Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document Number 233
CCTC-WAD Document Number 7515

Network Research in Front Ending and Intelligent Terminals

Experimental Network Front End Software Functional Description

August 1, 1977

CAC Document Number 233
CCTC-WAD Document Number 7515

Networking Research in Front Ending
and Intelligent Terminals

EXPERIMENTAL NETWORK FRONT END SOFTWARE FUNCTIONAL DESCRIPTION

Steven F. Holmgren
Elizabeth Kasprzycki
David C. Healy
Paul B. Jones

Prepared for the
Command and Control Technical Center
WWMCCS ADP Directorate
Defense Communications Agency
Washington, D.C. 20305

under contract
DCA100-76-C-0088

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

August 1, 1977

Approved for Release: Peter A. Alsberg
Peter A. Alsberg, Principal Investigator

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Background

At present, the storage, maintenance, and processing requirements of host-resident network software represent a significant burden on WWMCCS hosts. Offloading a major portion of this network software to a mini-computer front end, interposed between a host and a network, will reduce the extent and complexity of host-resident software. As a result, host performance will improve considerably. Proper design of front-end and interface software will also yield improved security.

Under contract DCA100-76-C-0088, the Center for Advanced Computation (CAC) of the University of Illinois at Urbana-Champaign is investigating the capabilities of network front ends. As part of that contract, an experimental network front end (ENFE) is being developed to interface a WWMCCS H6000 to the ARPA Network and to conduct experiments with the proposed ARPANET Host-to-Front-End Protocol.

The operating system for the front end is a hybrid Unix operating system. Unix, a general purpose operating system developed by Bell Telephone Laboratories, supports time-sharing and has facilities such as editors, compilers and word processors. The CAC has already constructed an ARPA Network Control Program (NCP) for the Unix system. The NCP is a system software module that implements the ARPA Network Host-Host and Initial Connection Protocols. These are the basic protocols used in communication across the ARPA Network. This document describes the further enhancements to Unix and the software modules that are necessary to support investigation of the front-end concept.

1.2 Organization

This document is logically divided into four major sections. Each section defines a major software module in the network front end (Channel Protocol module, Host-Host Service module, Server Virtual Terminal Service module, and Program Access Service module). Multiple chapters defining each module have been incorporated due to the extensive number of subroutines comprising each specific service.

The first section (chapters five and six) defines the internal hierarchy and actions of the Channel Protocol module. The second section (chapters seven, eight, and nine) defines a "typical" service and service support. Specific descriptions of the remaining three modules are adpated from this description of the "typical" service. Thus, section three (chapter ten and 11) defines the Host-Host Service module, section four (chapters 12, 13, 14, and 15) defines the Server Virtual Terminal Service module, and section five (chapters 16 and 17) defines the Program Access Service module.

2. FRONT-END HARDWARE ARCHITECTURE

2.1 Mainframe

The front-end mainframe is a Digital Equipment Corporation (DEC) PDP-11/70 computer with 128K words of memory and disk storage. The PDP-11/70 is the largest and fastest of the PDP-11 family of mini-computers. The PDP-11/70 is being used to determine the full extent of PDP-11/Unix front-end capabilities. Measurement of these capabilities in this effort and in a follow-on effort will determine the potential of other smaller members of the PDP-11 family.

2.2 Hardware Interfaces

Hardware interfaces to the ARPANET, the H6000, and to terminals will be provided.

2.2.1 ARPANET Interface. A DEC IMP-11A ARPANET interface will be used to connect the 11/70 to the ARPANET. The IMP-11A is a DEC standard product.

2.2.2 H6000-PDP-11/70 Interface. The H6000 and PDP-11/70 will be linked by a pair of interfaces, similar to ARPANET host-to-IMP interfaces, connected so that the outputs of one interface are the inputs of the other. The H6000 interface will be an Asynchronous Bit Serial Interface (ABSI) which uses two Common Peripheral Interchange (CPI) channels on the H6000 I/O multiplexor. The PDP-11/70 interface will be a general purpose, full duplex, direct memory access (DMA) interface. The interfaces will use ARPANET IMP-to-host data transmission techniques to communicate with each other.

3. FRONT-END SOFTWARE ARCHITECTURE

3.1 General Description

The offloaded network software can be thought of as a set of services provided to host processes or to users. These services allow the network and the various hosts connected to the network to be conveniently used.

3.2 Host-to-Front-End Communications

A basic mechanism must be provided to support communication between host processes and front-end services. This mechanism is the Host-to-Front-End Protocol (HFP), which is defined in CAC Document 219 (ARPA Request for Comments (RFC) 710). The HFP specification distinguishes two protocol layers: the channel protocol and the process-to-service protocols.

3.2.1 Channel Protocol.

By means of the channel protocol, logical channels are set up between host processes and the front-end services, and messages are transmitted on these channels. Provisions are made for flow control and for out-of-sequence signaling. The channel protocol defines five types of HFP Messages:

1. BEGIN, which sets up logical channels;
2. END, which terminates logical channels;
3. TRANSMIT, which transmits data;
4. SIGNAL, which provides a means for

synchronizing the ends of a logical channel, for interrupting the other end, and for flushing data from the other end of the channel; and

5. EXECUTE, which provides a means for passing service-specific information "out-of-band" (i.e. outside of the strict sequencing required for the TRANSMIT Messages).

Each Message type can be either a Command (requesting that the action defined by the Message be taken) or a Response (indicating whether the action was taken and, if not, providing some explanation). The HFP Specifications use the capitalized word, Message, to refer to these Message types. This convention will be followed throughout the document.

3.2.2 Channel Protocol module. The front end contains a software module, the Channel Protocol module (CPM), which manages the logical channels and serves as a bi-directional multiplexor. The host also contains a CPM which similarly manages the other ends of the logical channels. The CPM performs several functions.

1. It de-multiplexes HFP Messages arriving from the host interface and passes them to the appropriate service modules in the front end.

2. It accepts input in the form of HFP Messages from the service modules and multiplexes them to the host interface.

3. It controls the flow of HFP TRANSMIT Commands into the front end.

4. It performs error checking at the Channel Protocol level.

3.3 Process-to-Service Communications

Communications between a host process and a front-end service may be divided into three stages:

1. communications between the host process and the host CPM (described in CSC Document No. R493700056-2-1, "Host to Front-End Processor Protocol Interface Functional Description"),

2. communications between the host CPM and the front-end CPM (described in CAC Document No. 220, "H6000 Software Specifications"), and

3. communications between the front-end CPM and a front-end service (described in section 7).

Figure 1 on the next page illustrates the process-to-service communication architecture.

PROCESS-TO-SERVICE COMMUNICATION ARCHITECTURE

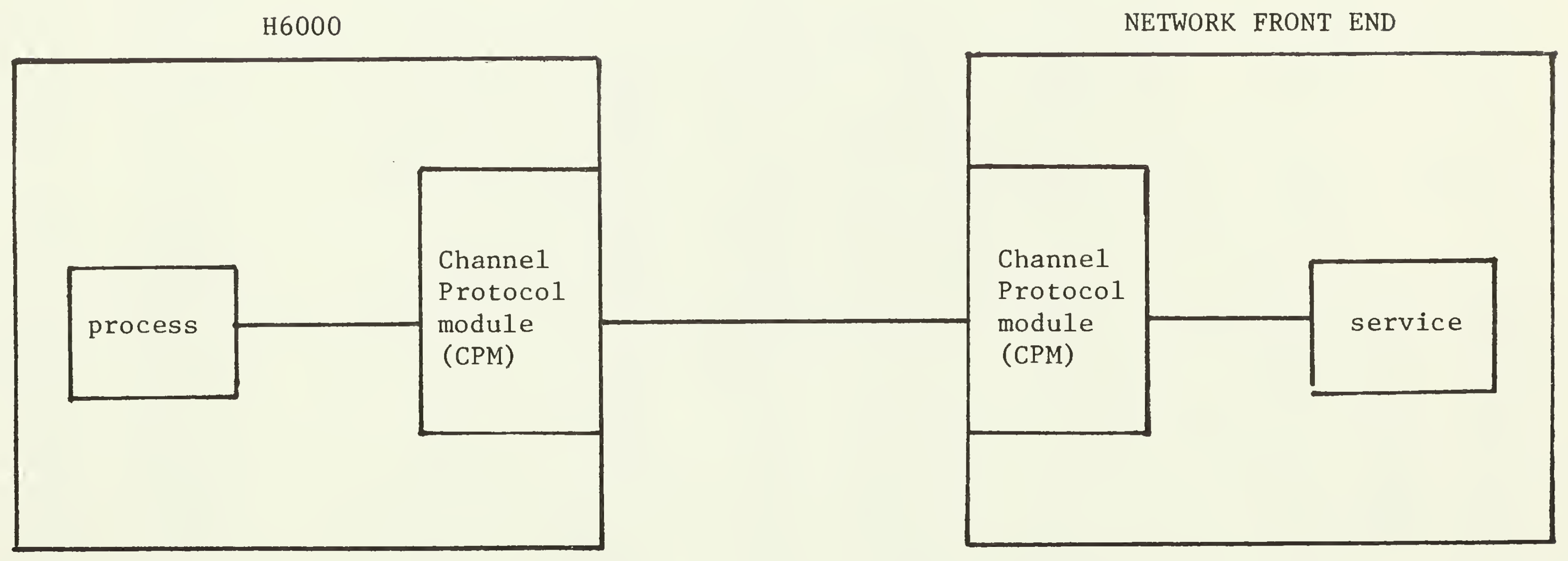


Figure 1

3.3.1 Process-to-Service Protocols. The process-to-service protocols specify the content, sequencing, and type of HFP Messages by which host processes communicate with front-end services. The process-to-service protocols implemented to date are:

1. ARPANET Host-Host Process-to-Service Protocol (CAC Technical Memorandum No. 8Ø),

2. Program Access Process-to-Service Protocol (CAC Technical Memorandum No. 81), and

3. Server Virtual Terminal Process-to-Service Protocol (CAC Technical Memorandum No. 82).

3.3.2 Service Structure. Each front-end service implements one process-to-service protocol. All front-end services execute within their own address spaces; e.g., as user level programs.

Each program is structured as a finite state machine accepting HFP Messages from the front-end. HFP Message inputs are generated by processes in the host requesting action from the front-end services. I/O completion event inputs are generated in response to service-initiated device I/O operations. Each input is associated with an HFP logical channel. The input type and current channel state determine the immediate action and next channel state. Most actions result in the transmission of data to another destination and in the generation of an HFP Response indicating the success or failure of the action. The specific action taken is process-to-service protocol dependent.

3.3.3 Host-Host Service module. The ARPANET Host-Host Service (HHS) module enables programs running in the host to use the ARPANET NCP in the front end. It implements the ARPANET Host-Host process-to-service protocol. The HHS module performs several functions, using the ARPANET NCP in the front end.

1. It opens and closes ARPANET connections to hosts on the network.

2. It passes data between the host and foreign hosts on the network.

3. It maintains connection status information.

3.3.4 Program Access Service module. The Program Access Service (PAS) module enables programs running in the host to execute arbitrary programs in the front end. It implements the Program Access process-to-service protocol. The PAS module performs the following functions.

1. It enables programs on the host to log in to and log out of the front-end system.

2. It enables programs on the host to run programs on the front end (for example, the User Telnet program).

3. It passes data between programs on the host and programs running on the front end.

3.3.5 Server Virtual Terminal Service module. The ARPANET Server Virtual Terminal Service (SVTS) module enables programs on the host to be accessed by terminals connected to other hosts on the ARPANET. It implements the ARPANET Server Virtual Terminal process-to-service protocol. It also implements the ARPANET Telnet protocol described in NIC Document No. 15372. The SVTS module performs the following services.

1. It opens and closes ARPANET connections to hosts on the network.

2. It passes data between the local host and foreign hosts on the network, transforming the data in accordance with Telnet protocol.

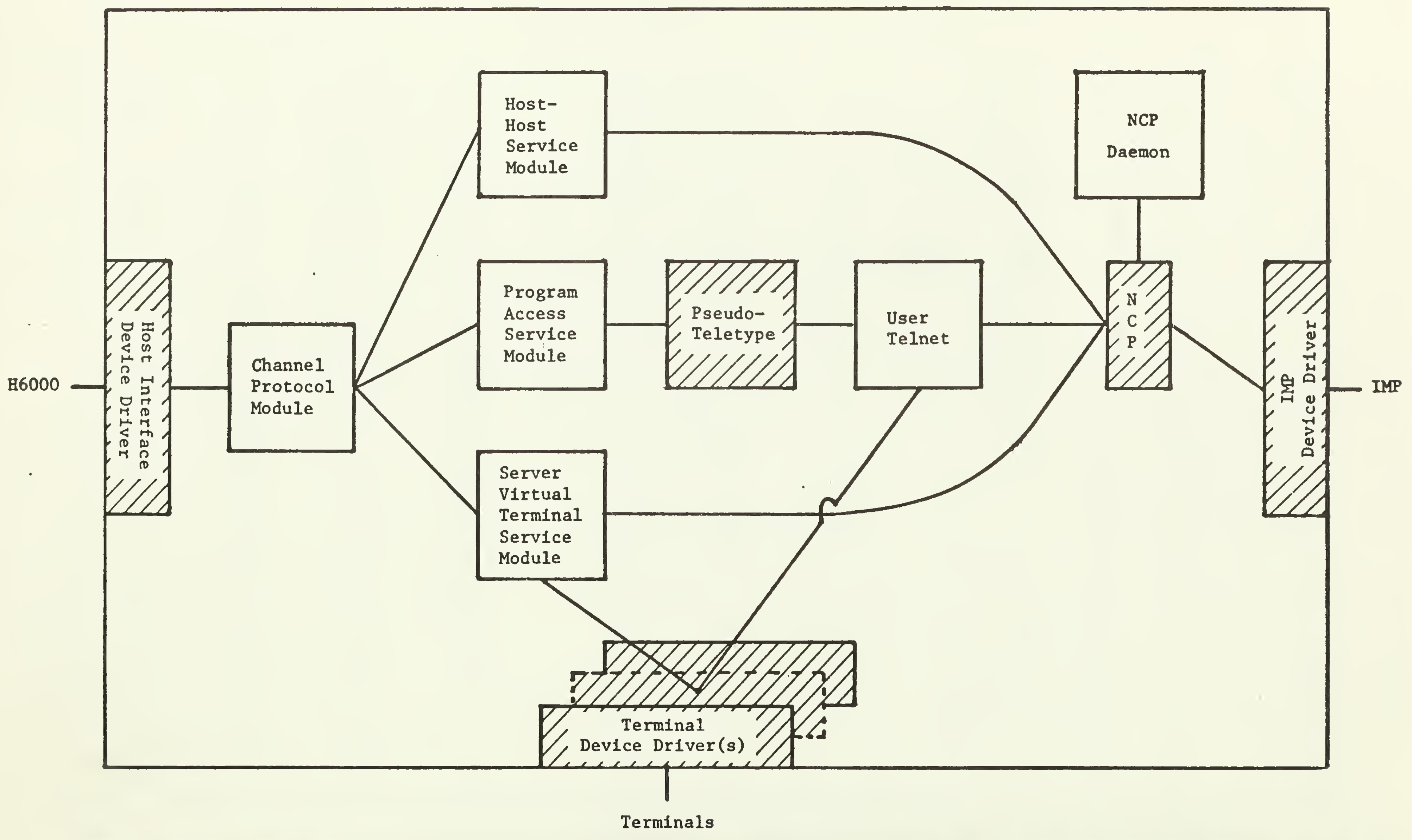
3. It maintains connection status information.

4. It performs Telnet option negotiation.

5. It enables front-end terminals to access the host.

Figure 2 on the next page illustrates the relationship between these ENFE service modules and other front-end components.

ENFE SOFTWARE ARCHITECTURE



▨ = software resident in the operating system

Figure 2

4. ADDITIONS TO UNIX

4.1 Introduction

The Unix operating system has been modified to support HFP operations. Parts of some general purpose system functions are not needed. These functions have been streamlined. Several system functions have been removed completely.

Device drivers have been added to Unix to manage the IMP-to-host interface, the H6000 to PDP-11 interface, and VIP terminals. The Unix terminal handler and numerous other system modules have been modified for the front-end effort.

Two additional facilities have been implemented to efficiently support HFP operations: an inter-process communication facility and a non-blocking I/O facility.

4.2 Inter-Process Communication Facility

An inter-process communication (IPC) facility has been added to effect efficient communication between ENFE programs and to provide a convenient mechanism for the implementation of non-blocking I/O. IPC communications consist of events and messages.

Events communicate small amounts of control information. Events have a source process, a destination process, an opcode, and a word of data. The contents of the opcode and data fields are application-dependent.

Messages transfer large amounts of data. Messages are created, transmitted, and received within segments. A segment is an area of physical core memory dynamically mapped into and out of the address spaces of communicating processes.

Each process has an IPC queue where events and message descriptors are stored until requested.

The IPC facility is described in CAC Technical Memorandum No. 84, "Illinois Inter-Process Communication Facility for Unix." An understanding of the contents of this memorandum is essential to a complete comprehension of the detailed technical descriptions which follow.

4.3 Non-Blocking I/O Facility

The non-blocking I/O facility enables a single program to perform I/O concurrently on multiple devices. As a result, the number of active programs required for the ENFE is greatly reduced. The standard Unix system does not have this capability.

Non-blocking I/O uses IPC events to notify user processes of the completion of I/O operations. The Unix I/O system has been modified to generate events at appropriate times:

1. the opening of an I/O device,
2. the closing of an I/O device,
3. the arrival of input data,
4. the completion of output operations, and
5. the occurrence of special device conditions.

When programs receive these events, they execute standard I/O system primitives in the usual manner.

CHANNEL PROTOCOL MODULE

5. CHANNEL PROTOCOL MODULE

5.1 Function

The Channel Protocol module manages logical channels and serves as a bi-directional multiplexor. The CPM is implemented within its own address space as a user level program. The CPM performs several functions.

1. It de-multiplexes HFP Messages arriving from the host interface and passes them to the appropriate service modules in the front end.

2. It accepts input in the form of HFP Messages from the service modules and multiplexes them to the host interface.

3. It controls the flow of HFP TRANSMIT Commands into the front end.

4. It performs error checking at the Channel Protocol level.

5.2 Operation

The Channel Protocol module (CPM) waits for input from the host and from the service modules. Messages from both sources have the form of HFP Commands and Responses. These HFP Commands and Responses are transported within inter-process communication messages. As the CPM receives each input, it calls a subroutine appropriate to the service and type of HFP Message. These subroutines perform multiplexing, flow control, and error checking.

5.3 Subroutine-Calling Hierarchy

The following table illustrates the principal subroutines in the CPM subroutine-calling hierarchy. The "c/r" following subroutine names indicates the existence of both a command subroutine and a response subroutine. For example, "BEGIN c/r" indicates that subroutines exist at that level to process both an HFP BEGIN Command and an HFP BEGIN Response.

```
                              MAIN
                                |
                                |
            -------------------------------------------
            |                       |                 |
            |  (host             FINDCHAN             |  (service
            |   messages)                             |   messages)
            |                                         |
     ---------------        ---------------------------------------
     |             |        |           |             |           |
 BEGIN c/r   FLOWCTRL   END c/r   TRANSMIT c/r   SIGNL c/r   EXEC c/r
                   |
                   |
       -------------------------------------
       |           |           |           |
   END c/r   TRANSMIT c/r   SIGNL c/r   EXEC c/r
```

5.4 State Transition Table

The following table depicts CPM logical channel states. The STATE column indicates the current channel state, the EVENT column indicates the occurrence of a specific event, the ACTION column indicates the action taken when an event occurs, and the NEXT STATE column indicates the state of the channel after the action is performed.

STATE	EVENT	ACTION	NEXT STATE
NULL	BEGIN Command from host	Initialize channel data structure, pass Command to service	PEND
PEND	BEGIN Response Status = Ø from service	Pass Command to host or service	ESTAB
	BEGIN Response Status not = Ø from service	Pass Command to host or service	NULL
	END Command	Pass Command to destination	TERM
ESTAB	END Command w/drain	Wait for queued TRANSMITS to drain	DRAIN
	END Command w/o drain	Cleanup channel data structure, pass command to destination	TERM
	SIGNAL Command	Take appropriate signal action	ESTAB
	All other Commands	Update flow control and acknowledge information	ESTAB

DRAIN	TRANSMITS drained	Send END Command	TERM
TERM	END Response	Release channel data structure	NULL
	BEGIN Response from service	Ignore	TERM

5.5 Decision Matrix

The following table is a matrix of the subroutines called when an HFP Message is received.

MSG TYPE	CHANNEL STATES				
	NULL	PEND	ESTAB	DRAIN	TERM
BEGIN	BEGIN	ERROR	ERROR	ERROR	ERROR
BEGINR	DISCARD	BEGINR	DISCARD	DISCARD	DISCARD
TRANSMIT	DISCARD	DISCARD	TRANSMIT	DRNFLCL	DISCARD
TRANSMITR	DISCARD	DISCARD	TSER	TSER	DISCARD
SIGNAL	DISCARD	DISCARD	SIGNAL	SIGNAL	DISCARD
SIGNALR	DISCARD	DISCARD	TSER	TSER	DISCARD
EXECUTE	DISCARD	DISCARD	EXECUTE	DRNFLCL	DISCARD
EXECUTER	DISCARD	DISCARD	TSER	TSER	DISCARD
END	END	END	END	ERROR	ERROR
ENDR	DISCARD	ENDR	DISCARD	DISCARD	ENDR

5.6 CPM Data Structures

Channel information is kept in an ordered list of data structures. The list is ordered by channel group and channel member. Each active HFP logical channel has an entry in this list. Each entry contains the following fields (the numbers in parens are the field widths in bits):

c_link	(16)	... pointer to the next channel element, NULL indicates end of list
c_group	(16)	... channel group member
c_member	(16)	... channel member number
c_state	(Ø8)	... channel state (see states below)
c_service number	(Ø8)	... number of the service for this channel
c_service IPC	(Ø8)	... IPC address of the service for this channel
c_hiscredit	(Ø8)	... current credit given by host
c_ack	(Ø8)	... sequence number of last TRANSMIT received from host
c_credit	(Ø8)	... amount of credit given by front end
c_myseq	(Ø8)	... last TRANSMIT acknowledged by host
c_seq	(Ø8)	... sequence number of last TRANSMIT sent to host
c_qhd	(16)	... queue head of Commands waiting to be sent or acknowledged
c_sendq	(16)	... Commands waiting to be sent

The following are state variable values:

NULL	Ø
PEND	1
ESTAB	2
DRAIN	3
TERM	4

6. CHANNEL PROTOCOL MODULE (CPM): PROGRAM ANALYSIS

The following subroutines comprise the Channel Protocol module.

6.1 MAIN

Abstract. MAIN procures required inter-process communication (IPC) resources, initializes free lists, initiates communications with the host, and provides the "driving loop" for the program.

Logic. The ABSI (Asynchronous Bit Serial Interface) process located in the network front end, stores data from the host CPM in Messages for the front-end CPM. This data has the form of HFP Commands and Responses. In order to facilitate identification of both the front-end CPM and the ABSI process, "generic" names must be assigned. MAIN calls a system primitive, IPCGEN, for this assignment.

A certain portion of the CPM's memory is reserved for channel data structures. For organizational purposes, MAIN calls INIT_CHAN to link these structures into a free list. When needed, the channel data structures are transferred from a free list to an active list. Each channel data structure in the active list represents the existence of a particular HFP logical channel.

HFP TRANSMIT Commands from the services may be blocked from transmission to the host CPM due to the channel protocol flow control mechanism. These TRANSMIT Commands must be buffered. As a result, each channel data structure has an associated waiting queue which stores TRANSMIT Command identifiers. The waiting queue is constructed as a circularly-linked list holding

SIGNAL Commands with the Synchronize bit set, END Commands with the Flush Away bit cleared, and TRANSMIT Commands from the service modules. The TRANSMIT Commands may or may not have been sent.

HFP Message identifiers are stored in queue elements which are linked onto the end of the waiting queue. MAIN calls TINIT to link these queue elements into a free list where they remain until needed.

MAIN executes the GETSBA system primitive. The operating system returns a segment base address or SBA. An SBA is a user-level virtual address used to access HFP Messages received from the host CPM and the service modules.

MAIN then calls COMM_INIT to establish direct CPM-to-CPM communications with the H6000.

After setting up the necessary IPC resources (generic names and SBA), MAIN falls into a loop where it calls the system primitive WALL to wait for HFP Messages from the service modules and the ABSI process. As these HFP Messages arrive, command/response subroutines are called to process specific HFP Message types.

6.2 COMM_INIT

Abstract. The COMM_INIT subroutine "brings up" the logical communications link with the H6000.

Logic. COMM_INIT calls H SEND to transmit an END Command to the ABSI process and thus to the host CPM. COMM_INIT then calls the system primitive FREESEG to free the HFP Message just transmitted. W_TYPE is called to wait for an END Command from the host CPM.

Upon receipt of an END Command, COMM_INIT again calls the system primitive FREESEG to discard the HFP Message. W_TYPE is called to wait for a BEGIN Command from the host CPM.

Upon receipt of a BEGIN Command, COMM_INIT calls H_SEND. H_SEND transmits a BEGIN Response to the host CPM. The system primitive FREESEG is then called to discard the BEGIN Response. At this point, a communications link has been established between the host CPM (H6000) and the front-end CPM.

6.3 MESSAGE

Abstract. The MESSAGE subroutine sets up a series of actions to be taken upon the receipt of an HFP Message.

Logic.

The MESSAGE subroutine manipulates system primitives to gain access to HFP Message input. If the input is not an HFP Message, MESSAGE calls the system primitive FREESEG to discard the Message. However, if an HFP Message is received, the size field in the HEADER of the Message is examined. If the message size is greater than a predefined maximum size limitation, MESSAGE calls ERROR. If the size field is within the allowable range, MESSAGE calls FINDCHAN. If FINDCHAN returns a pointer to a valid channel data structure, the state is copied out of the structure. Otherwise, the state is set to NULL.

The MESSAGE subroutine then computes the source of the Message, the type of Message, and the process identification of the destination of the Message. The MESSAGE subroutine logs the Message. The MESSAGE subroutine uses the Message type and channel state to call a subordinate subroutine which completes the processing of the HFP Message.

6.4 BEGIN

Abstract. The BEGIN subroutine processes a BEGIN Command from the host.

Logic. The BEGIN subroutine will only be called if the channel is in the NULL state. The BEGIN subroutine verifies the requested service module's validity and existence. If the service is invalid, BEGIN calls ERROR to generate a BEGIN Response. If the service is valid, BEGIN calls the system primitive IPCGEN to determine the address of the requested service module. If IPCGEN returns an error, BEGIN calls ERROR to generate a BEGIN Response indicating that the service is dead.

MAKCHAN is then called to allocate a free channel data structure to the BEGIN subroutine. Pertinent information is copied into the channel data structure. BEGIN sets the channel state to PEND (pending). The BEGIN Command is then forwarded to the requested service module.

6.5 BEGINR

Abstract. The BEGINR subroutine processes a BEGIN Response.

Logic. If the BEGIN Response indicates that the begin request failed, BEGINR calls FORWARD to forward the BEGIN Response and FREE_CHAN to free the channel data structure.

If the begin request was successful, BEGINR sets the channel state to ESTAB (established). If the BEGIN Response was sent by a service module, the CPM initializes the amount of credit given by the front end. BEGINR then calls FORWARD to forward the Message.

6.6 EXEC

Abstract. The EXEC subroutine processes an EXECUTE Command.

Logic. If an EXECUTE Command is received from the host, the EXEC subroutine updates flow control information and forwards the EXECUTE Command to the service module associated with the Command.

If an EXECUTE Command arrives from a service module, the EXEC subroutine updates the amount of credit given by the front end. EXEC then forwards the EXECUTE Command to the host CPM.

6.7 TSER

Abstract. The TSER subroutine is called when HFP TRANSMIT, SIGNAL, and EXECUTE Responses are received.

Logic. If a TRANSMIT, SIGNAL, or EXECUTE Response arrives from a service module, the TSER subroutine updates the amount of credit given by the front end and forwards the Message to the host CPM.

If the channel is in the DRAIN state or the the waiting queue is not empty, and a TRANSMIT Response is received from a service module to update flow control information, the TRANSMIT Response is discarded.

If a TRANSMIT, SIGNAL or EXECUTE Response arrives from the host indicating a channel protocol error, TSER calls LOGERR to log the error. The TSER subroutine then updates flow control information and forwards the Message to the service module.

However, if a TRANSMIT, SIGNAL, or EXECUTE Response arrives from the host indicating that the channel was not found, TSER calls CLEAN. CLEAN removes all Commands on the channel's waiting queue. FREE_CHAN is called to return the channel data structure to the free list and a system primitive FREESEG is called to discard the Message.

If a TRANSMIT Response arrives from the host with a CPM-specific error value or if the channel state is set to DRAIN, TSER calls the system primitive FREESEG to discard the Message.

If a SIGNAL Response or an EXECUTE Response or a TRANSMIT Response with a service-specific error value arrives, TSER updates flow control information and then forwards the Message to the service module.

6.8 DRNFLCL

Abstract. DRNFLCL processes TRANSMIT or EXECUTE Commands which reference a channel in the DRAIN state.

Logic. If a TRANSMIT or EXECUTE Command arrives from the host, DRNFLCL calls FLOWCTRL to update flow control information. DRNFLCL then calls the system primitive FREESEG to discard the Message.

6.9 SIGNL

Abstract. The SIGNL subroutine processes HFP SIGNAL Commands.

Logic. If a SIGNAL Command arrives from the host requesting an immediate SIGNAL Response, SIGNL calls FLOWCTRL to update flow control information and ECHO to return the Message to the host.

However, if a SIGNAL Command arrives from the host requesting Flush Toward, SIGNL calls CLEAN to remove all TRANSMIT Commands on the associated channel's waiting queue. SIGNL then calls FLOWCTRL to update flow control information and FORWARD to send the Message to the associated service module.

If a SIGNAL Command arrives from a service module, SIGNL updates the amount of credit given by the front end and forwards the Message to the host. However, if the SIGNAL Command requested Flush Away, SIGNL first calls CLEAN to remove all TRANSMIT Commands on the associated channel's waiting queue. Any previously queued SIGNAL Commands are forwarded to the host.

If a Synchronize action is requested, and the associated channel's waiting queue is not empty, SIGNL calls ENQUE. ENQUE adds the SIGNAL Command to the end of the waiting queue and SIGNL returns.

If a SIGNAL Command arrives from the host and the associated channel state is set to DRAIN, SIGNL calls the system primitive FREESEG to discard the Message. Otherwise, the Message is forwarded to the the service module.

6.10 TRANSMIT

Abstract. The TRANSMIT subroutine processes HFP TRANSMIT Commands.

Logic. If a TRANSMIT Command arrives from a service module, the TRANSMIT subroutine updates credit information and queues the TRANSMIT Command. TRANSMIT then calls HT_TEST to determine whether any credit is available from the host. If credit is available, the TRANSMIT Command is immediately forwarded to the host.

However, if the TRANSMIT Command cannot be forwarded (credit unavailable), TRANSMIT must determine whether the service module should stop sending TRANSMIT Commands over the associated logical channel. If the flow of TRANSMIT Commands has not already been stopped, and if the number of queued TRANSMIT Commands is greater than a predefined limit (HIWAT), TRANSMIT calls SERVFLOW. SERVFLOW sends an XOFF request to instruct the service module to stop the transmission of TRANSMIT Commands on the associated channel. The CPM sets a flag in the channel structure to avoid duplicating this request.

If a TRANSMIT Command arrives from the host, TRANSMIT updates flow control information. The TRANSMIT subroutine then determines whether the TRANSMIT Command is in sequence, in window (correct ack and credit fields), and is not a duplicate. If the TRANSMIT Command is out-of-sequence or out-of-window, TRANSMIT calls H_TR to send a TRANSMIT Response to the host indicating the error. However, if the TRANSMIT Command is a duplicate, TRANSMIT calls the system primitive FREESEG to discard the Message.

If the TRANSMIT Command is in sequence and in window, TRANSMIT updates flow control and acknowledgement information. If no HFP Messages are waiting for transmission to the host, TRANSMIT calls H_TR to generate a TRANSMIT Response. This TRANSMIT Response will advise the host CPM of current flow control and acknowledgement status.

TRANSMIT now calls FLOWCTRL to update front-end CPM flow control information. The TRANSMIT subroutine then calls FORWARD to send the Message to the associated service module.

6.11 ENDR

Abstract. ENDR is called when an HFP END Response is received. ENDR performs the functions necessary to terminate a logical channel.

Logic. ENDR calls CLEAN to remove all TRANSMIT Commands on the waiting queue. ENDR then forwards the END Response to the specified destination and calls FREE_CHAN. FREE_CHAN returns the channel data structure to the free list.

6.12 END

Abstract. The END subroutine processes HFP END Commands.

Logic. If an END Command refers to one specific logical channel, END calls ENDONE. ENDONE generates an HFP END Command on the specified logical channel and returns.

If an END Command does not refer to one specific logical channel, END determines whether the END Command specifies a valid group. If not, ERROR is called to generate an END Response with an error indication.

Otherwise, WILDCHAN is repeatedly called to locate those channel data structures which are members of the specified group. A separate END Command is constructed for each located channel data structure.

6.13 ENDONE

Abstract. The ENDONE subroutine processes an HFP END Command on a specific logical channel.

Logic. If the END Command is from the host, or if the END Command has the Flush Away bit set, ENDONE calls CLEAN.

CLEAN removes all TRANSMIT Commands on the channel's waiting queue.

If the channel's waiting queue is empty, the logical channel state is set to TERM (terminating). The ENDONE subroutine forwards the END Command.

However, if the channel's waiting queue is not empty, the logical channel's state is set to DRAIN. ENDONE calls ENQUE to add the END Command to the waiting queue. The END Command must be retained until all TRANSMIT Commands on the waiting queue have been sent and acknowledged.

6.14 Waiting Queue

An understanding of the ENQUE, DEQUE, MVQ, and CLEANQ subroutines described below requires a basic knowledge of the purpose and function of the waiting queue. As noted previously, the waiting queue is constructed as a circularly-linked list holding SIGNAL Commands with the Synchronize bit set, END Commands with the Flush Away bit cleared, and TRANSMIT Commands from the service modules. The TRANSMIT Commands may or may not have been sent. The waiting queue is required for several reasons.

The first concerns the amount of credit remaining with the host. The credit field contains the number of TRANSMIT Commands which the host is currently prepared to receive. If the value of the credit field is zero (no credit), a TRANSMIT Command sent by a service module must be placed on the waiting queue to await transmission to the host.

The second concerns the HFP acknowledgement mechanism.

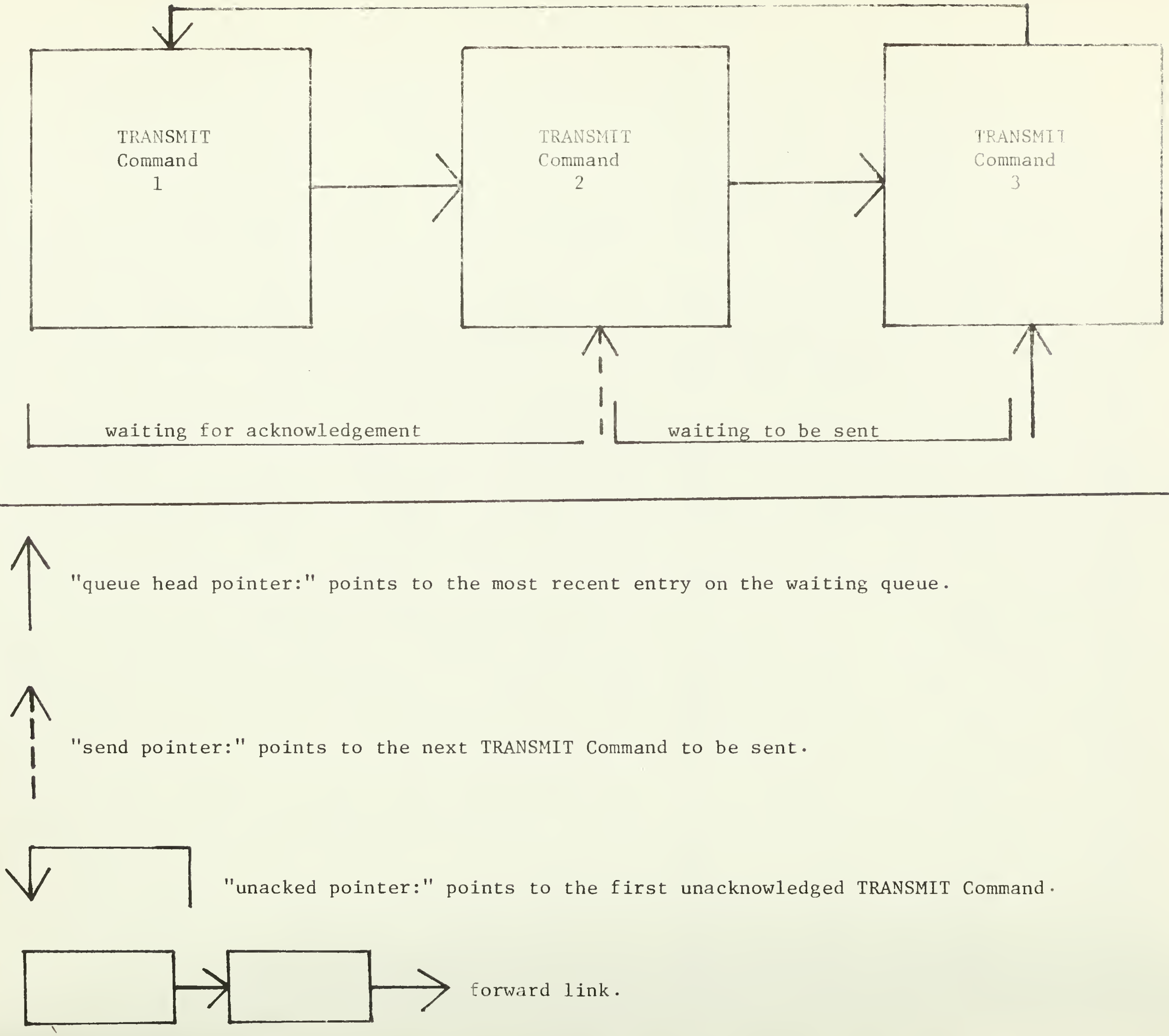
Any sent, but unacknowledged, TRANSMIT Commands must be retained on the waiting queue.

The third concerns the arrival of a SIGNAL Command with the Synchronize bit set sent by a service module. If the waiting queue is empty, the SIGNAL Command is immediately sent to the host. If, however, the waiting queue contains HFP Messages waiting to be sent or acknowledged, the SIGNAL Command must be entered at the end of the queue.

The final reason concerns the receipt of an END Command with the Flush Away bit cleared sent by a service module. If the waiting queue contains HFP Messages waiting to be sent or acknowledged, the END Command must be added to the end of the queue.

Figure 3 on page 38 illustrates the construction of the waiting queue. The three pointers establish the distinction between sent, unsent, and unacknowledged TRANSMIT Commands. The "send" pointer (pointing to the element containing TRANSMIT Command 2) distinguishes between sent and unsent (wait for credit) TRANSMIT Commands. The distance between the "unacked" pointer and the "send" pointer indicates that TRANSMIT Command 1 has been sent, but not acknowledged. The distance between the "send" pointer and the "queue head" pointer (pointing to the element containing TRANSMIT Command 3) indicates that TRANSMIT Commands 2 and 3 are awaiting transmission. The "queue head" pointer, which points to the most recent entry on the waiting queue, is incorporated solely for the purpose of maintaining the circularly-linked list.

Descriptions of the ENQUE, DEQUE, MVQ, and CLEANQ subroutines follow. Each description is correlated to the illustration of the waiting queue on page 38.



"queue head pointer:" points to the most recent entry on the waiting queue.

"send pointer:" points to the next TRANSMIT Command to be sent.

"unacked pointer:" points to the first unacknowledged TRANSMIT Command.

forward link.

Figure 3

6.15 ENQUE

Abstract. The ENQUE subroutine adds an HFP Message to a logical channel's waiting queue on a first-in, first-out basis.

Logic. If there are no available queue elements, ENQUE calls PANIC. PANIC logs a fatal error and restarts the front-end CPM.

However, if there are available queue elements, ENQUE delinks the last queue element from the free list and links it onto the end of the waiting queue. The HFP Message identifier is then copied into the queue element.

The ENQUE subroutine updates the "queue head" pointer. For purposes of illustration, it will be assumed that the element containing TRANSMIT Command 3 has just been entered on the waiting queue. The "queue head" pointer points to the most recent entry; i.e., that containing TRANSMIT Command 3. If TRANSMIT Command 2 has not yet been sent, the "send" pointer remains intact. However, if TRANSMIT Command 2 has been sent, the "send" pointer will point to the element containing TRANSMIT Command 3. In this case, the "send" pointer would be equal to the "queue head" pointer.

6.16 DEQUE

Abstract. The DEQUE subroutine removes an HFP Message from the waiting queue.

Logic. If a Message arrives from the host indicating that TRANSMIT Command 1 has been acknowledged, the DEQUE subroutine removes the oldest entry on the waiting queue. As illustrated in the diagram, the oldest entry contains TRANSMIT

Command 1.

If the "send" pointer had been pointing to the oldest entry, DEQUE moves it to the next entry; i.e.,that which contains TRANSMIT Command 2. DEQUE calls the system primitive FREESEG to discard TRANSMIT Command 1. If the element containing TRANSMIT Command 1 is the last and only element on the waiting queue, DEQUE initializes both the "send" pointer and the "queue head" pointer.

However, if there are other elements on the waiting queue, the value of the forward link between the elements containing TRANSMIT Command 1 and TRANSMIT Command 2 is copied to the "unacked" pointer. As a result, the "unacked" pointer now points to the element containing TRANSMIT Command 2. The element which contained TRANSMIT Command 1 is returned to the free list.

6.17 MVQ

Abstract. The MVQ subroutine updates the "send" pointer.

Logic. For purposes of illustration, it will be assumed that due to the availability of credit, TRANSMIT Command 2 has been sent to the host. The MVQ subroutine moves the "send" pointer to the next element awaiting transmission. According to the diagram, the next element is that which contains TRANSMIT Command 3. The "send" pointer is now equal to the "queue head" pointer. The MVQ subroutine initializes the value of the "send" pointer.

However, if the "send" pointer had previously been equal to the "queue head" pointer, the value of the "send" pointer would be set to zero.

6.18 CLEANQ

Abstract. The CLEANQ subroutine removes all HFP Commands from the waiting queue.

Logic. If the waiting queue is empty, CLEANQ returns. If, however, the waiting queue is not empty, the CLEANQ subroutine discards all entries on the queue from the oldest to the most recent.

CLEANQ first points to the oldest element on the waiting queue. CLEANQ calls the system primitive MAPSEG to obtain access to the associated HFP Message. If the Message is a TRANSMIT Command, the transmit count is decremented and CLEANQ calls DEQUE. DEQUE removes the Message from the waiting queue.

If the Message is an END Command, CLEANQ sets the channel state to TERM. CLEANQ calls H_SEND to send the END Command to the host and the END Command is dequeued.

If the Message is a SIGNAL Command, and only TRANSMIT Commands are to be discarded, CLEANQ calls H_SEND to send the SIGNAL Command to the host. The SIGNAL Command is dequeued.

6.19 FLOWCTRL

Abstract. The FLOWCTRL subroutine updates flow control information.

Logic. Upon receipt of an HFP Message from the host, the credit field in the HEADER is examined. The credit field contains the number of TRANSMIT Commands which the host is currently prepared to receive. If the value of the credit field is greater than the allowable limit of eight, an error is logged and the value is set to eight. If the value of the credit field is less than eight, FLOWCTRL copies that value into the hiscredit (current credit given by the host) field in the channel data structure.

The ack field in the HEADER of the HFP Message is then examined. The ack field contains the sequence number of the last TRANSMIT Command correctly received by the host. FLOWCTRL copies the value of the ack field into the myseq (last TRANSMIT Comamnd acknowledged by the host) field in the channel data structure. FLOWCTRL calls DEQUE to discard acknowledged TRANSMIT Commands from the waiting queue and the count of the number of queued TRANSMIT Commands is decremented.

If the status field in the HEADER indicates an out-of-window or out-of-sequence error, FLOWCTRL calls NFELOG to indicate an error. As a result, the "send" pointer must be moved back to the beginning of the waiting queue so that all unacknowledged TRANSMIT Commands will be resent. The sequence number of the oldest previously sent TRANSMIT Command is used as the sequence number for the first retransmitted HFP Command.

FLOWCTRL obtains access to TRANSMIT Commands which must be retransmitted to the host. FLOWCTRL then calls HT_TEST to determine whether the host is accepting TRANSMIT Commands (credit available). If so, H_SEND is called to send as many queued TRANSMIT Commands as credit allows.

HFP Messages containing SIGNAL or END Commands which have been entered on the waiting queue must be retained until all TRANSMIT Commands on the queue have been sent and acknowledged. Once all TRANSMIT Commands have been sent and acknowledged up to the current Command ("send" pointer equal to the oldest entry), FLOWCTRL calls H_SEND to send the first queued SIGNAL or END Command. If an END Command is found, FLOWCTRL sets the channel state to TERM, and sends the END Command. FLOWCTRL calls DEQUE to remove the END Command from the queue.

FLOWCTRL must now determine whether the service module should be allowed to send TRANSMIT Commands. If the CPM previously instructed the service module to stop sending TRANSMIT Commands on a logical channel (XOFF), and if the number of queued TRANSMIT Commands is less than a predefined minimum (LOWAT), FLOWCTRL calls SERVFLOW. SERVFLOW sends an XON request to instruct the service module to restart the transmission of TRANSMIT Commands on the channel.

6.20 HT_TEST

Abstract. The HT_TEST subroutine determines whether a TRANSMIT Command may be sent to the host.

Logic. If the sequence number of the next TRANSMIT Command to be sent is within the credit window specified by the host, that sequence number is copied into the HEADER of the next TRANSMIT Command to be sent. HT_TEST then calls MVQ to update the waiting queue "send" pointer.

6.21 FORWARD

Abstract. The FORWARD subroutine passes an HFP Message to the host or a service module.

Logic. If a Message arrives from the host, S_SEND is called to send the HFP Message to the requested service module.

If a Message arrives from a service module, H_SEND is called to send the HFP Message to the host.

6.22 H_SEND

Abstract. The H_SEND subroutine sends an HFP Message to the host.

Logic. If the Message is not a BEGIN Command, the credit, ack, and seq fields must be copied from the channel data structure into the HEADER of the Message.

If the specified logical channel state is set to either DRAIN or TERM, H_SEND sets the credit field value to eight. H_SEND then calls the system primitive SNDSEG to send an HFP Message to the host CPM. If SNDSEG returns an error, H_SEND determines whether the error type is temporary or permanent. If temporary, H_SEND waits for one second, and then calls SNDSEG again. This process is repeated until the Message is successfully transmitted. If the error is permanent, H_SEND calls PANIC to restart the CPM.

If the Message is not a TRANSMIT Command, H_SEND calls the system primitive FREESEG to discard the HFP Message.

6.23 S_SEND

Abstract. The S SEND subroutine sends an HFP Message to a service module.

Logic. S_SEND calls the system primitive SNDSEG to send the HFP Message to the requested service module. If SNDSEG returns an error, S_SEND determines whether the error is temporary or permanent. If temporary, S_SEND waits for one second, and calls SNDSEG again. This process is repeated until the Message is successfully transmitted. When SNDSEG finally sends the HFP Message to the service module, S_SEND calls the system primitive FREESEG to discard the HFP Message.

If the error is permanent, S_SEND calls LOGERR. S_SEND then calls the system primitive FREESEG to discard the HFP Message.

6.24 H_TR

Abstract. The H_TR subroutine sends a TRANSMIT Response with the specified status to the host.

Logic. H_TR manipulates IPC system primitives to construct an HFP TRANSMIT Response. H_TR then fills in the following fields in the HEADER of the Message: size, type, group, member, and status. Finally, H_TR calls H_SEND to send the TRANSMIT Response.

6.25 SERVFLOW

Abstract. The SERVFLOW subroutine sends an XON or XOFF request to a service module.

Logic. SERVFLOW manipulates IPC system primitives to construct an HFP Message. SERVFLOW then fills in the group and member fields of the HEADER and calls H_SEND to send the HFP Message to a service module.

6.26 DISCARD

Abstract. DISCARD is called when an HFP Message is received referencing a logical channel which is in an inappropriate state. DISCARD releases the Message.

Logic. An error is logged, but it is not reported to the host. The system primitive FREESEG is called to free the Message.

6.27 ERROR

Abstract. ERROR is called to send an HFP Response with an error indication.

Logic. An error is logged. The error code is copied into the status field of the Message. ECHO is called to return the Message to the sender.

A FRONT-END SERVICE

7. INTERNAL SERVICE ARCHITECTURE

7.1 General Description

Communications between the front-end CPM and a front-end service (stage 3 communications) have been structured in terms of Host-to-Front-End Protocol Messages. The inter-process communication (IPC) facility used to convey these Messages between the front-end CPM and a service is described in the section entitled "Additions to Unix" on page 17. This facility is functionally equivalent to the CPM-to-service protocol described in the HFP specification.

The following section describes the internal hierarchy and actions of a "typical" service. Descriptions of the specific services (HHS Module, PAS Module, SVTS Module) are adapted from this section. These descriptions begin in section 10 on page 77.

7.2 Service Operation

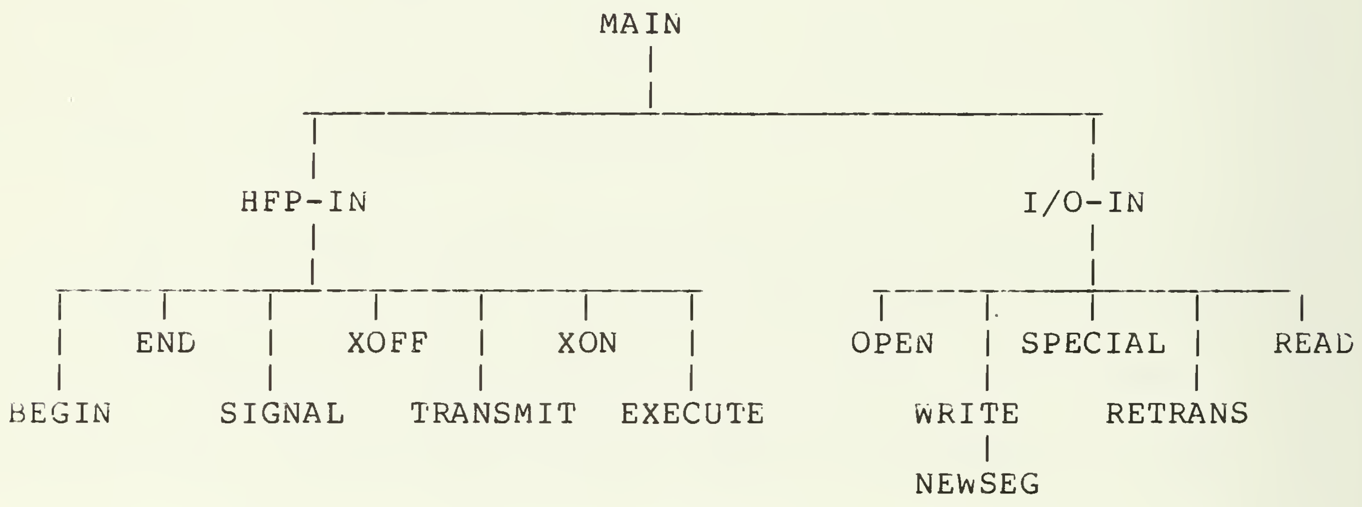
A service receives HFP Messages from the front-end CPM. As each Message is received, the service calls a subroutine appropriate to the HFP Message type: BEGIN, TRANSMIT, SIGNAL, EXECUTE, or END. These subroutines perform Command-specific functions, handle error situations, execute state transitions, and generate HFP Responses.

The Unix system generates I/O completion events to indicate the completion of device I/O operations. As each event is received, the service calls a subroutine appropriate to the event type: OPEN, READ, WRITE, RETRANS (retransmit), and SPECIAL.

These subroutines complete the processing of HFP TRANSMIT Command data being output to an I/O device, handle error situations, execute state transitions, and generate HFP Responses.

7.3 Service Subroutine Hierarchy

This form of operation dictates the internal structure of a service. Two major divisions in program logic occur to accomodate input from the front-end CPM and the Unix I/O system. The following chart illustrates this hierarchy. Each node in the hierarchy represents a subroutine.



7.4 Service State Transition Table

Each service is programmed as a finite state machine. Inputs are received from the front-end CPM and the Unix I/O system. Each input is associated with a logical channel. The input type and current channel state determine immediate action and next channel state. The following table depicts the channel states, actions, and state transitions.

STATE	EVENT(input)	ACTION(output)	NEXT STATE
NULL	BEGIN Command	Open I/O Device	PEND
PEND	I/O open success	notify host	ESTAB
	I/O open failure	notify host	NULL
	END Command	close I/O device free resources	NULL
	SIGNAL Command	error response to host	PEND
	EXECUTE Command	error response to host	PEND
ESTAB	I/O error	notify host free resources	NULL
	TRANSMIT Command	data to I/O device	ESTAB
	data from I/O device	TRANSMIT Command to host	ESTAB
	END Command	close channel free resources	NULL
	SIGNAL Command	process command	ESTAB
	EXECUTE Command	process command	ESTAB
BUSY	I/O error	END Command to host flush buffers free resources	NULL
	TRANSMIT Command	queue command	BUSY
	data from I/O device	TRANSMIT Command to host	BUSY
	I/O completion - transmit queue empty		ESTAB

	I/O completion - queue not empty	start a new transfer	BUSY
	END Command	let data drain	TERM
	SIGNAL Command	process command	BUSY
	EXECUTE Command	process command	BUSY
TERM	data drained to device	END Response to host	NULL
	SIGNAL Command	process command	TERM
	EXECUTE Command	process command	TERM
	I/O error	END Response to host	NULL

7.5 Service Data Structures

A certain amount of a service's memory is reserved for channel data structures. Each channel data structure contains all information relevant to an HFP logical channel. Certain utility subroutines for manipulating these data structures are explained in section 9 on page 71. The following table defines the fields within a channel data structure (the numbers in parens are the field widths in bits):

link	(16)	....	address of next channel list element
group	(16)	....	channel group identifier
member	(16)	....	channel member identifier
state	(Ø8)	....	channel state
flag	(Ø8)	....	channel flag bits
size	(16)	....	number of bytes waiting to be read from the I/O device
curseg	(Ø8)	....	identifier of TRANSMIT Command being output to the I/O device
fid	(Ø8)	....	I/O device identifier
nout	(16)	....	number of TRANSMIT Commands outstanding
seqhd	(16)	....	head of I/O output queue
oldaddr	(16)	....	last I/O starting address
bytslft	(16)	....	number of bytes remaining to be output from current TRANSMIT Command
bytstran	(16)	....	number of bytes last sent to the I/O device

8. SERVICE LOGIC

The following subroutines comprise the "typical" service.

8.1 MAIN

Abstract. The MAIN subroutine is the first subroutine executed. MAIN initializes variables, procures inter-process communication resources, and provides the "driving-loop" for a service. MAIN waits for HFP Messages and flow control inputs from the front-end CPM, and I/O completion events from the Unix system. When an input arrives, the source and type are used to call either HFP-IN or I/O-IN. These two subroutines provide input-specific processing. When control is returned to MAIN, it branches to the top of the "driving loop" and repeats the process.

Logic. In order to facilitate the identification of a service by other front-end programs, a service must be assigned a generic name. MAIN executes a system primitive, IPCGEN, for this assignment.

As stated, a certain portion of a service's memory is reserved for channel data structures. For organizational purposes, MAIN links these structures into a free list. When needed, the channel data structures are transferred from a free list to an active list.

A service must retain data transmitted to an I/O device until acknowledgement of its transfer is received. Other TRANSMIT Commands which arrive prior to this acknowledgement must be queued. The storage elements for these queues are linked into a free list.

MAIN executes the system primitive GETSBA to obtain a segment base address or SBA. An SBA is a user-level virtual address used to access HFP Messages.

After obtaining the necessary IPC resources (generic name and SBA), MAIN falls into a loop where it calls the system primitive WALL to wait for input. If an I/O completion event arrives from the Unix system, MAIN calls I/O-IN. If an HFP Message or flow control input arrives from the front-end CPM, MAIN calls HFP-IN.

8.2 HFP-IN

Abstract. HFP-IN processes HFP Commands and flow control inputs from the front-end CPM. HFP-IN manipulates IPC system primitives to access an input message. If the input is of an "unknown" type, an error is logged and the input is discarded. If the input is of a "known" type, the associated logical channel data structure is found. HFP-IN then calls a subroutine specific to the input type.

Logic. HFP-IN issues the system primitive MAPSEG requesting access to the input transmitted by the front-end CPM.

If the input is of an "unknown" type, HFP-IN calls the system primitive FREESEG to deallocate the input and exits.

If the input is of a "known" type, HFP-IN calls FINDCHAN to locate the HFP logical channel data structure referenced by the input. FINDCHAN searches the active channel list and returns the address in memory where the logical channel data structure may be found. HFP-IN calls a subordinate subroutine appropriate to the HFP Message or flow control input type. The BEGIN, END, TRANSMIT, EXECUTE, SIGNAL, XOFF, and XON subroutines are described below.

8.3 BEGIN

Abstract. BEGIN is called when an HFP BEGIN Command is received. BEGIN obtains a logical channel data structure and fills in appropriate information. BEGIN then "opens" the I/O device described in the TEXT field of the BEGIN Command. Later, the Unix system will respond with an I/O completion event indicating the success or failure of the open request. The OPEN subroutine will be called upon receipt of this event. OPEN will complete the processing of the BEGIN Command.

Logic. A BEGIN Command may only reference a logical channel which is in the NULL state (non-existent). If the logical channel is not in the NULL state, BEGIN calls RESPOND to generate an HFP error Response. If the logical channel is in the NULL state, BEGIN calls MAKCHAN. MAKCHAN returns a partially initialized channel data structure. BEGIN sets the channel state to PEND or pending.

The information from the BEGIN Command TEXT field is used to initiate an I/O system open request by calling NBOPEN (system subroutine: non-blocking open).

NBOPEN returns an I/O system file descriptor. The file descriptor is used in later READ, WRITE, and CLOSE I/O system calls.

If the open attempt fails at this point, BEGIN calls RESPOND. RESPOND generates an HFP BEGIN Response to report the failure of the open attempt. Otherwise, the channel remains idle until an open I/O completion event arrives. The I/O completion event will indicate the success or failure of the open request (see OPEN on page 64).

8.4 END

Abstract. The END subroutine is called when an HFP END Command is received. If there is no data queued for output to the I/O device, the logical channel is destroyed and an HFP END Response is generated. Otherwise, the channel state is set to TERM or terminating and queued data is allowed to "drain" to the I/O device. The WRITE subroutine manages the processing of queued output data. When all queued data is processed, WRITE will complete the processing of the END Command.

Logic. An HFP END Command may only reference an existing logical channel (a channel not in the NULL state). If the logical channel referenced by the END Command is in the NULL state, an HFP error Response is generated and END exits.

If there is no queued data awaiting output, END calls

KILLCHAN. KILLCHAN destroys an existing channel, regardless of its state, and returns channel resources to their various pools.

If there is queued data awaiting output (channel state is BUSY), and the Flush Away bit is cleared, the END Command will be queued on the end of the I/O output queue. WRITE will complete the processing of the END Command after all previously queued data has been output to the I/O device. The channel state is set to TERM and data is allowed to drain.

If, however, the FLUSH Away bit is set in the control field of the HFP END Command, data which is queued for output is immediately discarded and KILLCHAN is called.

END now calls RESPOND. RESPOND generates an HFP END Response to notify the host process that the channel has been closed as requested.

8.5 TRANSMIT

Abstract. The TRANSMIT subroutine is called when an HFP TRANSMIT Command is received. Data is transferred to an I/O device via a series of I/O write operations. I/O completion events are returned after a write operation. Because of this structure, a TRANSMIT Command may require several I/O write operations before all data is transferred to an I/O device. The arrival of other TRANSMIT Commands during this process necessitates that they be queued in a first-in, first-out I/O output queue and processed later.

If data is not being output to an I/O device when a TRANSMIT Command arrives, a write I/O operation is initiated. Later, the Unix system will generate a write I/O completion event. The WRITE subroutine will complete TRANSMIT Command processing.

<u>Logic</u>. The TRANSMIT subroutine attempts to output the contents of an HFP TRANSMIT Command TEXT field to the I/O system as follows.

The two legal states for output are: BUSY and ESTAB (established). A channel is considered BUSY when a previous non-blocking I/O request could not accept all data, or the service is anticipating confirmation of a previous I/O write request via a write completion event. If the logical channel is BUSY, TRANSMIT calls ENQSEG. ENQSEG queues the TRANSMIT Command on the channel's I/O output queue. Later, the TRANSMIT Command will be processed by WRITE.

If the channel is in state ESTAB (no output in progress), the data is immediately transmitted. TRANSMIT issues a write I/O operation. The channel state is then set to BUSY.

Pertinent information is copied into the channel data structure. The IPC segment identifier (used to access the HFP TRANSMIT Command) is copied into 'curseg,' the address of the beginning of the TRANSMIT Command TEXT field is copied into 'oldaddr,' the total number of bytes of data remaining to be output is copied into 'bytslft,' and the number of bytes of data just output is copied into 'bytstran.'

If the I/O write call does not return an immediate error, the service anticipates a future I/O completion event indicating the success or failure of the write request, and the amount of additional data which may be sent (see WRITE on page 65, and NEWSEG on page 67).

If the I/O write call immediately returns an error, RESPOND is called to generate an HFP TRANSMIT Response indicating the type of error.

8.6 EXECUTE

Abstract. The EXECUTE subroutine is called when an HFP EXECUTE Command is received. The EXECUTE subroutine performs a service-specific function and returns an HFP EXECUTE. Service-specific functions include returning I/O device status information and initiation of "special" I/O operations.

Logic. The EXECUTE functions are service-specific. Detailed logic descriptions are delayed until each service is described (See descriptions beginning in section 10 on page 77).

8.7 SIGNAL

Abstract. The SIGNAL subroutine is called when an HFP SIGNAL Command is received. An HFP SIGNAL Command may request the "flushing" of logical channel data, the synchronization of the channel (return a SIGNAL Response when all queued data has been output), and the causing of an "interrupt" operation on the

I/O device.

Logic. The SIGNAL subroutine confirms the existence of the specified logical channel, parses the HFP control field, and executes the actions specified by four SIGNAL Command control bits.

The four control bits define data flushing, synchronize, and interrupt functions. Any combination of these control bits may be set within a single HFP SIGNAL Command. The precise meanings of these bits are service dependent. A generalized description of each is given below.

Synchronize. If a SIGNAL Command with the Synchronize bit set arrives and the channel state is BUSY or TERM, the SIGNAL subroutine calls ENQSEG. ENQSEG places the HFP SIGNAL Command at the end of the I/O output queue. The SIGNAL Command will be executed by NEWSEG when all TRANSMIT and SIGNAL Commands previously entered on the queue have been processed. Note that the processing of a SIGNAL Command with the Synchronize and Interrupt bits set is delayed until any queued TRANSMIT Commands have been output to the I/O device (see NEWSEG on page 67).

If a SIGNAL Command with the Synchronize bit cleared arrives or the channel is not BUSY, SIGNAL calls RESPOND to generate a SIGNAL Response.

Interrupt. If the SIGNAL Command has the Interrupt bit set, and the Synchronize bit is cleared, the SIGNAL subroutine issues an I/O device-dependent interrupt request.

Flush Toward. The SIGNAL subroutine ignores the Flush Toward bit. HFP TRANSMIT Commands in transit to the host are queued by the front-end CPM and flushed there.

Flush Away. If the Flush Away bit is set, the SIGNAL subroutine empties the channel's I/O output queue, one HFP Message at a time. The SIGNAL subroutine acts upon each queued HFP Command as described below.

The I/O output queue may contain any or all of the following HFP Command types: SIGNAL, END, and/or TRANSMIT.

1. TRANSMIT: All queued HFP TRANSMIT Commands are discarded.

2. SIGNAL: If the Interrupt bit is set, an I/O interrupt request is issued. RESPOND is called to generate a SIGNAL Response for the queued HFP SIGNAL Command.

3. END: KILLCHAN is called to destroy the logical channel data structure and generate an HFP END Response.

A SIGNAL Response is generated for the flushing SIGNAL Command.

8.8 XOFF

Abstract. The XOFF subroutine is called when an Xoff input is received. A flag in the channel data structure is used to inhibit the reception of data from an I/O device (see the READ subroutine on page 68). This flag is set by the XOFF subroutine.

Logic. The XOFF subroutine verifies the existence of the logical channel data structure. If the channel exists, the Xoff bit is set in the 'flags' field. FREESEG is then called to discard the Xoff input.

8.9 XON

Abstract. The XON subroutine is called when an Xon input is received. XON resets the flag blocking the receipt of data from an I/O device. If I/O device input data is available, XON calls the READ subroutine.

Logic. The XON subroutine verifies the existence of the logical channel data structure. If the channel exists, the Xoff flag is reset. If I/O device data has become available during the period when input was inhibited, XON calls READ to process this data.

8.1Ø I/O-IN

Abstract. The I/O-IN subroutine is called by MAIN when a Unix I/O completion event is received. I/O-IN searches for the associated logical channel data structure. If the channel data structure is not found, an error is logged and the I/O completion event is discarded. If the channel data structure is found, a subordinate subroutine is called to process the specific I/O completion event.

Logic. I/O-IN calls FINDFID to locate the HFP logical channel data structure in the active channel list. FINDFID returns a pointer to an address in memory where the channel data structure may be found. If, however, the HFP logical channel data structure cannot be found, an error is logged and the I/O completion event is discarded.

If the channel data structure is found, I/O-IN calls a subordinate subroutine appropriate to the I/O completion event type. These subroutines generate the necessary HFP Message sequences. The OPEN, WRITE, READ, RETRANS, and SPECIAL subroutines are described in detail below.

8.11 OPEN

Abstract. The OPEN subroutine is called when an open I/O completion event is received. Open I/O completion events are generated in response to file system open requests. The BEGIN subroutine issues such a request when an HFP BEGIN Command is received. OPEN completes the processing of a BEGIN Command. A

BEGIN Response is generated indicating the success or failure of the open request. If the request failed, the logical channel is destroyed.

Logic. If the I/O completion event indicates a successful open, OPEN sets the channel state to ESTAB and channel flags are initialized. ESTAB indicates that the channel is open and no output I/O is in progress.

OPEN now calls RESPOND to generate a BEGIN Response. The status field in the HEADER of the BEGIN Response is set to indicate the success or failure of the open attempt. The TEXT field of the BEGIN Response contains service-specific information.

If the I/O completion event indicates that input data is available from the I/O device, OPEN calls READ to process that data.

If, however, the open request was unsuccessful, OPEN calls KILLCHAN. KILLCHAN destroys the associated logical channel. OPEN then calls RESPOND to generate a BEGIN Response indicating the failure of the open attempt.

8.12 WRITE

Abstract. The WRITE subroutine is called in response to a write I/O completion event. WRITE manages a first-in, first-out queue of HFP TRANSMIT, SIGNAL, and END Commands. Write I/O operations are initiated either by the TRANSMIT, RETRANS subroutines, or by a previous WRITE invocation. WRITE repetitively processes each queued TRANSMIT Command until all data has been output to the associated I/O device. When processing of a TRANSMIT Command is complete, it is dequeued and a TRANSMIT Response is generated indicating the success of the TRANSMIT Command. If an unrecoverable I/O occurred during the output operation, a TRANSMIT Response with an error indication is generated.

Certain conditions will cause HFP SIGNAL and END Commands to be queued in the I/O device output queue (see SIGNAL on page 60, and END on page 57). If a SIGNAL Command is encountered, the appropriate functions are executed and a SIGNAL Response is generated. If an END Command is encountered, the channel is destroyed and an End Response is generated.

Logic. The Unix I/O system generates an I/O completion event indicating the disposition of a previous I/O write request. Receipt of such an event prompts the execution of the WRITE subroutine.

The channel data structure variables, 'oldaddr' and 'bytslft,' control the output of data from a TRANSMIT Command. The 'oldaddr' variable contains the last memory address from which data was output. The 'bytslft' variable contains the number of bytes remaining to be output. 'Bytslft' will be set to zero when all data from the associated TRANSMIT Command has been output. WRITE updates 'oldaddr' and 'bytslft.' If 'bytslft' is zero, NEWSEG is called to obtain another TRANSMIT Command from the I/O output queue. If NEWSEG returns an empty indication, the channel state is set to ESTAB, and WRITE exits. If NEWSEG returns an HFP END Command, WRITE calls KILLCHAN to terminate the logical channel, generate an END Response, and exit.

If, however, 'bytslft' is not equal to zero, another I/O write request is issued. Later, an I/O completion event will be generated by the Unix I/O system and WRITE will continue processing the HFP TRANSMIT Command.

8.13 NEWSEG

Abstract. The NEWSEG subroutine is called by WRITE when all data from an HFP TRANSMIT Command has been processed. NEWSEG generates a TRANSMIT Response indicating the successful output of TRANSMIT Command data. NEWSEG then loops, obtaining the first entry in the channel I/O output queue. If the queue is empty, NEWSEG exits. If the first command is a SIGNAL Command, the Command is processed and the next entry is obtained from the I/O output queue. If the first command is an END Command, it is returned to WRITE. If the first command is a TRANSMIT Command, some initial processing is performed and the Command is returned to WRITE for output to the I/O device.

Logic. NEWSEG calls RESPOND to generate a TRANSMIT Response. The TRANSMIT Response references the TRANSMIT Command whose processing has been completed. NEWSEG then calls DEQSEG to obtain the next queued HFP Command. IF DEQSEQ returns, the I/O output queue is empty and NEWSEG exits.

If the HFP Command returned by DEQSEG is a SIGNAL

Command, the Interrupt bit in the control field is tested. If the Interrupt bit is set, an I/O interrupt request is issued and a SIGNAL Response is generated. NEWSEG again calls DEQSEG. If the HFP Command returned is not a SIGNAL Command, it returns the HFP Command to the WRITE subroutine.

8.14 READ

Abstract. The READ subroutine is called when a read I/O completion event is received. Read I/O completion events are automatically generated by the Unix system when data is available. READ first checks the XON-XOFF flag. If this flag is on, READ indicates that data is available and exits. If this flag is off, READ formats an HFP TRANSMIT Command and obtains data from the I/O device. Any translation of input data takes place at this point. The TRANSMIT Command is then forwarded to the front-end CPM for transmission to the host.

Logic. A read I/O completion event is automatically generated indicating that a certain amount of I/O device input data is available. READ is called by either I/O-IN or OPEN. READ obtains input data from the I/O system. No further I/O completion events will be generated until an I/O system read is issued.

READ first calls BLDMSG to construct a "skeleton" HFP TRANSMIT Command. READ then issues an I/O read request to obtain I/O device data. If the I/O read request immediately fails, READ calls KILLCHAN to terminate the logical channel and generate an

HFP END Command. If the I/O read request is successful, READ calls SNDMSG to pass the HFP TRANSMIT Command to the front-end CPM. The front-end CPM will pass the TRANSMIT Command on to the host.

If the front-end CPM has stopped the transmission of data to itself with an Xoff input, the 'size' field in the channel data structure is loaded with the number of available input bytes, and a flag is set indicating the availablity of input data.

8.15 RETRANS

Abstract. The RETRANS subroutine is called in response to a retransmit I/O completion event. Certain situations occur which require the retransmission of previously output data. RETRANS performs this operation.

Logic. RETRANS is called to retransmit data previously output. RETRANS simply issues an I/O write request with the parameters previously saved in 'oldaddr' and 'bytstran.' If the I/O write request immediately returns an error, RETRANS calls KILLCHAN to destroy the logical channel and generate an HFP END Command.

8.16 SPECIAL

Abstract. The SPECIAL subroutine is called in response to certain exception I/O completion events. SPECIAL generates an HFP SIGNAL Command. This particular class of SIGNAL Command is service-specific.

Logic. Logic descriptions are delayed for service-specific presentations in later sections (descriptions begin in section 1Ø on page 77).

9. SERVICE UTILITIES

9.1 General Description

A group of subroutines perform service utility functions. These utility functions include manipulation of logical channel data structures (MAKCHAN, KILLCHAN, FINDCHAN, FINDFID), and manipulation of HFP Messages (BLDMSG, SNDMSG, RESPOND, FLUSHSEGS, ENQSEG, DEQSEG). These subroutines are detailed below.

9.2 MAKCHAN

Abstract. MAKCHAN obtains a channel data structure from the free list, initializes certain fields, and links the structure into an active list.

Logic. If the free list is empty, MAKCHAN returns a zero value. Otherwise, MAKCHAN delinks the first entry from the free list and links it into the active list. Channel Group and Member values are passed as parameters. These values are stored in the 'group' and 'membr' fields. The address of the initialized channel data structure is returned.

9.3 KILLCHAN

Abstract. KILLCHAN reinitializes an active logical channel data structure. As a result, the associated HFP logical channel is destroyed.

Logic. KILLCHAN calls FLUSHSEGS to free all HFP Messages queued on the I/O output queue. If the channel state is BUSY or TERM, KILLCHAN calls a system primitive FREESEG to release the

Message being output. The generation of an HFP END Command may be requested via a parameter. If the generation of an END Command was requested, KILLCHAN calls BLDMSG to construct a "skeleton" END Command. KILLCHAN then fills in certain HEADER and TEXT fields and calls SNDMSG to transmit the HFP Message to the front-end CPM.

The I/O device associated with the logical channel is closed, and the logical channel data structure is delinked from the channel data structure active list and linked in to the channel data structure free list.

9.4 FINDCHAN

Abstract. FINDCHAN is called to locate a logical channel data structure in the active channel list. HFP Group and Member fields are used as parameters to the search.

Logic. FINDCHAN first checks to see if there are any entries in the active channel list. If there are none, FINDCHAN returns zero. If there are entries, FINDCHAN initializes loop variables and checks each active channel data structure for a match between the 'group' and 'membr' fields and the subroutine 'agrp' and 'amemb' parameters. If a match is found, FINDCHAN returns the address of the logical channel data structure. If the active channel list is completely searched and a match is not found, FINDCHAN returns zero.

9.5 FINDFID

Abstract. Like FINDCHAN, FINDFID is called to locate a logical channel data structure. However, the channel I/O device file identifier is used as a parameter to the search.

Logic. The logic for FINDFID is the same as that for FINDCHAN except that the channel 'fid' field is compared to the 'afid' parameter. The return values are the same.

9.6 BLDMSG

Abstract. BLDMSG is called to construct a "skeleton" HFP Message. BLDMSG will obtain memory of the correct size, fill the 'group' and 'membr' parameters into the HFP HEADER and return an identifer which may be used to access the "skeleton" Message.

Logic. BLDMSG first determines the required memory size using manifest constants and a 'bsize' parameter. It then loops, requesting an area of memory of a calculated size. Once memory is obtained, the HFP Message HEADER size, type, group, and member fields are filled in. BLDMSG then returns an identifer which may be used to access the "skeleton" HFP Message.

9.7 SNDMSG

Abstract. SNDMSG is called to send an HFP Message to the front-end CPM. The parameters to SNDMSG are a logical channel data structure and an HFP Message identifier.

Logic. SNDMSG calls a system primitive SNDSEG to transmit the Message to the front-end CPM. If the SNDSEG call fails (returns negative one), an error message is logged, the front-end CPM's existence is verified, and another attempt is made to send the HFP Message. This process repeats until the Message is successfully transmitted. SNDMSG then calls a system primitive FREESEG to release the service's association with the HFP Message.

9.8 RESPOND

Abstract. RESPOND is called to send an HFP Response to the front-end CPM. RESPOND is called with an HFP Message identifier and a status.

Logic. Access to the HFP Message is obtained by calling a system primitive MAPSEG. RESPOND sets the response flag in the HFP HEADER 'type' field. This action turns an HFP Command into an HFP Response. The HFP HEADER 'status' field is filled in with the 'status' parameter and the HFP Response is sent to the front-end CPM via a call to SNDMSG.

9.9 FLUSHSEGS

Abstract. FLUSHSEGS is called to free all HFP Commands queued on a logical channel I/O output queue. FLUSHSEGS is called with a pointer to a logical channel data structure.

Logic. If the logical channel I/O output queue is empty, FLUSHSEGS simply returns. Otherwise, loop variables are initialized and each HFP Message in the I/O output queue is delinked from the queue and freed via a call to the system primitive FREESEG. This sequence continues until the queue is empty.

9.10 ENQSEG

Abstract. ENQSEG is called to add an HFP Message to the end of a logical channel I/O output queue. ENQSEG is called with a pointer to an HFP logical channel queue and an HFP Message identifier.

Logic. Each HFP Message is queued by storing its identifier in a queue element. These queue elements are linked into a logical channel I/O output queue. If no queue elements are available, ENQSEG returns negative one. Otherwise, the HFP Message identifier is stored in the queue element and the queue element is linked onto the end of the channel I/O output queue.

9.11 DEQSEG

Abstract. DEQSEG removes the first HFP Message from the front of a logical channel's I/O output queue. DEQSEG is called with a pointer to a logical channel's I/O output queue.

Logic. If the I/O output queue is empty, DEQSEG returns negative one. Otherwise, the first queue element is delinked from the front of the I/O output queue and the associated HFP Message identifier is returned. The queue element is then returned to a free pool.

ARPANET HOST-HOST SERVICE MODULE

10. ARPANET HOST-HOST SERVICE MODULE ADAPTATION

10.1 Function

The ARPANET Host-Host Service (HHS) module enables programs running in the host to manipulate the ARPANET NCP in the front end. It implements the ARPANET Host-Host process-to-service protocol definded in CAC Technical Memorandum No. 80. The HHS module performs several functions, using the ARPANET NCP in the front end.

1. It opens and closes ARPANET connections to foreign hosts on the network.

2. It passes data between the H6000 and foreign hosts on the network.

3. It maintains connection status information.

10.2 Adaptation

The ARPANET HHS module conforms to the operation of a "typical" service as described in section 7 beginning on page 48. This section adapts the HHS module's structure to the "typical" service architecture.

10.2.1 Subroutine Naming Conventions.

Various HHS module subroutines perform the same functions as those of the "typical" service. However, the names used for these subroutines are different. These differences will be resolved by placing the subroutine name from which the HHS module's subroutine name is derived, in parens, immediately following any usage of an alternate HHS module name.

10.2.2 HHS module I/O Device. The Unix ARPANET software is implemented as a Unix I/O device. ARPANET open, close, read, and write I/O system calls corespond to "typical" service I/O system operations. In this context, the ARPANET software is the HHS module's I/O device.

10.3 Operation

The basic operation of the ARPANET HHS module is the same as that of the "typical" service described in section 7.2 on page 48.

10.4 Service Subroutine Hierarchy

The subroutine-calling hierarchy of the HHS module is the same as that of the "typical" service described in section 7.3 on page 50.

```
                              MAIN
                                |
                                |
            -----------------------------------------
            |                                      |
            |                                      |
          HFEIN                                  NETIN
            |                                      |
            |                                      |
  --------------------------            ----------------------
  |   |   |   |   |   |   |             |    |    |    |    |
  | XMIT  |  EXEC |  XON  |           CHVRFY | SIG2HFE | READNET
  |       |       |       |                  |         |
BEGIN    SIG     END     XOFF              WRTNET    NETNAK
                                             |
                                           NEWSEG
```

10.5 Service State Transition Table

The HHS module's logical states and state transitions are identical to those of the "typical" service described in section 7.4 on pages 51 and 52.

10.6 Service Data Structures

10.6.1 Channel Data Structure. The channel data structure is the same as that of the "typical" service described in section 7.5 on page 53.

10.6.2 Network Software Open Data Structure. When the HHS module attempts to establish a network connection for a given logical channel, it must pass parameters with the request. These parameters are obtained from the TEXT field of the HFP BEGIN Command requesting the initiation of the network connection.

The Open Data Structure has the following fields (the numbers in parens indicate the field width in bits):

op	(08)	....	used internally by the network software
type	(08)	....	connection type
id	(16)	....	used internally by the network software
lskt	(16)	....	host's local socket for this connection
fskt	(32)	....	host's foreign socket for this connection
frnhost	(08)	....	foreign host to which the connection is to be initiated
bsize	(08)	....	number of bits per network logical byte
nomall	(16)	....	nominal allocation to be maintained
timeo	(16)	....	number of seconds to wait before timing out a network connection attempt
relid	(16)	....	internal to the network software

11. HHS MODULE: PROGRAM ANALYSIS

The following subroutines comprise the Host-Host Service module.

11.1 MAIN

Abstract. MAIN obtains IPC resources, initializes several data structure free lists, and provides the "driving loop" for the program.

Logic. The operation and logic of the MAIN subroutine is the same as the MAIN subroutine for the "typical" service described in section 8.1 on page 54.

11.2 HFEIN (HFP-IN)

Abstract. HFEIN (HFP-IN) handles HFP and flow control commands from the front-end CPM. There are seven subroutines immediately subordinate to HFEIN (HFP-IN) that handle HFP Command-specific processing: BEGIN, END, XMIT (TRANSMIT), EXEC (EXECUTE), SIGNAL, XOFF, and XON.

Logic. The operation and logic of the HFEIN (HFP-IN) subroutine is the same as that of the HFP-IN subroutine described in section 8.2 on page 55.

11.3 BEGIN

Abstract. The BEGIN subroutine parses the BEGIN Command TEXT field and initiates the requested network connection.

Logic. BEGIN calls NBOPEN to initiate an I/O network open request. Connection information (connection type, foreign host, foreign socket, nominal allocation, timeout, and byte size) is copied from the TEXT field of the BEGIN Command into a network software open structure. BEGIN uses this now initialized structure to open a network connection to the specified foreign host. The system primitive NBOPEN is executed for this purpose. The rest of the operation and logic is the same as that of the BEGIN subroutine described in section 8.3 on page 56.

11.4 END

Abstract. The END subroutine terminates a logical channel by destroying the associated logical channel data structure. Data queued for output to the network I/O device may be discarded.

Logic. The operation and logic of the END subroutine is the same as that of the END subroutine described in section 8.4 on page 57.

11.5 XMIT (TRANSMIT)

Abstract. The XMIT (TRANSMIT) subroutine initiates data transfers to the network I/O device and enqueues incoming TRANSMIT Commands.

Logic. The operation and logic of the XMIT (TRANSMIT) subroutine is the same as that of the TRANSMIT subroutine described in section 8.5 on page 58.

11.6 EXEC (EXECUTE)

Abstract. When an HFP EXECUTE Command is received, one of three functions are requested:

1. transmission of a Host-Host protocol Interrupt by Receiver (INR) over the associated network connection,

2. return of network connection status information, or

3. alteration of the suggested data allocation on the associated network connection.

Logic. If the EXECUTE Command specifies the transmission of a Host-Host INR protocol message, the EXEC (EXECUTE) subroutine calls the system primitive SENDINR. RESPOND is called to generate an HFP Response.

If the EXECUTE Command requests an alteration in the suggested allocation on the network data connection, the EXEC (EXECUTE) subroutine simply calls RESPOND to generate an HFP Response. The alteration of network allocation is not allowed in the Unix Network software.

If the EXECUTE Command requests status information, the EXEC (EXECUTE) subroutine calls the system primitive FSTAT. FSTAT returns the network connection's status (local socket, foreign socket, foreign host) and places this information and the channel state into the EXECUTE Response TEXT field. EXEC (EXECUTE) then calls RESPOND to transmit this Message to the host.

All other EXECUTE Command requests cause the generation of an HFP Response with an "unimplemented" status via the RESPOND subroutine.

11.7 SIGNAL

Abstract. The SIGNAL subroutine processes the HFP SIGNAL Command by performing data flushing, channel synchronization, and interrupt functions.

Logic. The data flushing and channel synchronization functions are implemented as described in section 8.7 on page 60. The interrupt function is implemented by calling the system primitive SNDINS. This transmits a Host-Host protocol Interrupt by Sender (INS) protocol message over the associated network connection. The action of calling SNDINS will be delayed if the synchronize flag is set and data is in the I/O output queue. The WRTNET (WRITE) subroutine will complete processing of the SIGNAL Command.

11.8 XOFF

Abstract. The XOFF subroutine inhibits the receipt of network data.

Logic. The operation and logic of the XOFF subroutine is the same as that of the XOFF subroutine described in section 8.8 on page 62.

11.9 XON

Abstract The XON subroutine restarts the flow of data from network software to the front-end CPM.

Logic. The operation and logic of the XON subroutine is the same as that of the XON subroutine described in section 8.9 on page 63.

11.10 NETIN (I/O-IN)

Abstract. NETIN (I/O-IN) receives I/O completion events from the network I/O device and dispatches control to an I/O completion event type-specific subroutine. The five subroutines called by NETIN (I/O-IN) include: CHVRFY (OPEN), READNET (READ), WRTNET (WRITE), NETNAK (RETRANS), and SIG2HFE (SPECIAL).

Logic. The operation and logic of the NETIN (I/O-IN) subroutine is the same as that of the IO-IN subroutine described in section 8.10 on page 64.

11.11 CHVRFY (OPEN)

Abstract. CHVRFY (OPEN) completes the processing of an HFP BEGIN Command. CHVRFY (OPEN) verifies the successful completion of the network connection and generates a BEGIN Response.

Logic. The operation and logic of the CHVRFY (OPEN) subroutine is the same as that of the OPEN subroutine described in section 8.11 on page 64 with the following addition. The TEXT field of the HFP BEGIN Response is filled in with the connection state, foreign host, foreign socket, number of messages allocated, number of bits allocated, local socket, and byte size.

11.12 WRTNET (WRITE)

Abstract. The WRTNET (WRITE) subroutine processes network write I/O completion events. These completion events are generated by the network I/O device to acknowledge the successful transmission of previous data.

Logic. The operation and logic of the WRTNET (WRITE) subroutine is the same as that of the WRITE subroutine described in section 8.12 on page 65.

11.13 NEWSEG

Abstract. The NEWSEG subroutine is called when all data for an HFP TRANSMIT Command has been processed. NEWSEG searches the I/O output queue for another HFP Command. If the HFP Command is a SIGNAL Command, it is processed. Otherwise, it is returned to the caller.

Logic. The operation and logic of the NEWSEG subroutine is described in section 8.13 on page 67.

11.14 READNET (READ)

Abstract. The READNET (READ) subroutine is called when a network read I/O completion event is received. READNET (READ) obtains network data, formats it into an HFP TRANSMIT Command and forwards it to the front-end CPM.

Logic. The operation and logic of the READNET (READ) subroutine is the same as that of the READ subroutine described in section 8.14 on page 68.

11.5 NETNAK (RETRANS)

Abstract. When the transmission of data to a foreign host fails, the network software generates a negative acknowledgement I/O completion event. NETNAK (RETRANS) is called when a negative acknowledgement I/O completion event is received.

Logic. The operation and logic of the NETNAK (RETRANS) subroutine is the same as that of the RETRANS subroutine described in section 8.15 on page 69.

11.16 SIG2HFE (SPECIAL)

Abstract. When the network software receives an ARPANET Host-Host Interrupt by Sender (INS) protocol message, it generates a special I/O completion event. SIG2HFE (SPECIAL) is called when a special I/O completion event is received. SIG2HFE (SPECIAL) generates an HFP SIGNAL Command with the interrupt flag set and sends this Message to the host.

Logic. SIG2HFE (SPECIAL) calls BLDMSG to construct a "skeleton" HFP SIGNAL Command. The interrupt flag is set in the status field of the SIGNAL Command. SIG2HFE (SPECIAL) then calls SNDMSG to forward the HFP SIGNAL Command to the front-end CPM.

SERVER VIRTUAL TERMINAL SERVICE MODULE

12. SERVER VIRTUAL TERMINAL SERVICE MODULE ADAPTATION

12.1 Function

The ARPANET Server Virtual Terminal Service (SVTS) module enables programs on the host to be accessed by terminals connected to other hosts on the ARPANET. It implements the ARPANET Server Virtual Terminal process-to-service protocol defined in CAC Technical Memorandum No. 82. It also implements the ARPANET Telnet Protocol described in NIC Document No. 15372. The SVTS module performs the following functions.

1. It opens and closes ARPANET connections to hosts on the ARPANET.

2. It passes data between the local host and hosts on the ARPANET in accordance with Telnet protocol.

3. It maintains connection status information.

4. It performs Telnet option negotiation.

5. It enables front-end terminals to access the host.

12.2 Adaptation

The ARPANET SVTS module conforms to the operation of a "typical" service as described in section 7 beginning on page 48. This section adapts the SVTS module's structure to the "typical" service architecture.

12.2.1 Subroutine Naming Conventions.

Various SVTS module subroutines perform the same functions as those of the "typical" service. However, the names used for these subroutines are different. These differences will be resolved by placing the subroutine name from which the SVTS module's subroutine name is derived, in parens, immediately following any usage of an alternate SVTS module name.

12.2.2 SVTS module I/O Device. The Unix ARPANET software is implemented as a Unix I/O device. ARPANET open, close, read, and write I/O system calls corespond to "typical" service I/O system operations. In this context, the ARPANET software is the SVTS module's I/O device.

12.3 Operation

The basic operation of the SVTS module is the same as that of the "typical" service described in section 7.2 on page 48. However, in the SVTS module, the XMIT and NEWSEG subroutines invoke a Telnet output translator (TELNETOUT) and the READNET (READ) subroutine invokes a Telnet input translator (TELNETIN). The Telnet input and output translators (and several auxiliary subroutines) comprise what is known as the Telnet handler.

The Telnet input and output translators operate as finite state machines accepting input from various sources. These inputs are characters and Telnet commands (inter-mixed with Telnet data) that drive the machines from state to state. Figure 4 on the next page shows the Telnet handler configuration. A description of the Telnet input and output translators, as well as the auxiliary subroutines, follows in a later section.

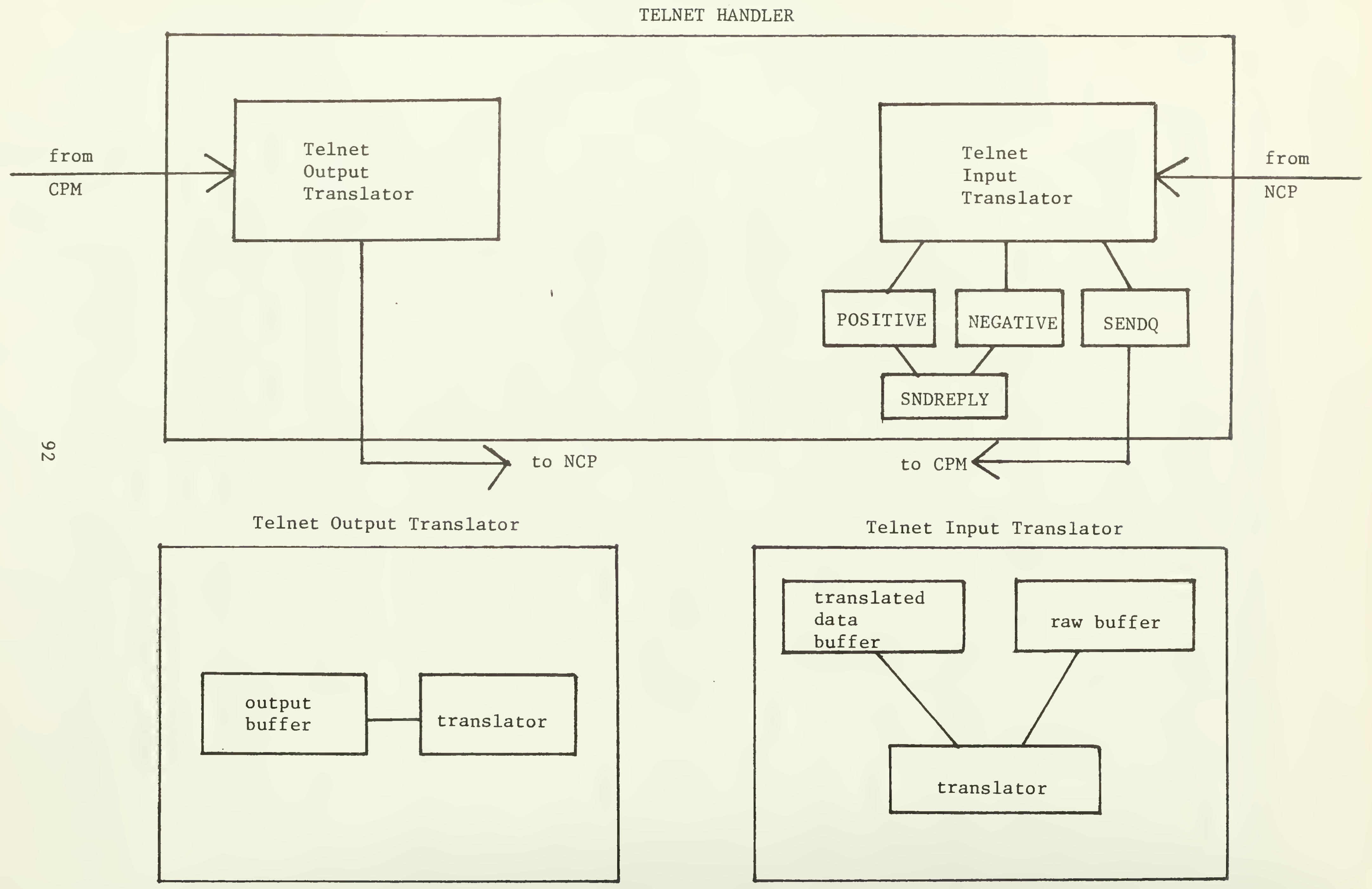

TELNET HANDLER
from CPM
Telnet Output Translator
Telnet Input Translator
from NCP
POSITIVE
NEGATIVE
SENDQ
SNDREPLY
to NCP
to CPM
Telnet Output Translator
output buffer
translator
Telnet Input Translator
translated data buffer
raw buffer
translator

Figure 4

12.4 Service Subroutine Hierarchy

The subroutine-calling hierarchy of the SVTS module is the same as that of the "typical" service described in section 7.3 on page 50 except for additional calls to TELNETIN and TELNETOUT as described above.

```
                           MAIN
                             |
                             |
           ------------------------------------
           |                                  |
           |                                  |
         HFEIN                              NETIN
           |                                  |
           |                                  |
  -----------------------           -------------------
  |   |  |   |  |   |  |            |   |    |    |   |
  | XMIT | EXEC | XON  |          CHVRFY | SIG2HFE | READNET
  |      |      |      |                |         |
BEGIN   SIG    END    XOFF            WRTNET   NETNAK
                                        |
                                      NEWSEG
```

12.5 Service State Transition Table

The SVTS module's logical states and state transitions are the same as those of the "typical" service described in section 7.4 on pages 51 and 52.

Service Data Structures

12.6.1 Channel Data Structure. The SVTS module's channel data structure is an expanded version of the "typical" service's channel data structure as described in section 7.5 on page 53. The following fields are used to hold channel information:

link	(16)	address of next channel list element
group	(16)	channel group identifier
member	(16)	channel member identifier
state	(Ø8)	channel state
flag	(Ø8)	channel flag bits
size	(16)	number of bytes waiting to be read from the I/O device
curseg	(Ø8)	identifier of TRANSMIT Command being output to the I/O device
fid	(Ø8)	I/O device identifier
nout	(16)	number of TRANSMIT Commands outstanding
seqhd	(16)	head of I/O output queue
oldaddr	(16)	last I/O starting address
bytslft	(16)	number of bytes remaining to be output from current TRANSMIT Command
bytstran	(16)	number of bytes last sent to the I/O device
rawbase	(16)	address to start in raw input buffer
rawptr	(16)	current position in raw buffer during translation
rawcnt	(16)	number of characters in raw buffer
trnbase	(16)	starting address in translated input buffer
trnptr	(16)	current position in translated buffer during translation
trncnt	(16)	number of characters in translated buffer
trnsiz	(16)	size of translated data buffer
outbase	(16)	starting address in output buffer
outsiz	(16)	size of output buffer
istate	(Ø8)	telnet input state
ostate	(Ø8)	telnet output state
options	(Ø8*n) ...	bit map of options in use

inscount	(Ø8)	....	number of network INS messages received
gacount	(16)	....	number of host go aheads received
inqhd	(16)	....	queue of TRANSMIT Commands awaiting go aheads from host.

12.6.2 Network Software Open Data Structure. When the SVTS module attempts to establish a network connection for a given logical channel, it must pass parameters with the request. These parameters are obtained from the TEXT field of the HFP BEGIN Comamnd requesting the initiation of the network connection and are stored in the Open Data Structure.

The Open Data Structure has the following fields (the numbers in parens indicate the field width in bits):

op	(Ø8)	....	used internally by the network software
type	(Ø8)	....	connection type
id	(16)	....	used internally by the network software
lskt	(16)	....	host's local socket for this connection
fskt	(32)	....	host's foreign socket for this connection
frnhost	(Ø8)	....	foreign host to which the connection is to be initiated
bsize	(Ø8)	....	number of bits per network logical byte
nomall	(16)	....	nominal allocation to be maintained
timeo	(16)	....	number of seconds to wait before timing out a network connection attempt
relid	(16)	....	internal to the network software

13. SVTS MODULE: PROGRAM ANALYSIS

The following subroutines comprise the Server Virtual Terminal Service module.

13.1 MAIN

Abstract. MAIN obtains IPC resources, initializes several data structure free lists, and provides the "driving loop" for the program.

Logic. The operation and logic of the MAIN subroutine is the same as that of the MAIN subroutine described in section 8.1 on page 54.

13.2 HFEIN (HFP-IN)

Abstract. HFEIN (HFP-IN) handles HFP and flow control commands from the front-end CPM. There are seven subroutines immediately subordinate to HFEIN (HFP-IN) that handle HFP Command-specific processing: BEGIN, END, XMIT (TRANSMIT), EXEC (EXECUTE), SIGNAL, XOFF, and XON.

Logic. The operation and logic of the HFEIN (HFP-IN) subroutine is the same as that of the HFP-IN subroutine described in section 8.2 on page 55.

13.3 BEGIN

Abstract. The BEGIN subroutine parses the BEGIN Command TEXT field and initiates the requested network connection.

Logic. BEGIN calls NBOPEN to initiate an I/O network open request. Connection information (connection type, foreign host, foreign socket, nominal allocation, timeout, and byte size) is copied from the TEXT field of the BEGIN Command into a network software open structure. BEGIN uses this now initialized structure to open a network connection to the specified foreign host. The system primitive NBOPEN is executed for this purpose. The rest of the operation and logic is the same as that of the BEGIN subroutine described in section 8.3 on page 56.

13.4 END

Abstract. The END subroutine terminates a logical channel by destroying the associated logical channel data structure. Data queued for output to the network I/O device may be discarded.

Logic. The operation and logic of the END subroutine is the same as that of the END subroutine described in section 8.4 on page 57.

13.5 XMIT (TRANSMIT)

Abstract. The XMIT (TRANSMIT) subroutine initiates data transfers to the network I/O device and enqueues incoming TRANSMIT Commands.

Logic. The operation and logic of the XMIT (TRANSMIT) subroutine is the same as that of the TRANSMIT subroutine described in section 8.5 on page 58 with the following addition. The TELNETOUT subroutine is called to translate characters from the host into Telnet protocol-defined ASCII, perform Telnet option processing, and translate certain command character sequences according to Telnet protocol.

13.6 EXEC (EXECUTE)

Abstract. The HFP EXECUTE Command is not used in this process-to-service protocol.

Logic. EXEC (EXECUTE) calls RESPOND to generate an HFP EXECUTE Response with an "unused" error status.

13.7 SIGNAL

Abstract. Like the EXECUTE Command, the HFP SIGNAL Command is not used in this process-to-service protocol.

Logic. The SIGNAL subroutine simply calls RESPOND to generate an HFP Signal Response with an "unused" error status.

13.8 XOFF

Abstract. The XOFF subroutine inhibits the receipt of network data

Logic. The operation and logic of the XOFF subroutine is the same as that of the XOFF subroutine described in section 8.8 on page 62.

13.9 XON

Abstract The XON subroutine restarts the flow of data from network software to the front-end CPM.

Logic. The operation and logic of the XON subroutine is the same as that of the XON subroutine described in section 8.9 on page 63.

13.10 NETIN (I/O-IN)

Abstract. NETIN (I/O-IN) receives I/O completion events from the network I/O device and dispatches control to an I/O completion event type-specific subroutine. The five subroutines called by NETIN (I/O-IN) are: CHVRFY (OPEN), WRTNET (WRITE), READNET (READ), NETNAK (RETRANS), and SIG2HFE (SPECIAL).

Logic. The operation and logic of the NETIN (I/O-IN) subroutine is the same as that of the I/O-IN subroutine described in section 8.10 on page 64.

13.11 CHVRFY (OPEN)

Abstract. CHVRFY (OPEN) completes the processing of an HFP BEGIN Comamnd. CHVRFY (OPEN) verifies the successful completion of the network connection and generates a BEGIN Response.

Logic. The operation and logic of the CHVRFY (OPEN) subroutine is the same as that of the OPEN subroutine described in section 8.11 on page 64 with the following addition. The TEXT field of the HFP BEGIN Response is filled in with the connection state, foreign host, foreign socket, number of messages allocated, number of bits allocated, local socket, and byte size.

13.12 WRTNET (WRITE)

Abstract. The WRTNET (WRITE) subroutine processes network write I/O completion events. These completion events are generated by the network I/O device to acknowledge the successful transmission of previous data.

Logic. The operation and logic of the WRTNET (WRITE) subroutine is the same as that of the WRITE subroutine described in section 8.12 on page 65.

13.13 NEWSEG

Abstract. The NEWSEG subroutine is called when all data for an HFP TRANSMIT Command has been processed. NEWSEG searches the I/O output queue for another HFP Command. If the HFP Command is a SIGNAL Command, it is processed. If the HFP Command is a TRANSMIT Command, TELNETOUT is called.

Logic. The operation and logic of the NEWSEG subroutine is the same as that of the NEWSEG subroutine described in section 8.13 on page 67 with the following addition. If a TRANSMIT Command is dequeued from the I/O output queue, NEWSEG calls TELNETOUT. TELNETOUT will translate the TRANSMIT Command data in accordance with Telnet protocol.

13.14 READNET (READ)

Abstract. The READNET (READ) subroutine is called when a network read I/O completion event is received. READNET (READ) obtains network data, calls TELNETIN to translate the data, formats it into an HFP TRANSMIT Command, and forwards it to the front-end CPM.

Logic. The operation and logic of the READNET (READ) subroutine is the same as that of the READ subroutine described in section 8.14 on page 68 with the following addition. READNET calls TELNETIN. TELNETIN translate the data in accordance with Telnet protocol.

13.15 NETNAK (RETRANS)

Abstract. When the transmission of data to a foreign host fails, the network software generates a negative acknowledgement I/O completion event. NETNAK (RETRANS) is called when a negative acknowledgement I/O completion event is received.

Logic. The operation and logic of the NETNAK (RETRANS) subroutine is the same as that of the RETRANS subroutine described in section 8.15 on page 69.

13.16 SIG2HFE (SPECIAL)

Abstract. When the network software receives an ARPANET Host-Host Interrupt by Sender (INS) protocol message, it generates a special I/O completion event. SIG2HFE (SPECIAL) is called when a special I/O completion event is received. SIG2HFE (SPECIAL) generates an HFP SIGNAL Command with the interrupt flag set and sends this Message to the host.

Logic. SIG2HFE (SPECIAL) calls BLDMSG to construct a "skeleton" HFP SIGNAL Command. The interrupt flag is set in the status field of the SIGNAL Command. SIG2HFE (SPECIAL) then calls SNDMSG to forward the HFP SIGNAL Command to the front-end CPM.

14. TELNET HANDLER

<u>14.1 Telnet Input Translator</u>

14.1.1 <u>Buffer Utilization</u>.

As Figure 4 on page 92 illustrates, the Telnet input translator has two buffers for each channel. The raw input buffer receives characters and Telnet commands (data) directly from the NCP and thus the network and foreign host. The translator then filters the Telnet commands and changes the characters from Telnet representation to host representation. The translated data buffer holds up to one line of these translated characters. This line of translated characters (data) is then placed into a message containing a TRANSMIT Command and sent to the CPM and thus to the local host.

<u>14.2 Telnet Output Translator</u>

14.2.1 <u>Buffer Utilization</u>.

As the diagram on page 92 indicates, the Telnet output translator has only one buffer: the output buffer. The Telnet output translator receives characters (data) from the CPM and thus from the local host. The translator changes the characters from host representation to Telnet representation. The translated data is placed in the output buffer and then sent to the network and thus to a foreign host.

14.3 Telnet Handler State Transition Tables

14.3.1 Input Translator. The following table depicts the Telnet input translator's logical states, actions, and state transitions.

STATE	CHARACTER	ACTION	NEXT STATE
NEUTRAL	<IAC>	---	IAC SEEN
	NULL	discard	---
	<LF> Line Feed	send queued data to host	NEUTRAL
	<CR>	---	CR_SEEN
	all others	place in buffer	NEUTRAL
IAC_SEEN	<IAC>	place in buffer	NEUTRAL
	<DM> <AO> <SE> <NOP> <IP> <AYT> <SB>	discard	NEUTRAL
	BREAK	send SIGNAL Command	NEUTRAL
	<EC> Erase Character	back up pointer, decrement count	NEUTRAL
	<EL> Erase Line	reset pointer zero count	NEUTRAL

STATE	CHARACTER	ACTION	NEXT STATE
	<GA>	send blank TRANSMIT Command	NEUTRAL
	WILL	---	WILL_SEEN
	DO	---	DO_SEEN
	WON'T	---	WON'T_SEEN
	DON'T	---	DON'T SEEN
	all others	back up raw pointer and count	NEUTRAL
CR_SEEN	<LF>	place <CR><LF> in buffer and send TRANSMIT Command to host	NEUTRAL
	all others	place <CR> in data stream and rescan other character	NEUTRAL
DO_SEEN WILL_SEEN		call positive negotiator	NEUTRAL
WON'T_SEEN DON'T_SEEN		call negative negotiator	NEUTRAL

14.3.2 Output Translator. The following table depicts the Telnet output translator's logical states, actions, and state transitions.

STATE	CHARACTER	ACTION	NEXT STATE
NEUTRAL	<CR>	---	CR SEEN
	<LF>	output <CR><LF>	NEUTRAL
	<IAC>	output <IAC><IAC>	NEUTRAL
	all others	place in output buffer	NEUTRAL
CR_SEEN	<LF>	output <CR><LF>	NEUTRAL
	all others	output <CR><NULL>, back up pointer	NEUTRAL

15. TELNET HANDLER: PROGRAM ANALYSIS

15.1 TELNETIN

Abstract. The TELNETIN subroutine translates data and handles offloaded Telnet options.

Logic. READNET calls TELNETIN to translate network data. The TELNETIN subroutine determines whether the translated data buffer is large enough to accomodate the new data. If the number of characters in the raw buffer ('rawcnt') in the channel data structure and the number of characters in the translated data buffer ('trncnt') are greater than the size of the translated data buffer ('trnsiz'), a new translated data buffer is necessary. The TELNETIN subroutine copies the characters from the old translated data buffer into the new. The 'trnbase' (pointer to the beginning of the translated data buffer) and 'trnsiz' (size of the translated data buffer) fields in the channel data structure are updated.

If the old translated data buffer is large enough, TELNETIN saves the current position in the translated data buffer during translation ('trnptr').

TELNETIN sets the current postion in the raw buffer during translation ('rawptr'). The current position is the beginning of the raw buffer, pointing to the first character obtained by the read I/O system operation. TELNETIN then switches on the input state.

Case 1: State=NEUTRAL

If the character is <IAC> or Interpret As Command, no action is taken and the state becomes IAC_SEEN.

If the character is NULL, it is simply discarded.

If the character is <LF> or Line Feed, TELNETIN must determine whether a Synch is being performed. A Synch signal consists of a Host/Host Protocol INS command, coupled with the Telnet command <DM> or Data Mark. The INS command, is used to invoke special handling of the data by the process which receives it. In this mode, the data stream is immediately scanned for "interesting" (<AO>, <IP>, <AYT>) signals, discarding intervening data.

The Telnet command <DM> is the synchronizing mark in the data stream which indicates that any special signal has already occurred and the recipient can return to normal processing of the data stream. When a <DM> arrives before its associated INS, the recipient should not process the data stream further until the matching INS is received, in order to insure that the two ends of the connection remain synchronized. In some cases, several Synch's may be sent in succession. In general, this will require a count of the INS's received so as to properly pair them with the associated <DM's>.

Thus, if the 'inscount' field (INS counter) in the channel data structure is equal to zero, normal processing of the data stream may occur. The character is stored in the translated data buffer and the buffer's character count is incremented. SENDQ calls BLDMSG to construct a "skeleton" HFP TRANSMIT Command. SENDQ copies in network data and transmits it to the CPM and thus to the host. TELNETIN resets the pointer to the start of the translated data buffer ('trnbase').

If the character is <CR> or Carriage Return, TELNETIN must determine whether a Synch is being performed. If the 'inscount' field in the channel data structure is equal to zero, normal processing of the data stream may take place. No action is taken and the state becomes CR_SEEN.

All other characters are data. If the 'inscount' field is equal to zero, normal processing of the data occurs. TELNETIN stores the particular character in the translated data buffer and the buffer count is incremented.

Case 2: State=IAC SEEN

If the command is another <IAC>, the <IAC> is placed in the translated data buffer and sent as data. The number of characters in the buffer is incremented and the state becomes NEUTRAL.

If any of the following Telnet commands are received, they are discarded and the state becomes NEUTRAL: <DM> (Data Mark), <AO> (Abort Output), <SE> (End of Subnegotiation), <NOP> (No operation), <IP> (Interrupt Process), <AYT> (Are You There?), and <SB> (Start Subnegotiation).

If the command is BREAK, a SIGNAL Command with the Synchronize, Flush Away, and Interrupt bits set in the control field is sent to the CPM and thus to the host. The go-ahead counter in the channel data structure is decremented and the state becomes NEUTRAL.

If the command is <EC> or Erase Character and the number of characters in the translated data buffer is greater than zero, the pointer to the current position in the buffer is moved back to the preceding undeleted character. The character count in the translated data buffer ('trncnt') is decremented and the state becomes NEUTRAL.

If the command is <EL> or Erase Line, the pointer to the start of the translated data buffer ('trnbase') is reset and the number of characters in the buffer ('trncnt') becomes zero. (The translated data buffer may never hold more than one line of data). The state becomes NEUTRAL.

If the command is <GA> or Go Ahead, TELNETIN calls SENDQ to send a TRANSMIT Command (with go-ahead flag set) to the CPM and thus to the host. The pointer to the start of the translated data buffer ('trnbase') is reset and the state becomes NEUTRAL.

The following Telnet commands indicate the initialization of some option negotiation. As the list indicates, no action is taken, only state transitions occur.

CHARACTER	NEXT STATE
WILL	WILL_SEEN
DO	DO_SEEN
WON'T	WON'T SEEN
DON'T	DON'T SEEN

If a character does not match any of the previously mentioned commands, the pointer to the current position in the raw buffer ('rawptr') is backed up. The number of characters in the raw buffer ('rawcnt') is incremented and the state becomes NEUTRAL.

Case 3: State=CR SEEN

If the character is not <LF>, <CR> is placed in the data stream and the number of characters in the translated data buffer ('trncnt') is incremented.

If the character is <LF> or Line Feed, it is placed in <CR><LF> and the translated data buffer count is incremented. TELNETIN calls SENDQ to send a TRANSMIT Command to the CPM and thus to the local host. The pointer to the start of the translated data buffer ('trnbase') is reset and the state becomes NEUTRAL.

Case 4: State=DO SEEN, State=WILL SEEN

In each of these states, positive options are negotiated. TELNETIN calls the subroutine POSITIVE to handle the WILL/DO negotiations and the state becomes NEUTRAL.

Case 5: State=WON'T SEEN, State=DON'T SEEN

In each of these states, negative options are negotiated. TELNETIN calls the subroutine NEGATIVE to handle the WON'T/DON'T negotiations and the state becomes NEUTRAL.

In all five cases mentioned above, after the character translation is complete, the number of characters in the raw input buffer becomes zero and the pointer to the current position in the buffer is copied for the next data stream.

15.2 Disposition Table

This table determines the disposition of telnet options (see POSITIVE and NEGATIVE). In this implementation, all option negotiations are refused.

DISPOSITION	TELNET OPTION
REFUSE	binary transmission
REFUSE	echo
REFUSE	reconnection
REFUSE	suppress go ahead
REFUSE	approximate message size
REFUSE	status
REFUSE	timing mark
REFUSE	RCTE
REFUSE	negotiate output line size
REFUSE	negotiate output page size
REFUSE	negotiate output carriage return disposition
REFUSE	negotiate output horizontal tab stops
REFUSE	negotiate output horizontal tab disposition
REFUSE	negotiate output form feed disposition
REFUSE	negotiate output vertical tab stops
REFUSE	negotiate output vertical tab disposition
REFUSE	negotiate output line feed disposition

15.3 POSITIVE

Abstract. The POSITIVE subroutine handles WILL/DO negotiations.

Logic. Upon receipt of a WILL or DO command, TELNETIN calls POSITIVE. The POSITIVE subroutine sets the state to NEUTRAL. If the option is not legal, POSITIVE returns. If the bit map of turned-on options in the channel data structure indicates that the option is already being implemented, POSITIVE returns.

If neither of the preceding cases is true, the POSITIVE subroutine examines the Disposition Table (page 111) to determine how to process each option.

If the option is accepted, POSITIVE sets an option specific bit in the channel data structure. POSITIVE then calls SNDREPLY to send an affirmative Telnet reply to the foreign host.

If the option is refused, POSITIVE calls SNDREPLY to send a negative reply to the foreign host.

If the option is to be forwarded to the local host, it is placed in the translated data buffer. The number of characters in the buffer is incremented by three and an <IAC><DO/WILL> <option> is constructed. POSITIVE then calls SENDQ to send a TRANSMIT Command to the CPM and thus to the local host. The pointer to the beginning of the translated data buffer ('trnbase') is then reset.

15.4 NEGATIVE

Abstract. The NEGATIVE subroutine handles WON'T and DON'T negotiations.

Logic. Upon receipt of a WON'T or DON'T command, TELNETIN calls NEGATIVE. The NEGATIVE subroutine sets the state to NEUTRAL. If the option is not legal, NEGATIVE returns.

The NEGATIVE subroutine examines the Disposition Table to determine how to process each option. If an option has previously been accepted, NEGATIVE clears an option-specific flag in the channel data structure. NEGATIVE calls SNDREPLY to send a proper reply to the foreign host.

If the option is to be forwarded to the local host, it is placed in the translated data buffer. The number of characters in the buffer is incremented by three. NEGATIVE calls SENDQ to send a TRANSMIT Command to the CPM and thus to the local host. The pointer to the beginning of the translated data buffer is then reset.

15.5 SENDQ

Abstract. The SENDQ subroutine sends translated Telnet characters (data) to the local host.

Logic. The SENDQ subroutine copies the number of characters in the translated data buffer ('trncnt'). SENDQ then calls BLDMSG to construct a "skeleton" HFP TRANSMIT Command. The data is then copied into the TRANSMIT Command. SENDQ then performs any host-specific data translation. If the host has not issued a Go Ahead, the message segment is enqueued to await transmission. Otherwise, the message is immediately transmitted to the host. The number of characters in the translated data buffer is now zero and the pointer to the beginning of the buffer is reset.

15.6 SNDREPLY

Abstract. The SNDREPLY subroutine sends data generated internally in the SVTS module to the network.

Logic. If the logical channel is in state ESTAB (established), SNDREPLY issues a network write I/O operation.

However, if the state of the logical channel is BUSY, SNDREPLY calls BLDMSG to construct a "skeleton" HFP TRANSMIT Command. The status field in the HEADER of the message is set to negative one (-1) to prevent the message from being sent back to the local host as an HFP Response (see NEWSEG in section 13.13 on page 1Ø1). The data is copied into the Message and the Message is added to the I/O output queue.

15.7 TELNETOUT

Abstract. The TELNETOUT subroutine handles translation to Network Virtual Terminal Representation (NVT) from the local host.

Logic. XMIT or NEWSEG call TELNETOUT to obtain data from the the local host. TELNETOUT extracts data size information from the TRANSMIT TEXT. TELNETOUT then determines whether the output buffer is large enough to accomodate the new data. If it is not large enough, a new output buffer is obtained. TELNETOUT saves the go-ahead flag in the channel data structure. TELNETOUT then performs any host-specific data translation. The pointer to the start of the output base is reset. The number of bytes remaining to be output to the network from TEXT ('bytslft') is set to zero.

If the number of bytes of data sent from the host is greater than zero, TELNETOUT obtains the first character. TELNETOUT then switches on the output state.

<u>Case 1</u>: <u>State=NEUTRAL</u>

If the character is <CR> or Carriage Return, no action is taken and the state becomes CR_SEEN.

If the character is <LF> or Line Feed, it is translated to <CR><LF> and the state becomes NEUTRAL. The number of characters in the output buffer is incremented. The number of bytes remaining to go to the network from TEXT ('bytslft') is incremented by two.

If the character is <IAC>, it is translated to <IAC><IAC> and the state becomes NEUTRAL.

All other characters are simply placed in the output buffer and the state remains NEUTRAL. The number of characters in the output buffer is incremented.

<u>Case 2</u>: <u>State=CR_SEEN</u>

If the character is <LF> or Line Feed, it is translated to <CR><LF> and the state becomes NEUTRAL. The number of characters in the output buffer is incremented.

All other characters are translated into a <CR><NULL> sequence and the pointer to the current position in the output buffer is moved back to rescan the character. The state becomes NEUTRAL.

After the character translation is complete, the pointer to the beginning of the output buffer is reset. If the go-ahead flag in the channel data structure is set, an <IAC><GA> is placed in the data stream.

If the go-ahead flag of the message containing the TRANSMIT Command (data just translated) is cleared, any data queued for the foreign host is sent. If no data is queued, a Telnet Go-Ahead command and the data just translated is sent to the foreign host.

PROGRAM ACCESS SERVICE MODULE

16. PROGRAM ACCESS SERVICE MODULE ADAPTATION

16.1 Function

The Program Access Service (PAS) module will enable programs running in the H6000 to execute arbitrary programs in the front end. It will implement the Program Access process-to-service protocol described in CAC Technical Memorandum No. 81. The PAS module performs several functions using the Unix pseudo-Teletype (PTY) facility.

1. It enables programs on the H6000 to log in to and log out of the Unix system.

2. It enables programs on the H6000 to run programs under Unix (for example, Telnet).

3. It passes data between programs on the H6000 and programs running under Unix.

16.2 Adaptation

The PAS module conforms to the operation of a "typical" service as described in section 7 beginning on page 48. This section will adapt the PAS module's structure to the "typical" service architecture.

16.2.1 Subroutine Naming Conventions. Various PAS module subroutines perform the same functions as those of the "typical" service. However, the names used for these subroutines are different. These differences is resolved by placing the subroutine name from which the PAS module's subroutine name is derived, in parens, immediately following any usage of an alternate PAS module name.

16.2.2 PAS module I/O Device. The Unix pseudo-Teletype (PTY) mechanism is implemented as a non-blocking I/O device. Pseudo-Teletypes are a Unix community term for software-controlled terminals. Pseudo-Teletypes are dual-sided, having a master side and a slave side.

Both the master and the slave sides have open, close, read, and write I/O system entry points. The slave I/O system write software is "coupled" to the master I/O system read software. The slave I/O system read software is "coupled" to the master I/O system write software. Thus, data "written" to one side may be "read" by the other side.

Slave software operates like a standard Unix terminal device. Slave write I/O operations process data as if it were to output to a terminal "printer." Master read I/O operations obtain this data. Master write I/O operations process data as if it were generated by a terminal "keyboard." Slave read I/O operations obtain this data.

The PTY facility allows arbitrary front-end programs to be initiated with the slave side of a pseudo-teletype as the "controlling" terminal. The PAS module manages the master side of all pseudo-Teletypes. Data flowing through the master side of a pseudo-Teletype is coupled to an HFP logical channel. Thus, the PAS module and the PTY facility enable host processes to execute arbitrary front-end programs.

16.3 Operation

The basic operation of the PAS module is the same as that of the "typical" service described in section 7.2 on page 48, with the addition of the Go-Ahead facility. The Go-Ahead facility is required to handle host half-duplex terminals. A detailed description of the Go-Ahead facility is found in the Program Access process-to-service protocol specification.

16.4 Service Subroutine Hierarchy

The subroutine-calling hierarchy is the same as the calling hierarchy of the "typical" service described in section 7.3 on page 50, with the addition of two subroutines (HSTGA, GOAHEAD) to implement the Go-Ahead facility.

The PAS module's form of operation dictates the internal structure of the service. Two major divisions in program logic occur to accomodate input from the front-end CPM and the Unix PTY facility. The chart on the following page illustrates this hierarchy. Each node in the hierarchy represents a subroutine.

```
                             MAIN
                               |
                               |
              --------------------------------------------
               |                                        |
               |                                        |
            HFEIN                                    PTYIN
               |                                        |
  -------------|~---------------      --------------------|-------
  |    |   |    |   |   |   |        |    |    |    |    |    |
  |  END   |  XOFF  |  XON  |        | CHVRFY  | GOAHEAD | READPTY
  |        |        |       |        |         |         |
BEGIN   SIGNAL    XMIT     EXEC   DEATH    WRTPTY     RETRANS
                    |                          |
                  HSTGA                     NEWSEG
                                               |
                                            HSTGA
```

16.5 Service State Transition Table

The PAS module's states and state transitions are the same as those of the "typical" service described in section 7.4 on pages 51 and 52. Some of the ACTIONS, however, are PAS module-specific.

The PAS module is programmed as a finite state machine. Inputs are received from the front-end CPM and Unix PTY facility. Each input is associated with a logical channel. The input type and current channel state determine immediate action and next channel state. The table on the following page depicts the channel states, actions, and state transitions.

STATE	EVENT(input)	ACTION(output)	NEXT STATE
NULL	BEGIN Command	Open PTY	PEND
PEND	PTY open success	notify host	ESTAB
	PTY open failure	notify host	NULL
	END Command	close PTY device	
		free resources	NULL
	SIGNAL Command	error response to host	PEND
	EXECUTE Command	error response to host	PEND
ESTAB	PTY error	notify host	
		free resources	NULL
	TRANSMIT Command	data to PTY device	BUSY
	data from PTY device	TRANSMIT Command	ESTAB
		to host	
	END Command	close channel	
		free resources	NULL
	SIGNAL Command	process command	ESTAB
	EXECUTE Command	process command	ESTAB
BUSY	PTY error	END Command to host	
		flush buffers	
		free resources	NULL
	TRANSMIT Command	queue command	BUSY
	data from PTY device	TRANSMIT Command	BUSY
		to host	
	I/O completion -		
	transmit queue empty		ESTAB
	I/O completion -		
	queue not empty	start a new transfer	BUSY
	END Command	let data drain	TERM
	SIGNAL Command	process command	BUSY
	EXECUTE Command	process command	BUSY
TERM	data drained to device	END Response to host	NULL
	SIGNAL Command	process command	TERM
	EXECUTE Command	process command	TERM
	PTY error	END Response to host	NULL

16.6 Service Data Structures

16.6.1 Channel Data Structure. The form and usage of the PAS module's channel data structure is the same as that of the "typical" service's channel data structure described in section 7.5 on page 53. The PAS module's channel data structure, however, has two additions: a pointer to a PTY data structure and a variable to store the process identifier of the program currently attached to the slave side of the PTY.

16.6.2 PTY Data Structure. When the PAS module receives a BEGIN Command, it allocates a PTY to the logical channel. This PTY is described by two Unix I/O system file names, a PTY slave file name, and a PTY master file name. The last character of the slave and master file names determine the specific PTY to be referenced. The last character of the slave and master file names are stored in the PTY structure.

The PTY data structure has the following fields (the numbers in parens indicates the field width in bits):

```
name    (08) ...... last character of PTY filenames
inuse   (08) ...... set to 1 if the PTY structure is
                    in use.
```

17. PAS MODULE: PROGRAM ANALYSIS

The following subroutines comprise the Progam Access Service module.

17.1 MAIN

Abstract. MAIN obtains IPC resources, initializes several data structure free lists, and provides the "driving loop" for the program.

Logic. The operation and logic of the MAIN subroutine is the same as the MAIN subroutine described in section 8.1 on page 54 with the addition of a small piece of initialization software that marks all PTY data structures as unused.

17.2 HFEIN (HFP-IN)

Abstract. HFEIN (HFP-IN) handles HFP and flow control commands from the front-end CPM. There are seven subroutines immediately subordinate to HFEIN (HFP-IN) that handle HFP and flow control commands: BEGIN, END, XMIT (TRANSMIT), EXEC (EXECUTE), SIGNAL, XOFF, and XON.

Logic. The operation and logic of the HFEIN (HFP-IN) subroutine is the same as that of the HFP-IN subroutine described in section 8.2 on page 55.

17.3 BEGIN

Abstract. The BEGIN subroutine parses the HFP BEGIN Command TEXT field, assigns a PTY to the logical channel, and initiates a "logger" program on the slave side of the PTY. This logger program will handle host login requests and initiate front-end programs.

<u>Logic</u>. BEGIN searches the PTY data structures for a free PTY. If one is not found, RESPOND is called to generate an HFP BEGIN Response with an error indication of "no-resources." If an unused PTY is found, BEGIN manufactures the correct Unix file name and issues a non-blocking Unix open I/O operation. Later, the Unix I/O system will generate an open I/O completion event. CHVRFY (OPEN) will be executed and will complete the processing of the HFP BEGIN Command. BEGIN then calls MAKCHAN to obtain a channel data structure. If there are no free channel data structures, BEGIN calls RESPOND to generate an HFP BEGIN Response with an error indication of "no-resources." If a channel data structure is found, BEGIN initializes several variables. Finally, BEGIN spawns a logger process whose controlling terminal is the slave side of the newly assigned PTY.

<u>17.4</u> <u>END</u>

<u>Abstract</u>. The END subroutine terminates a logical channel by destroying the associated logical channel data structure. Data queued for output to the PTY I/O device may be discarded.

<u>Logic</u>. The operation and logic of the END subroutine is the same as that of the END subroutine described in section 8.4 on page 57.

17.5 XMIT (TRANSMIT)

Abstract. The XMIT (TRANSMIT) subroutine enqueues an incoming HFP TRANSMIT Command on the logical channel I/O output queue. If the PTY is not busy, an I/O write operation is initiated.

Logic. The operation and logic of the XMIT (TRANSMIT) subroutine is the same as that of the TRANSMIT subroutine described in section 8.5 on page 58 with the following addition.

If all TRANSMIT Command data was transferred to the master side of the PTY, RESPOND is called to generate an HFP TRANSMIT Response with a "success status" indication. If the Go-Ahead bit in the TRANSMIT Command was zero (go-ahead), HSTGA is called to initiate a data transfer from the PTY to the host.

17.6 HSTGA

Abstract. HSTGA is called in response to go-ahead functions from the host. HSTGA initiates data transfers from the PTY to the host.

Logic. If data is available from the PTY, and if the XOFF flag is not inhibiting data input to the front end CPM, HSTGA calls READPTY (READ) to obtain the data and pass it on to the host via the front-end CPM.

17.7 EXEC (EXECUTE)

Abstract. The HFP EXECUTE Command is not used for this process-to-service protocol. This routine handles the error situation when one arrives.

Logic. The EXEC (EXECUTE) subroutine calls RESPOND to generate an HFP EXECUTE Response with a "not implemented" error status indication.

17.8 SIGNAL

Abstract. The SIGNAL subroutine executes the HFP SIGNAL Command by performing data flushing, channel synchronization, and interrupt functions.

Logic. The data flushing and channel synchronization functions are implemented as described in section 8.7 on page 60. The interrupt function is implemented by calling the system primitive STTY. This function causes a Unix standard Interrupt signal to be sent to the process controlling the slave side of the PTY. In most cases, a Unix Interrupt signal causes program termination. A Unix facility exists to "catch" these signals and interpret them in a program-specific manner. Some Unix programs use this facility. The action of calling STTY will be delayed if the synchronize flag is on and data is in the I/O output queue. The NEWSEG subroutine will complete processing of the SIGNAL Command.

17.9 XOFF

Abstract. The XOFF subroutine inhibits the receipt of PTY generated data.

Logic. The operation and logic of the XOFF subroutine is the same as that of the XOFF subroutine described in section 8.8 on page 62.

17.10 XON

Abstract The XON subroutine restarts the flow of data from network software to the front-end CPM.

Logic. The operation and logic of the XON subroutine is the same as that of the XON subroutine described in section 8.9 on page 63.

17.11 PTYIN (I/O-IN)

Abstract. PTYIN (I/O-IN) receives I/O completion events from the pseudo-Teletype I/O device and dispatches control to an I/O completion event type-specific subroutine. The five subroutines called by PTYIN (I/O-IN) are: CHVRFY (OPEN), WRTPTY (WRITE), READPTY (READ), GOAHEAD, and DEATH.

Logic. The operation and logic of the PTYIN (I/O-IN) subroutine is the same as that of the IO-IN subroutine described in section 8.10 on page 64. If the IPC event returned is not an I/O completion event, but rather a DEATH event (generated by the Unix processs' control software to detail a child process death), PTYIN (I/O-IN) calls DEATH.

17.12 CHVRFY (OPEN)

Abstract. CHVRFY (OPEN) completes the processing of an HFP BEGIN Command. CHVRFY (OPEN) verifies the successful completion of a PTY open I/O operation and generates a BEGIN Response.

Logic. The operation and logic of the CHVRFY (OPEN) subroutine is the same as that of the OPEN subroutine described in section 8.11 on page 64.

17.13 WRTPTY (WRITE)

Abstract. The WRTPTY (WRITE) subroutine processes PTY write I/O completion events. These completion events are generated by the PTY I/O device to acknowledge the successful transmission of previous data.

Logic. The operation and logic of the WRTPTY (WRITE) subroutine is the same as that of the WRITE subroutine described in section 8.12 on page 65.

17.14 NEWSEG

Abstract. The NEWSEG subroutine is called when all data for an HFP TRANSMIT Command has been processed. NEWSEG searches the I/O output queue for another HFP Command. If the HFP Command is a SIGNAL Command, it is processed. Otherwise, it is returned to the caller.

Logic. The operation and logic of the NEWSEG subroutine is the same as that of the NEWSEG subroutine described in section 8.13 on page 67 with the following addition. If the TRANSMIT Command whose processing has just been completed has the Go-Ahead bit set to zero (go ahead), NEWSEG calls HSTGA to potentially initiate a data transfer from the PTY to the host via the front-end CPM.

17.15 READPTY (READ)

Abstract. READPTY (READ) obtains slave process data, formats it into an HFP TRANSMIT Command, and forwards it to the front-end CPM.

Logic. The operation and logic of the READPTY (READ) subroutine is the same as that of the READ subroutine described in section 8.14 on page 68 with the following addition. Both the

XOFF and GO-AHEAD flags are checked in the channel data structure flag field before a PTY I/O read operation is initiated. If either of these flags is on, PTY data is available, and READPTY will exit.

17.16 GOAHEAD

Abstract. The GOAHEAD subroutine is called when a PTY go-ahead event is received. When an event of this type is received, the program which manages the slave side of the PTY has issued a read I/O operation.

Logic. If the front-end CPM has not turned off data input via an XOFF input, an empty (no data in the TEXT field) HFP TRANSMIT Command with the Go-Ahead flag set to zero (go ahead) is sent to the host via the front-end CPM.

17.17 DEATH

Abstract. DEATH is called when a Unix process-control death event is received. This event indicates that the program which manages the slave side of the PTY has completed processing.

Logic. If an associated logical channel data structure can be found, KILLCHAN is called to terminate the logical channel, de-allocate the channel data structure, and send an HFP END Command to the host via the front-end CPM.