include a Transmission Control Protocol (TCP), Internet Protocol (IP), File Transfer Protocol (FTP), and TELNET Protocols. The reader is assumed to be familiar with the purpose of these protocols. The protocol software will run under the HP Multiprocessing Executive (MPE) operating system.

The designs reflect our current understanding of the environment and the tasks ahead and may be changed as we proceed with implementation.

2 Introduction

The overall purpose of this project is to modify the Hewlett Packard 3000/Series III computer system running the MPE operating system so that it converses with the ARPANET.

A layered protocol approach will be used in our implementation. Protocol layers one through four represent the system layers which are responsible for moving a message reliably from one Host to another. The next protocol layer consists of a number of applications protocols which determine the content and meaning of the messages exchanged.

Protocol levels one and two are X.25 LAP link access protocols. These protocols are implemented in microcode on the Intelligent Network Processor (INP) interface available from Hewlett Packard. Since the X.25 LAP protocols are different from the standard 1822 IMP Host protocols, a special X.25 IMP interface is used to link the HP3000 with the ARPANET. The interface divides standard 1822 packets into a number of X.25 frames and transfers each frame separately. The diagram in Appendix A shows the hardware configuration used to link the HP3000 to the ARPANET.

The next two protocol layers consist of the DOD standard Internet Protocol (IP) and the Transmission Control Protocol

(TCP). The Internet protocol provides communication between Hosts on different networks via gateways between the networks. The Transmission Control Protocol provides reliable transmission between Hosts and performs some Host-to-Host flow control.

The initial implementation will include three application layer protocols. One of these is the File Transfer Protocol (FTP), which allows a user to move files from one computer system to another. The second and third application layer protocols are User and Server TELNET. User TELNET gives the user a remote terminal capability by taking the characters from the local input device and sending them to the foreign host, and returning characters from the foreign host to the local output device (typically a terminal). The foreign host will have a Server TELNET process which acts as a pseudo-Teletype, with incoming network messages providing TTY input, and TTY output being sent to the network. The operating system treats the Server TELNET pseudo-Teletype like an ordinary terminal.

Most of the protocol software is new code, the major exception being the INP microcode which is supplied by Hewlett Packard. The programs will be written in HP's Systems Programming Language (SPL), which resembles PASCAL and allows intermixing of assembly code and compiled code. In addition to new code, implementation will require changes to the MPE

operating system code, which is also written in SPL.

3 Current HP3000 Structure

This section describes the HP3000 system with an emphasis on the features that affect the network software design. The description includes both the processor hardware and the operating system. Some of the operating system features described are not currently released by Hewlett Packard, but are about to be released or are part of the planned MPE IV release this fall.

3.1 Processor Features

The HP3000 CPU is a medium speed machine which uses a stack architecture. It executes uncomplicated instructions in one to two microseconds. Code and data are separate and thus all code is re-entrant. There are approximately 38 hardware registers which make up the state of the processor, most of which are associated with the stack (data) and the current instruction address (code).

Memory is divided into segments. A segment is a contiguous block of memory of any desired length up to 32K words. Individual segments are swapped in and out of memory as needed. Memory paging, a scheme which uses fixed size memory chunks as the basis for memory swapping, is not used in the HP3000. A

segment may be designated as code or data by the operating system.

3.2 Network Interface Hardware

The interface unit between the HP3000 computer and the ARPANET machines will be HP's Intelligent Network Processor (INP). This device consists of two boards located in the HP3000 main cabinet. It is a microprogrammed interface unit whose microcode is down-line loaded by HP3000 software. HP will supply the microcode to make the INP obey the X.25 LAP protocol and will supply the device driver necessary to access the INP.

The INP will be connected to a BBN C/30 (MBB) computer. This machine will convert the X.25 protocols from the INP into suitable ARPANET protocols.

3.3 Operating System Software

The operating system for the HP3000 is known as the Multiprogramming Executive System (MPE). It offers both batch and interactive job capabilities and allows multiple concurrent users of either type. It offers a file system which manages files on disk, magnetic tape and/or punched cards. Some I/O devices, such as the line printer, have spooler programs built in

to the system.

User programs are run as processes within MPE. Each process has associated with it a code segment and a stack (data) segment. In privileged mode, it may run in "split-stack mode", where it is allowed to have two data segments. The most common use of split-stack mode is to access tables in the operating system during system calls.

The design of MPE is greatly influenced by the HP3000 hardware architecture. MPE's organization relies heavily on operations which incur little processor overhead while avoiding operations which incur large amounts of processor overhead. The most striking example of this is the MPE's dependence on user processes for a large number of what would ordinarily be considered system functions. MPE avoids the use of "system" processes to perform these functions.

The design organization is a direct result of the stack architecture of the HP3000. The large number of status registers which must be saved when a new process is invoked makes process switching a very expensive operation. The time needed to perform a procedure call into a new segment of system code is typically less than the time to switch context from one process to another. Writing efficient code for this machine has thus led to organizing the system as relatively independent "utility"

routines callable by the user rather than as a collection of separate processes which manage I/O devices and system utilities. These operating system calls, called Intrinsics, are implemented as subroutine calls into system code segments. The program segments which implement the Intrinsics run in a privileged mode which allows them to directly access system tables and I/O device tables.

One notable side-effect of this design is that system resources such as I/O devices are assigned to only a single program and are not normally shared. This approach has allowed the system programmers to create a complex operating system without tackling the problems of interprocess communication and resource sharing. As will be discussed later, it also has a significant effect on protocol software design.

3.4 Input/Output

Input/Output operations typically consist of two steps. The first step is initiation of the desired operation. This involves checking to insure that access to the device is allowed (software protection), and issuing I/O instructions to the device to initiate the desired action. This step usually occurs as a result of an intrinsic call to the device handler code and thus is executed on the user's stack. The second step is the

operation completion handling. This may occur using either the Interrupt Control Stack (ICS) or the System Control Stack, neither of which is the user's stack. The choice of which stack to use depends on the specific device's function.

A consequence of this system design is that "system code" tends to be executed using the data stack of the first user process needing the function. If process 1 wants to do an I/O operation, it invokes a system procedure which knows how to manage that I/O device. If process 2 now wishes to invoke the same device, and if the device is capable of supporting more than one request concurrently, it invokes the same routine. To avoid multiprocessing hazards in issuing I/O commands, the system procedure first checks to see if it is the first invocation of itself -- if not, it queues the request and exits; if it is, it proceeds to issue the I/O instructions. If the request was queued, it is assumed that the first process will detect the newly queued request and process it also. The first process is thus performing system functions for the second, and all later, processes, and will be charged run time for doing their work. In practice, we do not expect this to be significant, but in theory, the first process could run indefinitely, even if its own request has long since completed.

3.5 Interprocess Communication

Interprocess communication under the current version of MPE is a problem. Only two techniques are currently available and neither of them is really satisfactory.

One technique that may be used is that of the logical device. It is chiefly used to accomplish multiplexing of physical devices. This facility is implemented by creating a new entry in the system's Device Information Table, and by creating a set of procedures which act as a device handler. The handler will be run in privileged mode.

Like other system device handlers, the procedures to manage the device are invoked directly by the user process, and the user's stack is used by the system code. This has the advantage of speed, since it avoids some process context switching.

There are a number of drawbacks to this technique. First, the Device Information Table entry must be maintained as if it were a real hardware device. This requires knowledge of all the MPE internal functions that might access this table. Furthermore, since these tables are system internal, they are subject to change with each new release of MPE. Use of the table requires Privileged Mode. Bugs in the code would have a greater chance of crashing the system. The greatest drawback is that

logical devices are still under development at HP, and are more than usually likely to change over time.

A new operating system feature, not yet released officially, that has been written for MPE is an interprocess communication method known at HP as message files. These correspond to Unix ports, and allow unrelated processes to communicate with one another. Each message file has one or more "reader" processes and one or more "writer" processes. During use, these files act as FIFO queues.

Message files are implemented using the file system. Read, write, and query commands are all patterned after the file system commands. The message file code is designed so that if readers and writers stay more or less in synchronization, disk I/O will not be needed. However, if the writers get far enough ahead of the readers, the message file will start being spooled out onto disk.

Message files are to be introduced as user level functions by HP, and, as such, their use will not change with new releases of the operating system. Code for this feature has already been implemented at HP and is available with both MPE III and the future MPE IV. They appear to be relatively easy to use and do not require knowledge of the internals of the operating system. Their chief drawbacks are that a process context switch is

required between writer and reader, and that some file system overhead is incurred.

Timeouts, as seen in message files, are another new HP function that will be available. The older version of timeouts simply suspended the process for a fixed amount of time, but did not allow the process to be awakened by the completion of an I/O event during its sleep. The new version is equivalent to setting a timer whose alarm may be awaited with the same IOWAIT intrinsic that awaits I/O completion. It allows a process to wait for either some I/O device operation completion or the passage of some maximum amount of time, whichever occurs first. Alternatively, a timeout could be used to insure that waiting for a specific event will terminate if the expected event does occur soon enough. There will be both user level and system internal ways of accomplishing timeouts.

3.6 Existing INP Software

The code to drive the INP is part of the CS/3000 Communications Software package from HP. It contains code to send and receive packets via the INP and code to manipulate the Device Information Tables. The code also allows the user to down-line load microcode into the INP memory. It contains intrinsics to open and close the line and to read and write

packets. The microcode allows the INP to support X.25 LAP protocols and also allows the INP to buffer up to eight 128-byte packets. These packets are read by CS/3000 as soon as possible in order to keep the INP from losing packets due to a lack of buffer space in the INP. This technique allows the INP to function as a full duplex device, even though the MPE operating system offers only a half duplex control mechanism in its software.

4 Protocol Software Architecture

The protocol software architecture is dictated by a set of design requirements and MPE operating system constraints. These requirements and constraints are summarized as follows:

- The new network software must be isolated from the existing operating system as much as possible. The isolation will allow any site to add or remove the network software with a minimum of effort. It will also make the network software less vulnerable to any changes HP makes to MPE.

- Efficient high speed network communications are extremely important because this TCP version will be used on a production rather than an experimental basis.

- One of the problems with MPE is that, though the operating system performs device assignment and access control for its I/O devices, the user process is responsible for operating the I/O device. MPE does offer intrinsics to operate common devices, but these are very low level operations. This I/O arrangement makes it difficult to control an asynchronous network interface. The protocol software architecture will therefore require at least one process which has exclusive control of the INP interface.

- One of the properties of these network protocols is that the message acknowledgments and retransmissions occur at a relatively high level -- in the Transmission Control Protocol in layer four. A moderate amount of time passes from the time the originating TCP queues the message for transmission and the receiving TCP gets the message. In order to prevent acknowledgment delays which in turn cause the foreign host to retransmit data, the software architecture should minimize the amount of time it takes for incoming data to move through the 1822, IP, and TCP protocols.

- With many network users and many connections concurrently in use, the network software must be able to handle the problems of multiplexing use of the network interface hardware. The interface on which the multiplexing takes place must support a number of simultaneous users in such a way that the behavior of any individual user does not affect data throughput of the other users.

In order to meet all of the design requirements and constraints described above, the HP3000 protocol software is implemented in a set of processes (see diagram in Appendix B). One process which will be called the system protocol process is responsible for maintaining the INP interface as well as

supporting the 1822, IP and TCP protocols. The rest of the processes, called applications protocol processes, support the user interactive network functions including FTP and TELNET.

The use of a single system protocol process is a key element in the protocol design. The system protocol process provides control over the INP interface by providing buffers and acting as multiplexer and de-multiplexer of network traffic to and from the INP. Use of a single process minimizes inter-protocol layer communication delays which in turn minimize the acknowledgment delays for incoming data. A single system protocol process makes it possible to use interprocess communication primitives to provide a uniform network interface for the applications level protocol processes.

User TELNET and User FTP protocols are to be implemented as ordinary user programs. They use the same system calls as any other network accessing program, but are written to provide a higher level command language for the user. As user programs, they execute in the user's address space with the privileges normally available to the user. The User TELNET and User FTP programs are re-entrant, with as many processes running this code as users wishing the service.

Server TELNET is a single process created as the system starts up or whenever the first need for it arises. De-

multiplexing of Server TELNET inputs is accomplished via a pseudo-teletype driver. The driver acts as the interface between the Server TELNET process and the Teletype handler.

The interface between application protocol processes and the system protocol process is through a set of TCP intrinsics. The intrinsics are designed to form a uniform interface between the user and the TCP. Actual data communication between a user process and the system protocol process is done with a combination of message files and direct buffer-to-buffer transfers. Message files are used to pass flow control information while the actual data transfer is made by copying data between user buffers and system protocol buffers. The combination of message files and buffer copy is used to take advantage of the flexibility of message files and the data rates achieved by direct data copy.

5 System Protocol Software

Since this TCP implementation is to be used on a production rather than an experimental basis, the design effort has concentrated on the efficiency rather than the sophistication of the protocol software. This is especially true of the system protocol software whose initial design includes only those features needed to support the FTP and TELNET protocols.

At the same time, the software design does allow for the future enhancement of the protocol software. There are no inherent design limitations which will prevent implementation of the more sophisticated TCP and Internet features.

5.1 Implemented Features

The specific TCP and Internet features to be implemented include the following:

- multiple connections to multiple hosts,
- flow control at the 1822, Internet, and TCP layers,
- error recovery,
- fair allocation of resources among all connections,
- handling of urgent data,
- surviving incorrect packets,
- datagram reassembly,
- routing,
- source quenching.

5.2 Software Architecture Overview

The system protocol software architecture reflects the need to avoid packet processing delays rather than a strict hierarchy between protocol layers. The system protocol software is implemented as a single process to allow the system protocol layers to share software resources for greater efficiency. The shared resources include subroutines which perform functions required by more than one protocol layer and a common buffer pool to optimize storage resources and to allow efficient communication between protocol layers.

Network traffic through the system protocol process takes different forms including 1822 packets, datagrams, and TCP segments. These various forms are generically referred to as

"packets". Packets are passed into the system protocol process from either an applications protocol process or the ARPANET interface. Packets from the ARPANET are passed into the system protocol process by intrinsic calls to the INP interface. User generated network packets are passed to the system protocol process by using a combination of message files and data buffers. Message files are used to transfer control and status information while data transfer is done with buffer-to-buffer copies between the user protocol data segment and the system protocol data segment.

All read and write commands are done without wait to allow the system protocol process to simultaneously multiplex I/O channels and process network packets. I/O multiplexing is implemented through the IOWAIT intrinsic. The system protocol process issues an IOWAIT intrinsic after it finishes processing a data packet. The IOWAIT intrinsic returns the file number of the I/O channel associated with an I/O completion wakeup.

When the number of free buffers falls below a prescribed limit, an attempt is made to free buffers through data compaction. The attempt begins with a search for datagram fragments and unacknowledged TCP segments which waste buffer space by using only a fraction of the available space in each buffer assigned to them. This lack of efficiency can be

particularly damaging because there is no guarantee that the data contained in the buffers will ever be processed. Wherever possible, datagram fragments are combined into a single datagram fragment and TCP segments are combined into a single segment to more efficiently utilize system buffers. Any buffers freed by this compaction process are returned to the freelist.

Network packets from both the user process and the ARPANET are processed along one of a number of data paths in the system protocol process. The actual data path taken depends on the type of data packet and, in the case of TCP segments, the state of its associated network connection. Packet processing is performed by a series of function calls which act as processing steps along the data path.

In order to avoid processing delays which can tie up system resources, each arriving data packet is processed through as much of the protocol software as possible. Processing of a packet is suspended only when the lack of some resource or some external event prevents further processing.

5.3 Control Structures

All of the status information both for individual network connections and for the system protocol software as a whole is

kept in a set of control blocks as well as in a number of buffer list structures as shown in Appendix C. The control blocks include a general network resources control block, a foreign host control block for each foreign host connected to the local host, and send and receive control blocks for network connection. The list structures include a network buffer free list, a TCP buffer aging list and an Internet buffer aging list.

5.3.1 Network Resources Control Block

The Network Resources Control Block contains the information needed to maintain the network buffer free lists and aging lists. This information includes pointers to the network buffer free lists and aging lists and a count of the buffers in each of the lists.

The information contained in the Network Resources Control Block is used by the protocol software to control the distribution of network buffers among the various lists. The information is scanned at various times to determine the allocation or disposition of a particular network buffer. The determinations occur when new buffers are allocated from the free list and when buffers containing TCP segments are about to be acknowledged. Decisions are made based on the number of free buffers available and the priority of the task requiring the

buffers.

5.3.2 Foreign Host Control Blocks

Foreign Host Control Blocks maintain flow control within the 1822 protocol layer. The block contains a counter for the number of outstanding 1822 packets sent to a single host. The counter includes all of the packets sent to the host on all sockets. The counter is incremented when an 1822 packet is sent and is decremented when either a RFNM or an Incomplete Transmission is received from the host.

The counter is used to prevent transmission of too many 1822 packets to a single host. All transmission from the host is blocked when the counter reaches the limit of eight outstanding 1822 packets for any foreign host.

The 1822 level flow control is actually implemented by the send side of the TCP software. The TCP checks the RFNM count in the connection control block before it tries to transmit a segment to the foreign host.

5.3.3 Connection Control Block

Each TCP connection has an associated control block. The control block contains data associated with the Transmission Control Block (TCB) along with other connection related information. Specific information included in the control block is as follows:

- a connection state variable used to maintain the connection state,
- the local port number of the connection,
- the TCP interface control block number associated with this connection,
- the file number of the private message file associated with this connection,
- the TCB data associated with the receive side of this connection,
- the TCB data associated with the send side of this connection,
- A pointer to any buffers containing unacknowledged data received on this connection.

5.3.4 Network Buffer Resources List Structures

Three list structures are used to maintain the network buffer resources shared by all of the sockets. These list structures include the free list and the two buffer aging queues.

The network buffer free list contains all of the network buffers currently available for use by any socket. These buffers

are allocated when new data comes in from either the network or a user protocol process.

The Internet Aging Queue is a list of active buffers assigned to blocked datagram fragments and complete datagrams. These buffers are the first to be reclaimed when there are no free buffers available. The Queue is sorted according to datagram age. All buffers which belong to the same datagram are combined into a single list structure. The datagram list structures are linked into the Internet Aging Queue with the least recently updated datagram always at the head of the queue. When a new datagram fragment comes in it is moved to the end of the queue along with all of the other fragments which belong to the same datagram.

The TCP Aging Queue is a list of buffers which contain at least parts of unacknowledged TCP segments. These buffers can be reclaimed when there are no free buffers and no buffers on the Internet aging list. The Queue is sorted by socket. All buffers which contain data for the same socket are combined in a buffer list and each buffer list is linked into the queue. The queue is sorted by the age of the data associated with each socket. Data belonging to the socket which has been inactive for the longest period of time is placed at the head of the queue so it can be recycled first. When a user process reads data from a

connection, all the network buffers still waiting to be read on that connection are moved to the end of the TCP aging list. This assures that data associated with an active TCP connection will not be recycled ahead of data associated with an inactive TCP connection.

6 User Process/TCP Interface

The user process/TCP interface is designed to meet two basic requirements. First, the interface must allow for high speed data transmission across the interface; this is especially important since this interface involves interprocess communication which could be delayed by excessive system overhead due to context switching and process scheduling. Second, the interface must isolate the system protocol process from any buffer overhead burdens caused by processing delays in the user process. System protocol process buffers are too valuable a resource to be locked into storing TCP segments which are waiting for response from a user process.

High speed data transmission across tser process/TCP interface is achieved by copying data directly from buffers in the user process to buffers in the system protocol process. The direct transfer is implemented with the move-to-data-segment and move-from-data-segment instructions provided by the HP3000.

The system protocol process is isolated from delays in the user process by making the user process responsible for buffering TCP data segments. Acknowledged incoming TCP segments, and TCP segments waiting to be transmitted over the ARPANET, are stored in buffers in the user protocol process. This use of user buffers serves two functions: it frees system protocol buffers

from being locked into storing TCP segments, and also gives the user process some control of network connection throughput. Throughput control is accomplished by allowing each user process to choose the amount of buffer resources it dedicates to each connection.

6.1 Interface Intrinsics

The TCP/user interface is implemented through a set of TCP intrinsics. These intrinsics allow the user process to create and use network connections with other processes on foreign hosts.

Seven intrinsics, TCPWAIT, TCPOPEN, TCPCLOSE, TCPRECEIVE, TCPSEND, TCPSTAT, and TCPABORT, provide the basic control functions needed to transfer data through the user process/TCP interface. Conceptually, the intrinsics allow the user to create network connections with other processes on foreign hosts. Each connection consists of a pair of sockets as defined in the TCP protocol document. Connections are created with a TCPOPEN intrinsic whose parameters define the sockets which make up the connection. After a connection is created, the user process uses the TCPSEND and TCPRECEIVE intrinsics to send or receive data. The TCPSTAT intrinsic allows the user to check the status of a connection.

Within the user process, connections are identified through the combination of a connection file number and a connection buffer. The connection file number is returned by a successful TCPOPEN call. The connection buffer is an integer array allocated by the user process. The buffer is initialized by the TCPOPEN intrinsic and is then passed as the first parameter to all of the other TCP intrinsics. It is the responsibility of the user process to maintain the association between the connection file number and the connection buffer.

The TCP I/O interface is entirely asynchronous so that a user process can queue any number of read or write requests to a particular connection. The user process has three limitations in this regard: first, it must provide the buffers associated with each TCP intrinsic call; second, the user process must keep track of which buffers are associated with each I/O call; and third, the user process cannot dequeue buffers until it has permission to do so from the system protocol process.

The user process uses a combination of the IOWAIT ane TCPWAIT intrinsic calls to keep track of I/O completion events. The IOWAIT intrinsic is invoked when the user process has completed processing all of the current data. When the IOWAIT intrinsic returns with a file number associated with a TCP connection, the TCPWAIT intrinsic is called to handle the I/O

completion. The TCPWAIT intrinsic uses the connection buffer to determine the cause of the I/O completion and then performs the appropriate I/O cleanup task and returns an I/O type code to the user process.

The specific calling sequences of the TCP intrinsics are given below:

TCPOPEN(TCPCBUF,FHIA,FP,A/P,LP[,BADDR]) opens a TCP connection

TCPCBUF - TCP Connection Buffer - This is a pointer to an integer array ten elements long which acts as the control structure for all network connections. The array is allocated by the user process before any TCP intrinsics are called.

FHIA - Foreign Host Internet Address - 32 bit address. This address may be obtained with the HOSTADDR intrinsic which takes the host name text string as a parameter and returns the 32 bit internet address. In the case of a passive open a zero address indicates a listen for any host.

FP - Foreign Port - a 16 bit port address for this connection at the foreign host. In the case of a passive open a 0 port indicates a listen from any port on a foreign host.

A/P - Active/Passive - a boolean flag used to indicate if this open is for a listen socket (passive) or for an active connection.

LP - Local Port - 16 bit local port id. This parameter is optional. If it is not specified, the TCP picks a free local port id from a reserved part of the name space.

BADDR - Buffer Address - an optional buffer used to give the foreign host a window for transmission. If the buffer is not provided, the connection is opened with a zero window size until the user process calls the TCPRECEIVE intrinsic.

returns - local connection name or error code of -1 if the connection failed. The local connection name is actually the file number of the private message file associated with this connection.

TCPCLOSE(TCPCBUF) closes a TCP connection

TCPCBUF - TCP Connection Buffer - same as in the TCPOPEN intrinsic.

TCPABORT(TCPBUF,BUFPTR) aborts a TCP connection

TCPCBUF - TCP Connection Buffer - same as in the TCPOPEN

intrinsic.

BUFPTR - Buffer Pointer - pointer to a list of buffers released by the TCPABORT call. A zero value indicates that no buffers were released.

TCPRECEIVE(TCPCBUF,BADDR,BCNT) reads data from a TCP connection

TCPCBUF - TCP Connection Buffer - same as in the TCPOPEN intrinsic.

BADDR - Buffer Address - address of user buffer for receiving network data.

BCNT - Byte Count - number of bytes to be transferred.

returns - an error code of -1 if the TCPRECEIVE failed.

TCPSEND(TCPCBUF,BADDR,BCNT,EOL) writes data to a TCP connection

TCPCBUF - TCP Connection Buffer - same as in the TCPOPEN intrinsic.

BADDR - Buffer Address - address of user buffer for sending network data.

BCNT - Byte Count - number of bytes to be transferred.

EOL - End Of Letter - a boolean flag to indicate that this buffer is an end of letter.

TCPSTAT(TCPCBUF,SBADDR) gives TCP connection status

TCPCBUF - TCP Connection Buffer - same as in the TCPOPEN intrinsic.

SBADDR - Status Buffer Address - address of user buffer for receiving network data.

TCPWAIT(TCPCBUF,BUFPTR,DATAPTR) returns the result of a previous TCP intrinsic call.

TCPCBUF - TCP Connection Buffer - Same as in the TCPOPEN intrinsic.

BUFPTR - Buffer Pointer - used to return a pointer to a buffer list released by an I/O event. A zero pointer indicates that no buffers were released.

DATAPTR - Data Pointer - pointer to the first data element within a buffer returned by the intrinsic to a TCPRECEIVE intrinsic.

returns - a code indicating the type of I/O completed. A list of the I/O codes is given in Appendix D.

6.2 Flow Control Across the Interface

Flow control across the user process/TCP interface is implemented through the use of message files. The message files act as control channels to transmit flow control and status messages between the user process and the TCP. Each connection makes use of two message files. A general input message file is used to transmit control messages from user processes to the TCP. All user processes share the same general input message file. Each connection also uses a private message file to convey control and status information from the system protocol process to the user process.

The control messages passed between the user process and the system protocol process are short data buffers. These buffers contain the message type along with other information associated with the particular command. The formats for each of the message types is shown in Appendix D.

6.3 Interface Control Structures

Each network connection has an associated TCP interface control block. These blocks include a set of pointers and data segment numbers used to keep track of buffers within both the user process and the system protocol process. The pointers

contain buffer addresses relative to the beginning of the stack data segment for each process. A diagram of the TCP interface control block is given in Appendix E.

The control blocks are maintained in a separate data segment shared by both the user and system protocol processes. The data segment is initialized by the system protocol process when it starts up. The initialization of the data segment divides it into a number of control blocks. Individual control blocks are initialized by the TCPOPEN intrinsic. Responsibility for releasing the control blocks is shared among the TCPCLOSE, TCPABORT, and TCPWAIT intrinsics.

6.4 Interface Control Algorithms

The specific functions performed by each of the network I/O intrinsics are as follows:

TCPOPEN

1. The TCPOPEN intrinsic software searches for a free TCP connection interface control block and initializes it.

2. The TCPOPEN software creates a private message file with a unique file name. The unique file name is formed out of the prefix "TCP" and the id number of the TCP

interface control block.

3. The TCPOPEN software sends an OPEN CONNECTION command message to the TCP via the general input message file. The message includes all of the TCPOPEN parameters and the id number of the TCP interface control block.

4. The TCPOPEN software makes a read request with timeout on the private message file. If the read times out, the TCPOPEN software sends an ABORT CONNECTION command to the TCP, deletes the TCP interface control block, and returns an error code to the user process. The connection buffer provided as a parameter to TCPOPEN is used as the read buffer.

5. The TCP software reads the open command from the general input message file and uses the information to create a connection control block. The TCP software also initiates the connection protocols specified in the command message.

6. The TCP software sends an OPEN CONFIRM message back to the user process via the private message file created by the TCPOPEN intrinsic software. The OPEN CONFIRM message will fail if the user process is destroyed by a user abort. If this occurs, the TCP software takes

responsibility for cleaning up the TCP interface control block as well the connection control blocks.

7. The TCPOPEN software reads the OPEN CONFIRM message from the private message file. The TCPOPEN software initiates a read without wait on the private message file. The connection buffer is again used as the read buffer.

8. If the user provides a read buffer as the last parameter to the TCPOPEN intrinsic, a read operation is initiated. The TCPOPEN software attaches the buffer to the TCP interface control structure and sends a RECEIVE message to the TCP via the general input message file. The TCP uses this message to set the size of the connection window.

9. The TCPOPEN software returns the file number of the private message file to the user process.

TCPCLOSE

1. The TCPCLOSE software marks the connection closed bit for the send side in the TCP interface control block. The TCPCLOSE software checks to see if there are any data buffers waiting to be read by the TCP. If there are no data buffers, the TCPCLOSE software sends a CLOSE

CONNECTION command to the TCP. If the receive side is marked closed and there are no buffers waiting to be read, the TCPCLOSE intrinsic software deletes the TCP interface connection control block.

2. The TCPCLOSE software returns to the user process.

3. When the TCP receives a connection close command from the user process it sends a FIN to the foreign host and marks the send side of the connection as FINWAIT-1. When the TCP receives an ACK of the close the foreign host, it marks the send side of the connection as FINWAIT-2. If the receive side of the connection is marked closed, the TCP deletes the connection control block.

4. If the TCP receives a FIN from the foreign host, it marks the receive side of its connection as closing. When all data and the FIN sent by the foreign host are ACKED, the TCP sends a NETCLOSE command to the user process and marks the receive side of the connection closed. If the send side is also marked as closed, the TCP deletes the connection control block. The close message sent to the user process is processed by the TCPWAIT intrinsic.

TCPABORT

1. The TCPABORT software sends an ABORT CONNECTION command to the TCP via the general input command file. The TCPABORT software releases the TCP interface control block and deletes the connection's private message file.

2. The TCPABORT software returns to the user process. The return includes a pointer to a list of user buffers which were assigned to the connection.

3. When the TCP receives an ABORT CONNECTION command from the user process it sends a reset to the foreign host, deletes any unacknowledged data it has for this connection, and deletes the connection control block.

4. If the TCP receives a reset from the foreign host, it deletes all of the data waiting to be transmitted to the user process and sends a NETABORT message to the user process via the private message file. The NETABORT message is handled by the TCPWAIT intrinsic.

TCPRECEIVE

1. The TCPRECEIVE software checks to see if the receive side connection closed flag is set. If the flag is set, the TCPRECEIVE software returns a connection closed

indication to the user process. It is up to the user process to close the send side of the connection and clean up the connection buffer if it has not done so.

2. If the connection is open, the TCPRECEIVE software attaches its read buffer to the TCP interface control block and sends a RECEIVE message to the TCP. The message is used to indicate to the TCP that the user has made a buffer available to the connection. The TCPRECEIVE software returns to the user process.

3. When the TCP receives the user's read message, it checks to see if it has any unacknowledged segments waiting to be transferred to the user process. If it has no segments, it uses the RECEIVE message to increase its receive window size. If the TCP has segments waiting for transfer, it transfers as much of the data as possible to the user process. All transferred data is immediately acknowledged to the foreign host. The TCP sends a PENDING RECEIVE message to the user process to advise it of the transfer of data. This message is processed by the TCPWAIT intrinsic.

4. If the TCP receives data from the foreign host, it checks to see if the user process has assigned any free buffers to this connection. If there are free buffers,

the TCP copies as much of the data it receives as possible into the user buffers and acknowledges the copied data to the foreign host. Any data which is not copied is maintained on the TCP aging list where it is stored until it is transferred to a user process buffer or discarded. The user process is informed of the data transfer through a PENDING RECEIVE command message via the private message file. This message is received by the TCPWAIT intrinsic.

TCPSEND

1. The TCPSEND intrinsic checks to see if the connection is still open. If the connection is marked closed, the TCPSEND returns an error code to the user.

2. If the connection is still open, the intrinsic software attaches the user supplied data buffer to the TCP interface control block. The TCPSEND software sends a SEND message to the TCP via the general input message file. The TCPSEND software now returns to the user process.

3. The TCP software, on receiving the data SEND message, checks to see if it can send the data to the foreign host. The decision on whether to send the data is made

by checking the following conditions:

- the foreign host has advertised sufficient window space,
- the number of outstanding RFNMs for all connections to the foreign host is less than eight,
- the amount of data waiting to be sent is sufficient to warrant a data packet. This condition prevents single byte segments from being sent out over the ARPANET. The TCP waits until it has at least 10 bytes of data before transmitting it out to the ARPANET.
- the user has specified an EOL.

If the TCP decides to send the data, it prepares a network packet and copies as much of the user data as it can transmit into the network packet. The data transfer is made directly from the list of user buffers queued by the TCPSEND intrinsic to the message packet buffer. All buffers filled by the data transfer are marked as filled and appended to the filled buffer list.

4. After the TCP has transferred all of the data from the user buffers, it checks the TCP interface control block. If the send side of the connection is marked closed, the TCP sends a Fin to the foreign host. If the receive side is also closed, the TCP sends a NETCLOSED command to the user process.

5. After the data is transmitted, the TCP sets a

retransmission timer.

6. If the TCP receives an acknowledgment from the foreign host, it updates the TCP interface control block to reflect the acknowledgment, turns off the timer, and sends a DATA SENT message to the user process via a connection private message file. The message contains the number of bytes acknowledged. This message is processed by the TCPWAIT command. If only some of the data is acknowledged, the TCP resets the timer for the unacknowledged data.

7. If the TCP does not get an acknowledgment from the foreign host and the connection times out, it again reads as much data as it can from the user buffer and sends it out as a network packet.

TCPSTATUS

1. The TCPSTATUS software checks to see if the connection is still open. If it is closed, it returns a connection closed code to the user process.

2. The TCPSTATUS command checks to see if there is an outstanding status request by the user process. If there is, it returns an error code to the user process.

3. If there is no pending status request, the TCPSTATUS software attaches the status request buffer to the TCP interface control block and sends a STATUS message to the TCP via the general input message file. The TCPSTATUS returns to the user process.

4. When the TCP receives the status request message, it formulates a status message and copies it into the user's status buffer attached to the connection buffer. The TCP then sends a status complete message to the user process via the connection private message file. The message from the TCP is processed by the TCPWAIT intrinsic.

TCPWAIT

1. The TCPWAIT software checks the message received from the TCP.

2. If the message is a NETCLOSE command, the TCPWAIT software checks if the send side of the connection is closed and there is no data waiting to be sent to the TCP. If the send side is closed and there is no pending TCP data, the TCPWAIT software deletes the TCP interface control block. If there is data waiting to be transmitted, the TCPWAIT software marks the receive side

of the connection closed. In either case, TCPWAIT returns a connection closed code to the user process. It is up to the user process to decide when to close the send side of the connection, if it has not already done so. If there are any user buffers still assigned to this connection, they are returned to the user process at this time.

3. If the message is a NETABORT command the TCPWAIT software deletes the TCP interface control block and returns a connection abort code to the user process. Any buffers associated with connection are also returned in a list structure through the buffer pointer parameter.

4. If the message is a PENDING RECEIVE command, the TCPWAIT returns the pointer to the head of the first data buffer, the first data byte, and a byte count. Since the data may be returned in a number of linked buffers, it is up to the user to follow the buffer links. As the user process reads the data it should check each buffer's header. Completely filled buffers marked with a zero in the in use field can be reclaimed by the user process.

5. If the message is a DATASENT message, the TCPWAIT

software checks the acknowledgment count and releases as many buffers as it can from the send buffer list. The released buffers are linked in a list and the buffer pointer parameter is set to point to the first buffer in the list. The TCPWAIT software returns a data acknowledgment code to the user process.

6. If the message is a STATUS COMPLETE message, the TCPWAIT software sets the buffer pointer parameter to point to the status buffer and returns a status complete command code to the user process.

6.5 Windowing, Acknowledgment, and Retransmission

The receive window size and data segment acknowledgment are completely dependent on the number of buffers the user process allocates to a connection. The receive window size of a connection is always set to the amount of free buffer space the user process allocates to the receive side of a connection. Acknowledgments of incoming TCP segments are limited to those sequence numbers which fit in the receive window. Acknowledgments are sent as soon as data is copied from the system protocol buffers to the user protocol buffers.

The initial retransmission algorithm is extremely simple. The first retransmission of unacknowledged data occurs 3 seconds after the original transmission. The second retransmission occurs 6 seconds after the first. The third and successive retransmissions occur 15 seconds after the previous retransmissions.

7 1822 Layer/INP Driver Communication

Communication between the system protocol process and the INP driver is implemented with four intrinsics: IOPEN, ICLOSE, IREAD, and IWRITE. These intrinsics are modified forms of the CS/3000 intrinsics. Their function is to open a connection to the INP network processor and to transmit data buffers to and from the INP. The IREAD and IWRITE intrinsics are always done without wait. The IOWAIT intrinsic is used to determine the completion of an I/O request.

Initialization of the INP interface begins with an IOPEN call which initializes the interface software. This is followed by four IREAD intrinsic calls to initialize buffers for incoming network packets. Four pending buffers should allow enough buffering to catch all of the incoming data without tying up too many network buffers.

The following is a summary of the commands used to communicate between the protocol process and the INP driver.

- IOPEN()

  returns error code on failure. Possible failure modes include failure to find the INP microcode file, failure to load the microcode file in the INP, and a hardware failure in the INP.

This command initializes the connection between the protocol process and the INP. The initialization includes activating the INP and loading its microcode.

- ICLOSE()

This command closes the connection between the INP and the protocol process when the network software is brought down.

- IREAD(buffer)

This intrinsic passes an empty buffer to the INP driver. The buffer is queued to a DIT with an ATTACHIO command. Control then returns to the protocol process.

- IWRITE(buffer)

This intrinsic passes a full buffer to the INP DIT with the ATTACHIO command. Control is returned after the buffer is attached to the DIT. The buffer is released when the calling process receives an interrupt indicating I/O completion.

8 Protocol Software Buffering Scheme

Data buffer management is the most important component of the network protocol software. Data buffers perform the key functions of data storage and data communication within the protocol software. These functions have complex and conflicting requirements which must be balanced by the buffer management scheme. An understanding of the buffer management scheme therefore begins with an understanding of its requirements.

First, data buffers must be considered a scarce resource shared by a number of competing "interests" within the protocol software. These "interests" include the various protocol layers as well as individual network connections within the TCP layer. The major problem is how to effectively allocate buffer resources among these interests. This problem becomes particularly difficult when there is a shortage of buffers.

An examination of the buffer requirements shows that they fall into two categories. The first category includes those buffers used to support general network functions. This includes buffers used in the 1822 and Internet protocol layers. These buffers are assigned to move and store data in these protocol layers without regard to particular network connections. The second category includes those buffers used by the TCP protocol to support specific connections.

The distinction between the two buffer categories is important because buffer use within each category is controlled by a different set of events. The use of buffers assigned to the general network functions can be controlled by the system protocol software. Buffers are processed through the Internet and 1822 protocol layers without regard to the behavior of user processes and their affect on individual connections. Buffers assigned to the connection specific network functions in the TCP and higher level protocol layers are greatly affected by events which occur in user processes. The rate at which data is accepted from or transmitted to the ARPANET by user processes is totally unpredictable. This unpredictability makes it difficult for the system protocol process to effectively assign buffer resources to individual network connections.

Two buffer pools are used to separate those buffering functions shared by all network connections from the connection specific buffering functions. A network buffer pool, maintained by the system protocol process, is used to support the 1822 and Internet and some TCP buffering functions. A user buffer pool, maintained by each user protocol process is used to support connection dependent buffering functions for the TCP and higher level protocols.

8.1 Network Buffer Pool

The network buffer pool supports the following specific functions.

- movement of network packets from the INP driver 1822 and Internet protocol layers;
- storage of Internet datagram fragments in the Internet protocol layer;
- storage of unacknowledged TCP segments which do not fall into the current window;
- movement of network packets from the TCP layer through the Internet and 1822 layer to the INP driver.

The network buffer pool is maintained on the system protocol process stack where it can be accessed easily by the various system protocol layers. All of the buffers in the pool are the same size to minimize the amount of software overhead needed to maintain the buffers. The buffer size is matched to the maximum frame size (128 bytes) which may be transmitted over the X.25 link between the INP and the ARPANET IMP.

The size choice is the result of two constraints. First, the buffers used to catch incoming data must be large enough for the largest incoming network packet. The packets are transferred directly into memory by the INP hardware making it impossible for a packet to cross buffer boundaries. Second, the single size buffer scheme limits the amount of software overhead needed to

maintain the buffer pool.

The single size buffer scheme does not waste buffer space because the buffer size is well matched with the data it processes. The 128 byte buffer size allows room for all of the protocol headers and a small amount of data. Messages with more data will use multiple buffers. The buffers are large enough to hold a significant amount of data yet small enough to limit the waste caused by partially filled buffers.

No attempt is made to assign network buffers to any particular protocol layer or task. Buffers are allocated either when data is read from the ARPANET or when the TCP layer sends data out to the ARPANET.

8.1.1 Packet Compaction

When the total number of network buffers in the free list falls below a set value, a data compaction algorithm is invoked. This algorithm searches for partially filled buffers used to store Internet datagram fragments and unacknowledged TCP segments waiting to be transferred to a user process. These buffers are chosen because processing of the data in them is indefinitely suspended. Compaction of the data in these buffers allows some of the buffers to be released to the free list.

8.1.2 Buffer Recycling

A buffer recycling algorithm is invoked when the system protocol process runs out of free network buffers. The algorithm allows buffers to be reused even if they currently contain data. The mechanism works by identifying which data buffers can be reused without losing irreplaceable information. These buffers are sorted in a priority scheme wallows the least important buffers to be recycled first. The buffer recycling scheme prevents one socket from tying up too much of the network buffer resources. It also helps assure a supply of network buffers even under heavy load conditions.

The buffer algorithm scheme divides network buffers into three categories: free buffers, in-use buffers, and aging buffers. Free buffers are available for immediate use by any protocol layer and are maintained on a common free list. In-use buffers are buffers bound to messages currently being processed and cannot be used for any other purpose. Aging buffers are used in messages where processing is suspended for any number of reasons. These buffers are placed in one of two special lists arranged in order of decreasing age. That is, message buffers which have been blocked for the longest time are at the front of the queue, while the message buffers which were most recently blocked are at the end of the queue.

There are two points in the protocol software where messages may be blocked. The first point is in the Internet Protocol software. Fragmented datagrams cannot be passed on to the TCP and can be blocked indefinitely if one or more of the fragments which make up the datagram is lost. A duplicate datagram may eventually be transmitted leaving the fragmented datagram in a suspended state. The second blocking point is in the TCP software. Unacknowledged segments sent by a foreign host remain suspended in the TCP until they are transferred to a user process buffer. Any segments which are not transferred to a user process will remain blocked indefinitely.

Buffer recycling is implemented through buffer aging lists which are adjuncts to the buffer free list. When an incoming message is blocked, its buffers are attached to the end of one of two aging lists. Buffers bound to datagram fragments are attached to one aging lists while buffers bound to TCP segments waiting to be read by user processes are attached to the second aging list.

The aging lists are periodically manipulated when a new datagram fragment comes in or when a user process receives some data from the TCP. Buffers associated with the particular datagram fragments or TCP segments are moved to the end of their respective aging lists. This helps assure that any data which

has a chance of being used will not be thrown away.

The buffers bound to fragmented datagrams are recycled first because they are the most expendable. Blocked datagram buffers may be a part of datagrams which have been retransmitted and passed on to the TCP. When the blocked datagram buffers are exhausted the buffers bound to blocked TCP segments are used. These buffers contain the unacknowledged segments which have not been claimed by a user process. The assumptions here are that the user process will never claim these segments and that they are expendable.

User Process Buffer Pool

The user process is responsible for maintaining a set of fixed length buffers for passing the user data to the TCP. Each buffer must include a four byte header along with 80 bytes of data space.

The first element of the header is used as either a byte count or a full buffer marker. The count is used by the TCPSEND intrinsic to indicate the number of data bytes in the buffer. The TCPRECEIVE intrinsic uses the buffer full marker to identify buffers which may be reclaimed by the user process.

The second element in the array header contains a list pointer. This pointer is maintained by the intrinsic software and should not be altered by the user process until the buffer is released.

9 Data Flow Through the Protocol Software

Data flow through the protocol software is effected through a series of tests and function calls. The tests check the type and processing state of each packet while the function calls perform specific operations on each packet. These operations include such things as creating or checking headers and queueing or de-queueing packet buffers.

Whenever possible, network packets are processed through all of the system protocol layers without interruption. This helps increase throughput by minimizing two important parameters. First, the amount of buffering required to process data is decreased because all network buffers associated with a packet are released when the packet has passed through the protocol software. Second, the time between the receipt of a packet from the ARPANET and the transmission of an ACK is reduced.

There are a number of instances when the processing of a packet can be interrupted within the system protocol process. This can occur when the lack of some resource or event prevents further processing. Examples of this are as follows:

- Internet datagram fragments waiting for reassembly;
- TCP segments from a foreign host waiting to be read by a user process;
- TCP segments from a user process waiting for window

allocation before being transmitted to the ARPANET;

- TCP segments from a user process already sent to the ARPANET but waiting for an acknowledgment.

9.1 ARPANET to the User Level Data Flow

Data packets come in from the network via a DMA interface to the INP network processor. Incoming data is first transferred into the protocol process via network buffers passed to the IREAD intrinsic which places a read request on the DIT queue of the INP. An arriving network packet is placed in the network buffer by the INP driver. The system protocol process is notified of each I/O completion through the IOWAIT intrinsic.

Processing of network packets begins when an IOWAIT call returns on completion of an IREAD intrinsic. The first processing step is to link the network buffers which contain the pieces of an 1822 packet.

The next processing steps are performed by the 1822 protocol software. If this is a normal data packet the 1822 header is removed and the data packet is passed as a datagram to the Internet Software. The transfer is done by calling a sequence of Internet protocol routines with the datagram as a parameter.

The Internet software checks the datagram header for integrity and then tries to find the proper address for this datagram. If the datagram is not for the local host it is routed to the proper ARPANET Host and the network buffers are returned to the free list.

If the datagram is a fragment of a larger datagram it is linked to any existing fragments waiting to be processed. If the new fragment does not complete the incoming datagram, the fragment is placed in an aging buffer queue next to the youngest buffer in the partially complete datagram. At this point all processing on the incoming datagram is suspended until the rest of the datagram fragments arrive.

A complete datagram is stripped of its Internet header and sent to the TCP software as a data segment. The TCP performs a number of functions on incoming segments: first the segment header is checked to see if it belongs to a known socket -- if it does, any acknowledgment information from the header is taken to update the socket status; next, the segment is checked to see if it falls within a window -- if it is not within the window (or a reasonable approximation thereof), the segment is discarded and its buffers are returned to the free list.

Accepted TCP segments are transferred into the user buffers. The transfer is initiated by the user process which provides a

buffer through the TCPOPEN or TCPRECEIVE intrinsic. A command message sent via the general input message file is used to inform the system protocol process that a buffer is available. The system protocol process transfers as much of its segment as possible to the user buffer. The user process is then notified of the data transfer via the connection's private message file. Only the transferred portions of the segments are acknowledged to the foreign host. Any portions of segments which do not fit in the receive window are stored in the TCP aging queue.

The acknowledgment may be sent in a number of ways. If the same network connection has an outgoing packet waiting for transmission, the acknowledgment information is added to the outgoing packet. If there is no pending outgoing packet, a check is made to see if there is sufficient unacknowledged data to warrant an acknowledgment packet. If there is enough information, a separate acknowledgment packet is generated and transmitted out to the ARPANET as if it were a normal message. If the number of unacknowledged segments is insufficient to justify an acknowledgment packet, the pending acknowledgment bit in the TCB is set and a timer is started. If the timer runs out, an acknowledgment packet is sent regardless of the number of unacknowledged segments.

9.2 User Level to the ARPANET Data Flow

Transfer of data from the user process out to the ARPANET begins with a NETSEND intrinsic call. The intrinsic software sends a message to the system protocol process to inform it that it has data to send. The system protocol process tests the state of the connection to see if data transmission is feasible. The following are sufficient conditions for data transmission out to the ARPANET:

- enough data has collected to justify transmitting it to the foreign host;
- the user process has specified an EOL in the data transmission;
- there are fewer than eight outstanding 1822 protocol packets waiting for RFNMs to the foreign host;
- the waiting data falls within the foreign host's window.

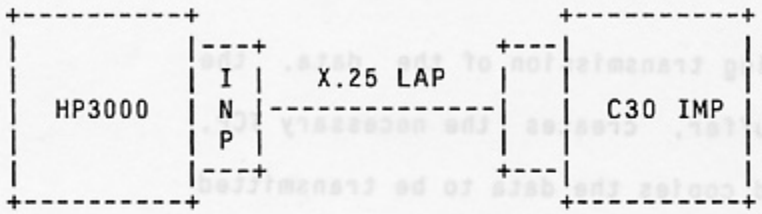
If the state of the connection does not allow a transmission to occur, a request-to-send data flag is set in the connection control block. When the connection state changes due to some external event, a check is made to see if the new state allows the transmission of waiting data. An example of such an event is the arrival of a RFNM from a foreign host; in this case all of the connections to the foreign host are checked for data waiting for transmission. The connection with data which has been waiting for the longest time is processed first. An attempt is

made to combine as many of the waiting TCP segments as possible into one data transfer to increase the amount of data transmitted.

If there is nothing blocking transmission of the data, the TCP software allocates a buffer, creates the necessary TCP, Internet, and 1822 headers, and copies the data to be transmitted from the user buffer to the system's buffer. The TCP header will include any acknowledgment information for data received on the return socket associated with the connection.

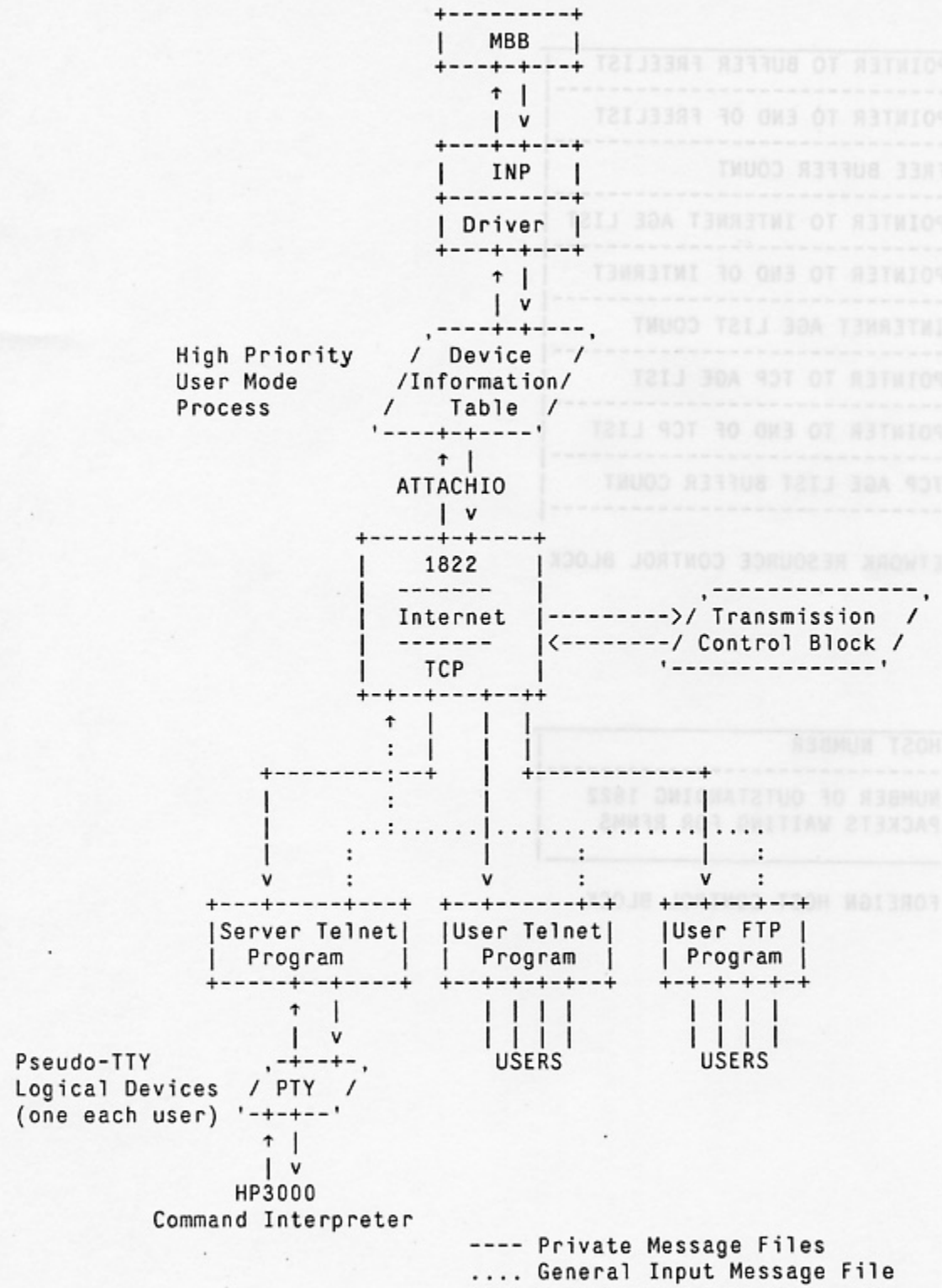
In order to assure proper transmission of the TCP segment a retransmission sequence is started. A retransmission timer is started to wake up the protocol software when a retransmission is needed. If a timeout occurs, the segment is retransmitted as soon as the state of the connection allows it. The necessary conditions for a retransmission are the same as those for the original transmission. If the segment is partially acknowledged, the data left in the retransmission queue is only that data represented by the unacknowledged sequence numbers.

APPENDIX A - HP3000 to ARPANET Link

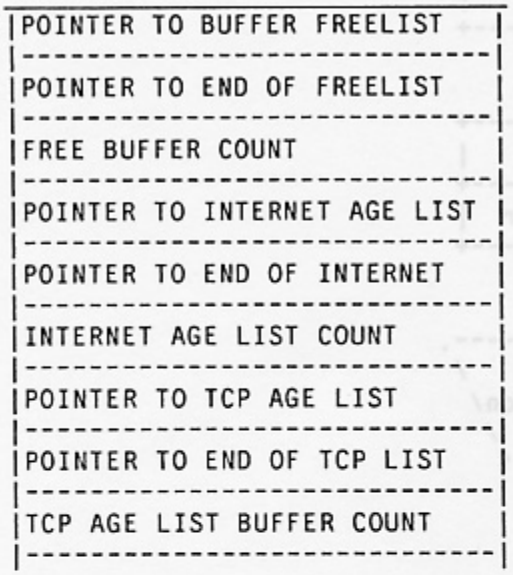


Sax and Edmond
Bolt Beranek and Newman Inc.
July 1980

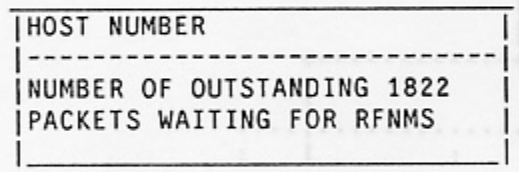
APPENDIX B - Protocol Software Organization



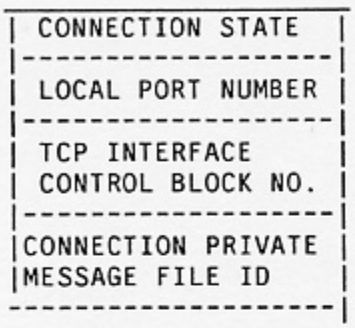
APPENDIX C - Control Structures



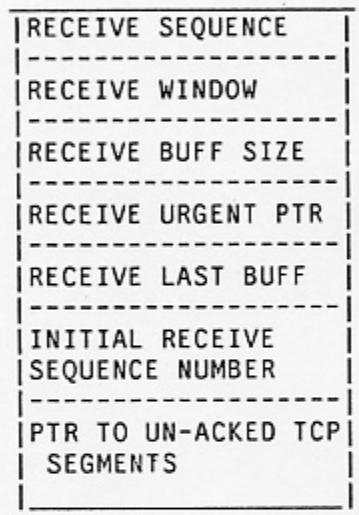
NETWORK RESOURCE CONTROL BLOCK



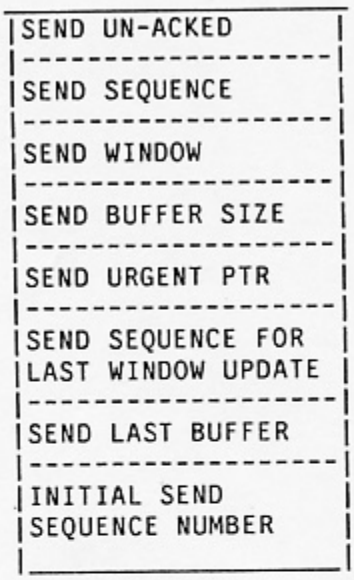
FOREIGN HOST CONTROL BLOCK



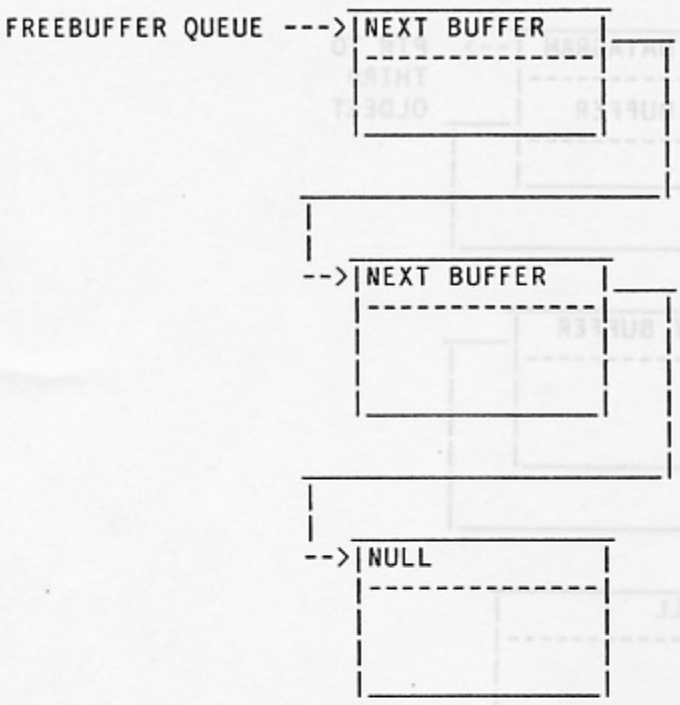
GENERAL INFORMATION SECTION
OF THE CONNECTION CONTROL
BLOCK



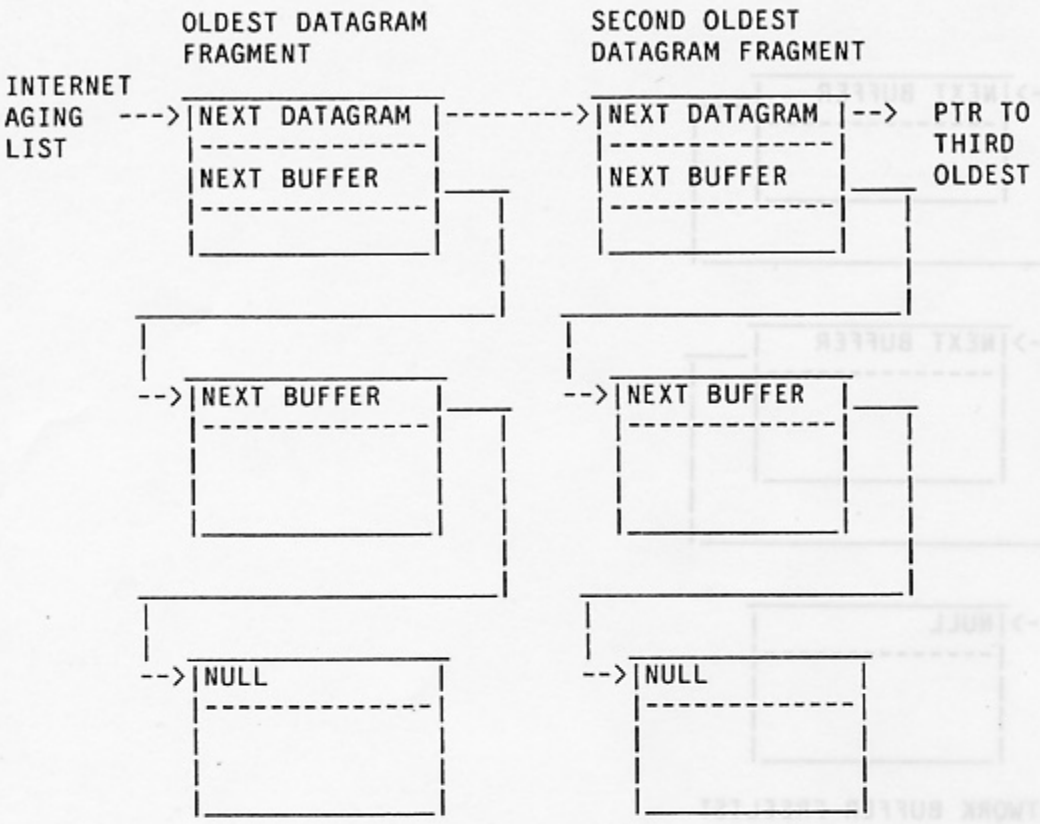
CONNECTION RECEIVE DATA



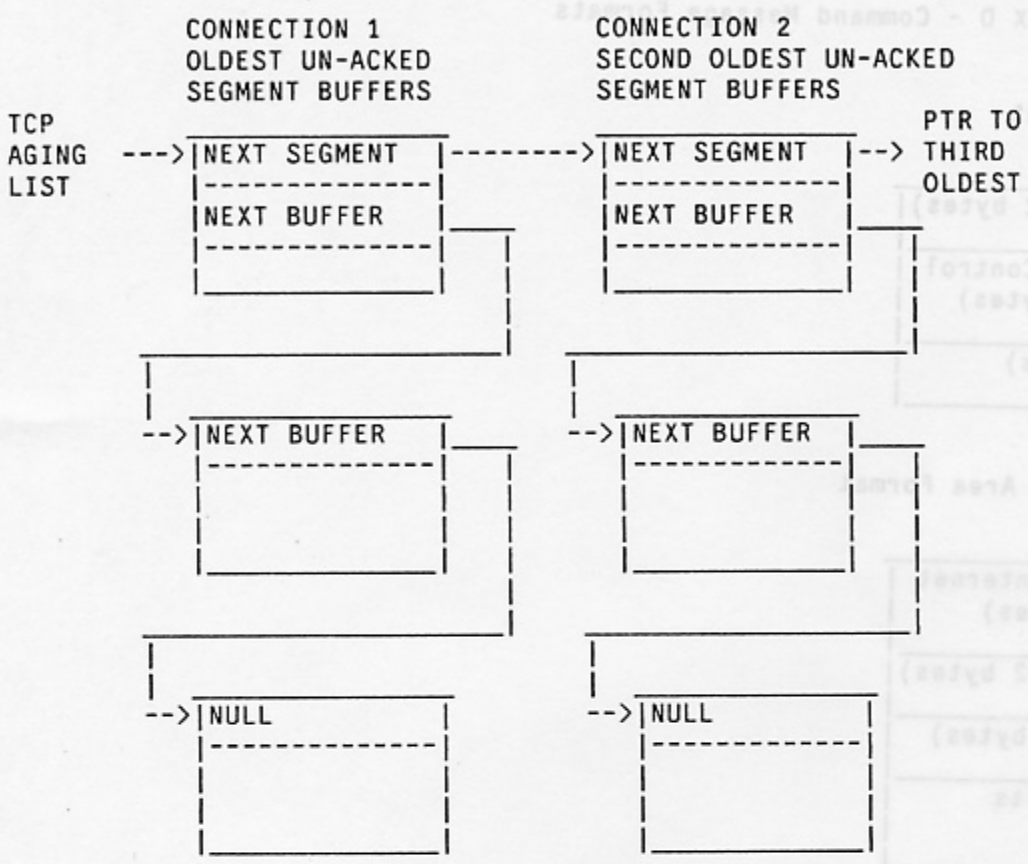
CONNECTION SEND DATA



NETWORK BUFFER FREELIST



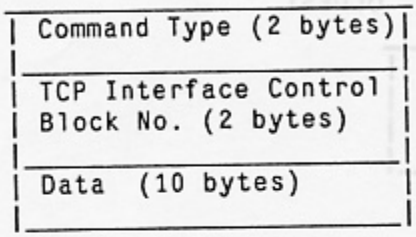
INTERNET AGING LIST



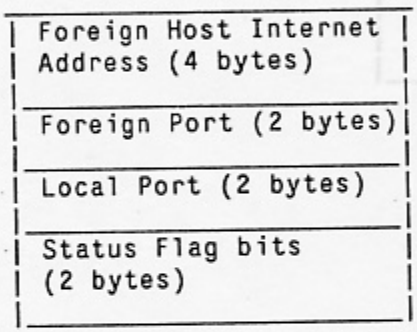
TCP AGING LIST

APPENDIX D - Command Message Formats

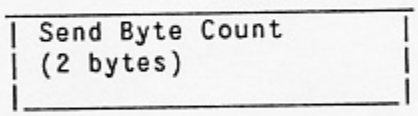
General Message Format



OPEN CONNECTION Data Area Format



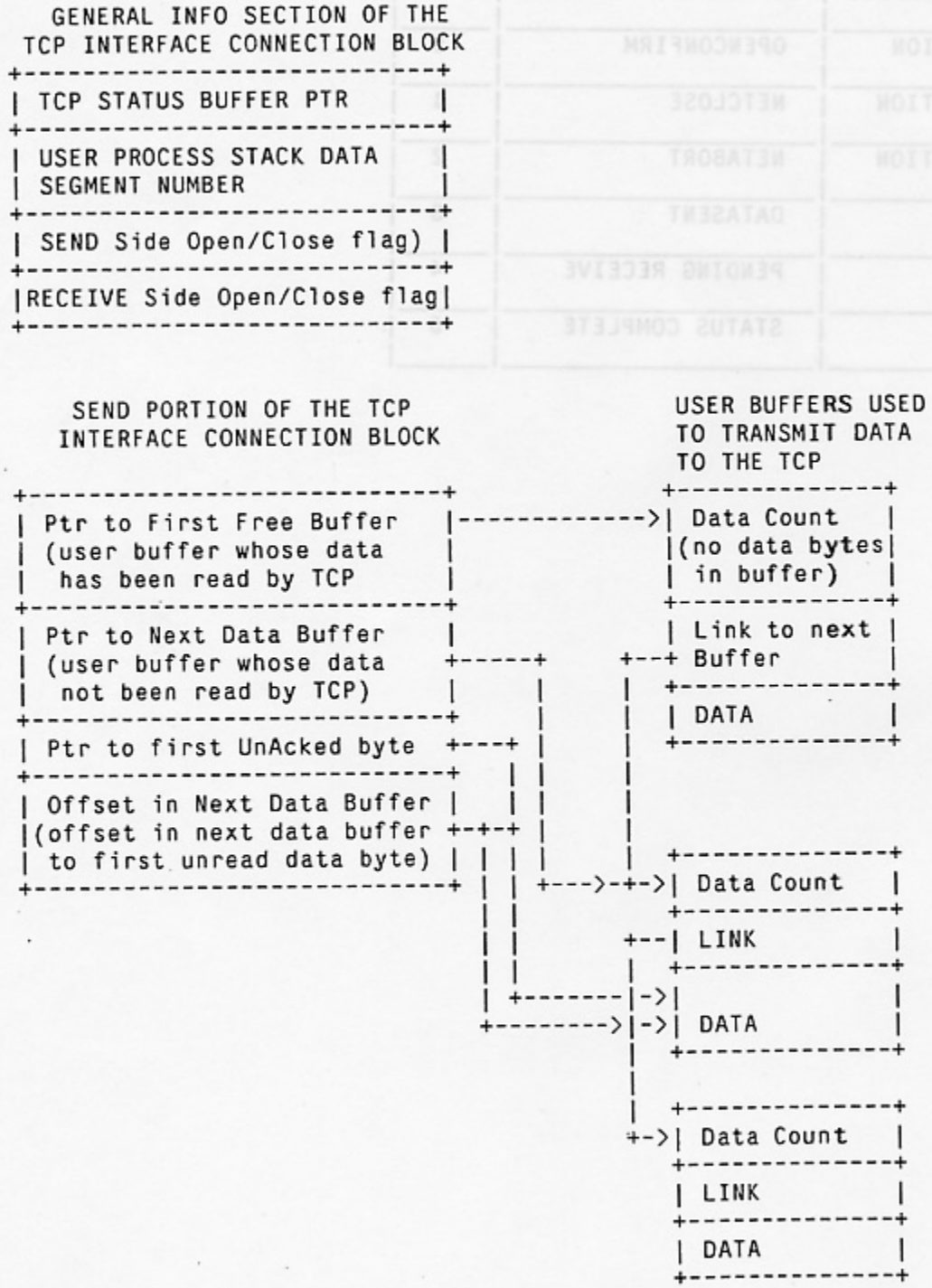
SEND Command Data Area Format

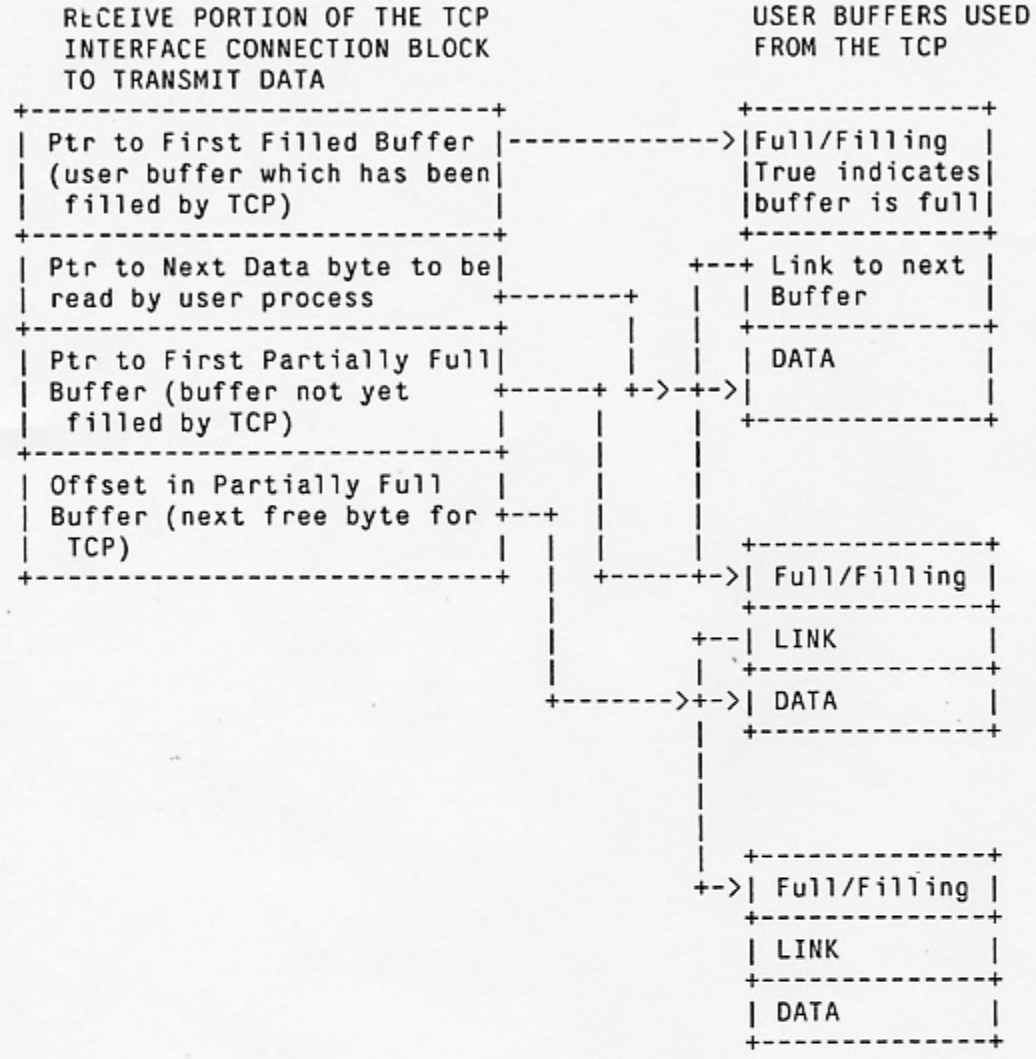


Message File Command Codes

User To TCP Command	TCP to User Command	Code
OPEN CONNECTION	OPENCONFIRM	0
CLOSE CONNECTION	NETCLOSE	1
ABORT CONNECTION	NETABORT	2
SEND	DATASENT	3
RECEIVE	PENDING RECEIVE	4
STATUS	STATUS COMPLETE	5

APPENDIX E - TCP Interface Control Block




RECEIVE PORTION OF THE TCP INTERFACE CONNECTION BLOCK TO TRANSMIT DATA
USER BUFFERS USED FROM THE TCP
Ptr to First Filled Buffer (user buffer which has been filled by TCP)
Full/Filling True indicates buffer is full
Ptr to Next Data byte to be read by user process
Link to next Buffer
DATA
Ptr to First Partially Full Buffer (buffer not yet filled by TCP)
DATA
Offset in Partially Full Buffer (next free byte for TCP)
Full/Filling
LINK
DATA
Full/Filling
LINK
DATA