UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN
ENGINEERING

NOTICE: **Return or renew all Library Materials! The *Minimum* Fee for each Lost Book is $50.00.**

JUL 6 1988

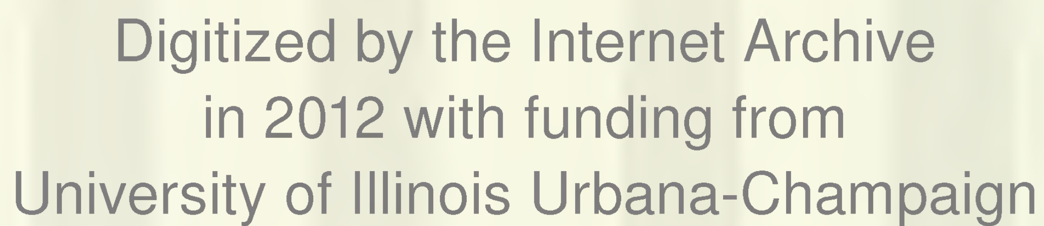
The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.
To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING

L161—O-1096


Digitized by the Internet Archive
in 2012 with funding from
University of Illinois Urbana-Champaign

http://archive.org/details/networkingresearuniv

Engin

ENGINEERING LIBRARY
UNIVERSITY OF ILLINOIS
URBANA, ILLINOIS

CONFERENCE ROOM

Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA ILLINOIS 61801

CAC Document Number 221
CCTC-WAD Document Number 7502

Networking Research in Front Ending and Intelligent Terminals

Experimental Network Front End Functional Description

January 15, 1977

SEP 13 1977

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

ENGINEERING CONFERENCE ROOM

MAR 17 1982

APR 20 1982

MAR 20 REC'D

L161—O-1096

CAC Document Number 221
CCTC-WAD Document Number 7502

Networking Research in Front Ending and Intelligent Terminals

EXPERIMENTAL NETWORK FRONT END FUNCTIONAL DESCRIPTION

by
Steven F. Holmgren
Peter A. Alsberg
Gary R. Grossman
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

January 15, 1977

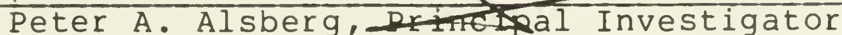
Approved for Release: 
Peter A. Alsberg, ~~Principal~~ Investigator

TABLE OF CONTENTS

INTRODUCTION

Background 1

The Hardware Configuration 1

Overview of Front-end Software Requirements 2

ADDITIONS TO UNIX

General 8

Inter-Process Communication (IPC) 9

Non-Blocking I/O 10

HFP SOFTWARE MODULES IN THE FRONT END

Introduction 12

Channel Protocol Module (CPM) 14

ARPANET Host-Host Service (HHS) Module 16

Program Access Service (PAS) Module 18

ARPANET Server Virtual Terminal Service (SVTS) Module 20

APPENDIX I. CHANNEL PROTOCOL MODULE 22

APPENDIX II. ARPANET HOST-HOST SERVICE MODULE 41

APPENDIX III. PROGRAM ACCESS SERVICE MODULE 60

APPENDIX IV. ARPANET SERVER VIRTUAL TERMINAL SERVICE MODULE . 81

INTRODUCTION

Background

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 a 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 experimental network front end is being developed on a DEC PDP-11/70.

The operating system for the front end is a modified Unix operating system. Unix, a general-purpose PDP-11 operating system developed by Bell Telephone Laboratories, supports time-sharing and has facilities such as editors, compilers and word processors. The CAC has already enhanced the Unix system by adding a Network Control Program (NCP) to it. 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 new software modules which are needed to support front-end experimentation.

The Hardware Configuration

The front end itself is a Digital Equipment Corporation (DEC) PDP-11/70 computer with 128K words of memory and disk storage. Hardware interfaces to terminals, the ARPANET, and the

H6000 will be provided. 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.

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 Interchanges (CPI) 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.

<u>Overview of Front-end Software Requirements</u>

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 front end should reduce the extent and complexity of host-resident software. As a result, host performance should improve considerably. Proper design of front-end and interface software should also yield improved security.

The network software can be thought of as a set of services provided to host processes or users. These services allow the network and the various hosts connected to the network to be conveniently used. Offloading shifts the major burden of providing these services from the local host to the front end.

The major services to be offloaded to the experimental network front end are:

1. the Network Control Program (NCP), which controls access to the network and to remote hosts on the network, and

2. Telnet, which provides an interface between terminals (which may be of widely differing types) and interactive processes on remote hosts.

Since an NCP has already been added to the Unix system, only a simple software module, a message relay, must be added to provide access to the NCP. This software module is the ARPANET Host-Host Service Module, which is described in detail below.

Telnet is usually implemented as two separate facilities: User Telnet and Server Telnet. User Telnet accepts input from terminals and initiates connections to ARPA Network hosts. Server Telnet accepts those connections. Normally, a user of a Telnet facility invokes a User Telnet "program" at his local ARPA Network host and is connected to a server Telnet "program" at a remote ARPA Network host. The result is that the user's terminal appears to be connected to the remote host as if it were a local terminal at the remote host.

The Program Access process-to-service protocol will be used to access the existing Unix User Telnet program in the front end. The Program Access Service module will use the Unix pseudo-Teletype (PTY) mechanism to make a terminal connected to

the host appear to be directly connected to the front end. Then the terminal connected to the host can use the Unix User Telnet program as if it were a terminal connected to the front end.

The implementation of Server Telnet presents a more difficult problem. A Server Virtual Terminal Service Module will be implemented in the front end. This module (described in detail below) interfaces with the NCP to manipulate network connections and to transmit data. Other Server Telnet functions which are appropriately offloaded to the front end will also be handled by this module.

To support process-to-service communication (i.e. communication between processes in the host and services in the front end) a basic mechanism must be provided for the host and the front end to communicate. 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.

The process-to-service protocols specify the content and type of the messages by which host processes communicate with the various service modules in the front end. Those process-to-service protocols defined to date are:

1. ARPANET Host-Host Process-to-Service Protocol (CAC Technical Memorandum No. 80),

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).

Thus these protocols define the communications between host processes and the three service modules described above.

By means of the channel protocol, logical channels are set up between the host and the front end, 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. These types are

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 front end will contain a software module, the Channel Protocol Module (CPM), which manages the logical channels and serves as a multiplexor. In one direction, messages from the host are obtained from the lower-level host-to-front-end communications link and transmitted to the correct service. In the other direction, the CPM accepts messages from the services and sends them on their way to the host. The host also contains a CPM which similarly manages the other ends of the logical channels. The front-end CPM is described in detail below.

Two additional facilities must be added to the Unix operating system. These are an inter-process communication facility and a non-blocking I/O facility. Brief descriptions of these facilities are given below. In addition, device drivers will be added to Unix to manage the IMP-to-host interface, the H6000 to PDP-11 interface, and VIP terminals.

The HFP identifies a "link handler" which operates the hardware interface between the host and the front end. The 11/70-H6000 interface device driver will fulfill this function in the ENFE.

The diagram on the next page shows the network-front-end software configuration. Note that this NCP consists of two software modules, one in the operating system and one which runs in user mode.

ENFE SOFTWARE ARCHITECTURE

H6000

Host Interface Device Driver

Channel Protocol Module

Host-Host Service Module

Program Access Service Module

Server Virtual Terminal Service Module

Pseudo-Teletype

User Telnet

NCP Daemon

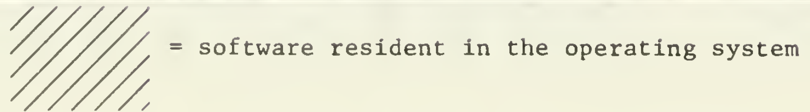
NCP

IMP Device Driver

IMP

Terminal Device Driver(s)

Terminals


= software resident in the operating system

ADDITIONS TO UNIX

General

The Unix operating system will be modified to support HFP operations. Parts of some general-purpose system functions are not needed to support the front-end programs. These functions will be streamlined. Some system functions are not needed at all and will be removed.

Device drivers will be added to Unix to manage the IMP-to-host interface, the 11/70-H6000 interface, and VIP terminals. The Unix terminal handler and numerous other system modules will be modified for front-end experiments.

Two additional facilities must be implemented to support network access through the front end: an inter-process communication facility and a non-blocking I/O facility. These are needed partly because the NCP interface is imbedded in the Unix file system. ARPANET connections are initiated by file OPEN requests. Data are transmitted to the network by file WRITE operations and received from the network by file READ operations. Each of these operations causes the initiating process to block until the operation is complete. When a network READ is executed by a process, the process is blocked until data arrives from the network. Therefore, simultaneous READ's on multiple network connections require multiple processes. To enable a single process to manage several concurrent network operations, the NCP user interface will be modified to allow non-blocking I/O operations. Non-

blocking I/O will be implemented with an inter-process communication facility.

Inter-Process Communication (IPC)

An inter-process communication (IPC) facility will be implemented to effect efficient communication between processes and to provide a convenient mechanism for the implementation of non-blocking I/O. Two types of communications will be used: events and messages.

Events transfer small amounts of control information between processes. Events have a source, a destination, an op-code, 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. Shared data areas are a useful by-product of the message facility.

Each process has an IPC queue where events and notifications of messages are stored until requested.

Details of the inter-process communication facility are described in CAC Technical Memorandum No. 84, "Unix Inter-Process Communication".

Non-Blocking I/O

Non-blocking I/O will be added to the front-end file system. This will enable a single process to perform I/O concurrently on multiple files.

At present, each Unix terminal typically uses three or four processes. As the number of active processes increases, system response degrades rapidly. The rapidity of the degradation appears to be due to the large number of processes resulting from the Unix multiple-process-per-terminal architecture (minimum process overhead = 1K words). The addition of non-blocking I/O will significantly reduce the number of processes typically used by each terminal.

Non-blocking I/O will use the inter-process communication facility. Events will be used to notify user processes of the completion of non-blocking I/O operations. File system software will be modified to generate events at appropriate times:

1. the opening of a file,

2. the arrival of input data,

3. the completion of output operations, and

4. the closing of a file.

When processes receive these events, they are free to execute READ, WRITE, and CLOSE file-system primitives in the usual manner.

There are other non-blocking I/O strategies. The proposed strategy represents a relatively simple modification to the Unix system.

Experience with the HFP is necessary to determine some aspects of front-end operating system requirements. Initially, non-essential system modifications will be kept to a minimum. After HFP test data are available, alternative strategies for non-blocking I/O, process management, and memory management will be investigated.

HFP SOFTWARE MODULES IN THE FRONT END

Introduction

There will be four major HFP software modules in the front end:

1. the Channel Protocol Module,
2. the ARPANET Host-Host Service Module,
3. the Program Access Service Module, and
4. the Server Virtual Terminal Service Module.

Each component will be implemented as a user level program. Each program is structured as a finite state machine accepting input from multiple sources. These inputs may be thought of as messages that request some action and drive the machine from state to state. Each message is associated with a logical communications channel. The message type and current state determine the action and the next state. Most actions result in the transmission of a message to another destination and in the generation of a response indicating the success or failure of the action.

The functions required in the interface between the channel protocol module and the service modules (the service-to-CPM interface) are described in the HFP specifications, CAC Document 219 (ARPANET RFC 710). The service-to-CPM interface employed in the experimental network front end is implemented via the IPC mechanism for efficiency. It is functionally equivalent to that described in the HFP specification. The IPC messages passed between the channel protocol module and the service modules are identical in format and content to HFP Messages. Appendix I includes a description of the relation between the service-to-CPM

interface as described in the HFP specification and the service-to-CPM interface employed in the experimental network front end.

A brief description of each component follows. More detailed descriptions are contained in Appendices I through IV.

Channel Protocol Module (CPM)

Function. The channel protocol module (CPM) will enable programs running in the H6ØØØ to communicate with service modules in the front end. It will implement the HFP channel protocol described in the HFP specifications, CAC Document 219 (ARPANET RFC 71Ø). The CPM will perform several functions.

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

2. It will accept input in the form of HFP Messages from the service modules and multiplex them to the host interface.

3. It will implement the HFP flow control mechanism.

4. It will perform error checking at the channel protocol level.

Structure. The CPM will be implemented as a finite state machine which relays information flowing in two directions:

1. from the host to the service modules and

2. from the service modules to the host.

The CPM will communicate with the host via the 11/7Ø-H6ØØØ interface device driver using the non-blocking I/O mechanism. It will communicate with the service modules via the IPC mechanism.

The CPM will be a user-level program in order to facilitate its testing and evaluation. It may be transformed into a kernel process in order to improve the performance of the front end.

Operation. The CPM will wait for HFP Messages from the host and IPC Messages from the service modules. Messages from both sources will have the form of HFP Commands and Responses. As the CPM receives each message, it will call a routine appropriate to the service and type of message. These routines will perform multiplexing, flow control, and error checking.

ARPANET Host-Host Service (HHS) Module

Function. The ARPANET Host-Host Service (HHS) module will enable programs running in the H6000 to use the ARPANET NCP in the front end. It will implement the ARPANET Host-Host process-to-service protocol described in CAC Technical Memorandum No. 80. The HHS module will perform several functions, using the ARPANET NCP in the front end.

1. It will open and close ARPANET connections to hosts on the network.
2. It will pass data between the H6000 and hosts on the network.
3. It will maintain connection status information.

The HHS module will communicate with programs in the H6000 via the CPM.

Structure. The HHS module will be implemented as a finite state machine which relays information flowing in two directions:

1. from the CPM (and thus from the H6000) to the NCP and
2. from the NCP to the CPM (and thus to the H6000).

The HHS module will communicate with the CPM via the IPC mechanism. It will communicate with the NCP via the non-blocking I/O mechanism.

Operation. The HHS module will wait for IPC messages from the CPM and IPC events from the NCP.

The IPC messages from the CPM will have the form of HFP Commands and Responses. As the HHS module receives each message, it will call a routine appropriate to the message type. These routines will perform Command-specific functions, handle error situations, initiate state transitions, and generate HFP Responses.

IPC events from the NCP will signal the completion of network I/O operations. As the HHS module receives each event, it will call an appropriate routine. These routines will generate the necessary HFP Message sequences.

Program Access Service (PAS) Module

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 will perform several functions using the Unix pseudo-Teletype (PTY) mechanism.

1. It will enable programs on the H6000 to log in to and log out of the Unix system.
2. It will enable programs on the H6000 to run programs under Unix (for example, User Telnet).
3. It will pass data between programs on the H6000 and programs running under Unix.

The PAS module will communicate with programs on the H6000 via the CPM.

Structure. The PAS module will be implemented as a finite state machine which relays information flowing in two directions:

1. from the CPM (and thus from the H6000) to the pseudo-Teletypes, and
2. from the pseudo-Teletypes to the CPM (and thus to the H6000).

The PAS module will communicate with the CPM via the IPC mechanism. It will use the pseudo-Teletypes via the non-blocking I/O mechanism. The PAS module will be a user-level program.

Operation. The PAS module will wait for IPC messages

from the CPM and IPC events from the pseudo-Teletypes.

The IPC messages from the CPM will have the form of HFP Commands and Responses. As the PAS module receives each message, it will call a routine appropriate to the message type. These routines will perform Command-specific functions, handle error situations, initiate state transitions, and generate HFP Responses.

IPC events from the pseudo-Teletypes will signal the completion of pseudo-Teletype I/O operations. As the PAS module receives each event, it will call an appropriate routine. These routines will generate the necessary HFP Message sequences.

ARPANET Server Virtual Terminal Service (SVTS) Module

Function. The ARPANET Server Virtual Terminal Service (SVTS) module will enable programs in the H6000 to be accessed by terminals on the ARPANET. It will implement the ARPANET Server Virtual Terminal process-to-service protocol described in CAC Technical Memorandum No. 82. It will also implement the ARPANET Telnet protocol described in NIC Document No. 15372. The SVTS module will perform several functions, using the ARPANET NCP in the front end.

1. It will open and close ARPANET connections to hosts on the network.
2. It will pass data between the H6000 and hosts on the network, transforming the data in accordance with Telnet protocol.
3. It will maintain connection status information.
4. It will perform Telnet option negotiation.

The SVTS module will communicate with programs in the H6000 via the CPM.

Structure. The SVTS module will be implemented as a finite state machine which relays and transforms information flowing in two directions:

1. from the CPM (and thus from the H6000) to the NCP and
2. from the NCP to the CPM (and thus to the H6000).

The SVTS module will communicate with the CPM via the IPC mechanism. It will communicate with the NCP via the non-blocking I/O mechanism.

Operation. The SVTS module will wait for IPC messages from the CPM and IPC events from the NCP.

The IPC messages from the CPM will have the form of HFP Commands and Responses. As the SVTS module receives each message, it will call a routine appropriate to the message type. These routines will perform Command-specific functions, transform data, handle error situations, initiate state transitions, and generate HFP Responses.

IPC events from the NCP will signal the completion of network I/O operations. As the SVTS module receives each event, it will call an appropriate routine. These routines will generate the necessary HFP Message sequences and perform Telnet option negotiation.

APPENDIX I

CHANNEL PROTOCOL MODULE

Channel Protocol Module (CPM)

Current status. The detailed description of the module which follows is a preliminary version and is subject to change.

Function. The channel protocol module (CPM) will enable programs running in the H6000 to communicate with service modules in the front end. It will implement the HFP channel protocol described in the HFP specifications, CAC Document 219 (ARPANET RFC 710). The CPM will perform several functions.

1. It will de-multiplex HFP Messages arriving from the host interface and pass them to the appropriate service modules in the front end.
2. It will accept input in the form of HFP Messages from the service modules and multiplex them to the host interface.
3. It will implement the HFP flow control mechanism.
4. It will perform error checking at the channel protocol level.

Structure. The CPM will be implemented as a finite state machine which relays information flowing in two directions:

1. from the host to the service modules and
2. from the service modules to the host.

The CPM will communicate with the host via the 11/70-H6000 interface device driver using the non-blocking I/O mechanism. It will communicate with the service modules via the IPC mechanism.

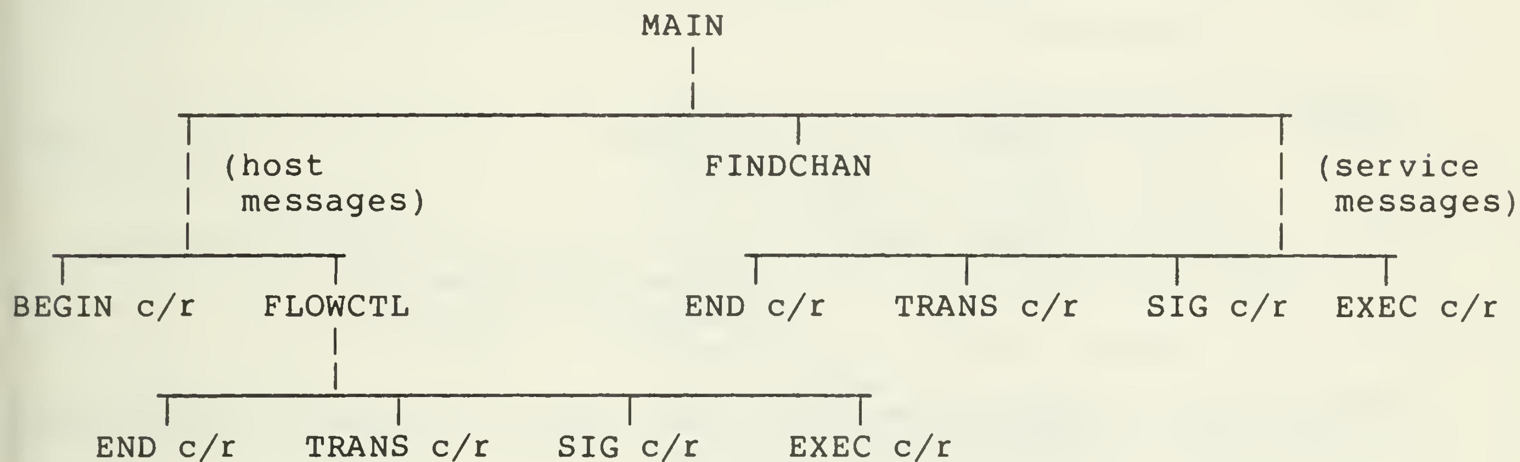
The CPM will be a user-level program in order to facilitate its testing and evaluation. It may be transformed into a

kernel process in order to improve the performance of the front end.

Operation. The CPM will wait for HFP Messages from the host and IPC Messages from the service modules. Messages from both sources will have the form of HFP Commands and Responses. As the CPM receives each message, it will call a routine appropriate to the service and type of message. These routines will perform multiplexing, flow control, and error checking.

Software Architecture

The following table illustrates the CPM procedure call hierarchy. The "c/r" following procedure names indicates the existence of both a command routine and a response routine. For example, "BEGIN c/r" indicates that routines exist at that level to process both an HFP BEGIN Command and an HFP BEGIN Response.



State Transition Table

The following table depicts CPM logical channel states. The STATE column indicates 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	Initialize channel data structure Pass Command to service or host.	PENDING
PENDING	BEGIN Response Status = Ø	Pass Command to host or service.	ESTABLISHED
	BEGIN Response Status not Ø	Pass Command to host or service.	NULL
	END Command	Pass Command to destination.	TERMINATING
ESTABLISHED	END Command w/drain	Wait for queued TRANS. to drain.	DRAINING
	END Command w/o drain	Cleanup channel data structure pass command to destination.	TERMINATING
	SIGNAL Command	Take appropriate signal action.	ESTABLISHED
	All other Commands	Update flow control and acknowledge information.	ESTABLISHED
DRAINING	Transmits drained	Send END Command	TERMINATING
TERMINATING	END Response	Release channel data structure.	NULL
	BEGIN Response	Ignore	TERMINATING

CPM Data Structures

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

link(16)	-pointer to the next channel element Null indicates end of list
group(16)	-this channel's group number
member(16)	-this channel's member number
state(8)	-channel state (see states below)
service number(8)	-number of the service for this channel
service IPC(8)	-IPC address of the service for this channel
hiscredit(8)	-current credit given by host
his_seq(8)	-sequence number of last TRANSMIT received from host
mycredit(8)	-amount of credit given by front end
myseq(8)	-last TRANSMIT acknowledged by host
currseq(8)	-sequence number of last TRANSMIT sent to host
trans-q(16)	-queue head of TRANSMITS waiting to be sent
retran-q(16)	-TRANSMITS waiting to be acknowledged

The following are state variable values:

NULL	0
PEND	1
ESTAB	2
DRAIN	3
TERM	4

Service-to-CPM Interface

The HFP Specification describes 12 service-to-CPM interface primitives. These primitives identify the operating system functions needed to interface a service module to the CPM.

With the exception of S_ACCEPT, S_ACK, and S_IDENTIFY, each of these primitives requests the CPM to generate an HFP Message. For example, the S_TERMINATE primitive requests the generation of an HFP END Command, and the S_SEND primitive requests the generation of an HFP TRANSMIT Command. In the ENFE, the service modules will generate these HFP Messages. They will be conveyed to the CPM (and thus to the H6000) via IPC messages. The S_ACCEPT primitive corresponds to the IPC system primitive that reads an IPC message or event. The acknowledgement function that would be provided by the S_ACK primitive will be provided by passing HFP TRANSMIT Responses between the service modules and the CPM. The S_IDENTIFY primitive will be implemented via a service-to-CPM IPC event.

Program Logic

HFP Response status codes and their names are defined in the HFP Specification. In the software logic descriptions, each status code is represented by its name followed by its value in parentheses.

MAIN

MAIN links the channel data structures into a free list,

procures any required IPC resources, calls INITIATE to begin communications with the H6000, and falls into a loop waiting for messages. As the messages arrive, they are multiplexed via the command/response routines.

Logic

```
Link channel data structures into a free list.
Procure required IPC resources.
call INITIATE to bring up the H6000 link.
loop
        wait for IPC message
        based on message source and type
                call common code routine
        based on message source and type
                call command/response routine.
```

INITIATE

INITIATE "brings up" the communications link with the H6000. INITIATE sends an END Command with channel group and member equal to zero and waits for an END Response. When the END Response arrives, INITIATE waits for a BEGIN Command addressed to the HFP Maintenance Service. When the BEGIN Command arrives, INITIATE sends a BEGIN Response and marks the host as "alive". All other messages from the host are ignored during this period.

Logic

```
        send END Command w/group & member = 0
        wait for END Response w/group & member = 0

        wait for BEGIN Command for HFP
          Maintenance Service
        send BEGIN Response

        mark host "alive"
```

BEGIN Command (from host)

A BEGIN Command requests the construction of a logical HFP channel. The BEGIN routine must get a channel data structure, fill it in, and pass the Command on to the appropriate service.

Logic

```
check service valid?
no:
        return BEGIN Response
        w/status = SERV_NOT_FOUND(33)
        return

Call FINDCHAN
FINDCHAN return non-zero?
(channel already exists)
yes:
        return BEGIN Response
        w/status = CHAN_IN_USE(32)
        return

GETCHAN return zero?
yes:
        return BEGIN Response
        w/status = NO_RESOURCES(34)
        return

fill in
        service#
        service IPC address
        hiscredit

clear myseq, his_seq, currseq
set state <- PEND
set mycredit <- 8

pass BEGIN to service
```

BEGIN Response (from service)

A BEGIN Response is generated to indicate the success or failure of a BEGIN Command.

Logic

```
FINDCHAN return zero?  (Channel doesn't exist)
yes:
        log error and discard
        return

state = PEND?
no:
        log and discard (not necessarily an error)
        return

status not zero?
yes:
        call FREECHAN
no:
        state <- ESTAB
        credit <- mycredit

call HOSTSEND
```

TRANSMIT (from host)

A TRANSMIT Command is generated when the host wants to send data over a logical channel to a front-end service.

Logic

```
Call FINDCHAN

FINDCHAN return zero?

yes:
        return TRANSMIT Response
        w/status <- CHAN_NOT_FOUND(1)
        return

channel state = ESTAB?
no:
        return TRANSMIT Response
        w/status <- ILLEGAL_STATE(2)
        return

message myseq = (his_seq+1)mod16?
no:     (TRANSMIT out of order)

        return TRANSMIT Response
        w/status <- OUT_OF_SEQ(35)
```

```
        return

increment his_seq mod 16
decrement mycredit

call FLOWCTL

pass TRANSMIT Command to service
```

TRANSMIT Response (from host)

A TRANSMIT Response is generated by the host to update flow control information and signal error situations.

Logic

```
FINDCHAN return zero?
yes:    log error and discard
        return

status not zero?
yes:    log error

call FLOWCTL

status = SERVICE ERROR?
        pass TRANSMIT Response on to service
```

TRANSMIT (from service)

A TRANSMIT is generated when data becomes available for the associated host process.

Logic

```
FINDCHAN return zero?
yes:    log error and discard?
        return

increment currseq mod 16
copy currseq to message myseq

(struct myseq + hiscredit)mod 16 >= currseq?
yes:
        call FLOWSEND
no:
```

```
        queue TRANSMIT to be sent by FLOWCTL later
        number of queued TRANSMITS = MAX_QUEUABLE?
        yes:
                send XOFF event to service
```

<u>TRANSMIT Response (from service)</u>

A TRANSMIT Response is generated by a service when it finishes sending TRANSMIT data over a network connection.

<u>Logic</u>

```
FINDCHAN return zero?
yes:
        log error and discard
        return

increment mycredit

call FLOWSEND
```

<u>END (from host with or without flush)</u>

An END Command is generated by the host when it wishes to terminate a logical channel. Only messages in transit to the host are queued here. Because the host has requested the destruction of the logical channel, queued TRANSMITS waiting for flow control permission before they would be sent to the host are released to the system and not sent to the host.

<u>Logic</u>

```
FINDCHAN return 0?
yes:
        return END Response w/status <- CHAN_NOT_FOUND(1)
        return

state  <-  TERM

release any queued TRANSMIT Commands
release any unacknowledged TRANSMIT Commands
```

Pass END to service

END (from service)

An END Command is generated by the services when they wish to terminate a logical channel.

Logic

```
FINDCHAN return zero?
yes:
        return END Response
        w/status <- CHAN_NOT_FOUND(1)
        return

state  <-  TERM

flush requested?
yes:    release any queued TRANSMIT Commands
no:     any queued TRANSMIT Commands?
        yes:
                queue END at end of TRANSMIT queue. It
                will go out after the last TRANSMIT.
                return.

call FLOWSEND
```

END Response (from service or host)

An END Response is generated when either the service or the host wishes to acknowledge the closing of a channel.

Logic

```
FINDCHAN return zero?
yes:    log and discard
        return

release any queued TRANSMIT Commands
release any unacknowledged TRANSMIT Commands

call FREECHAN

call HOSTSEND or pass to service
```

SIGNAL (from host)

A SIGNAL Command is generated when the host wishes to change the status of either the logical communication channel, or the service at the opposite end of the channel.

Logic

```
FINDCHAN return zero?
yes:
        return SIGNAL Response
        w/status <- CHAN_NOT_FOUND(1)
        return

call FLOWCTL

CONTROL = 0?
yes:
        (return flow control information)
        return SIGNAL Response w/status <- SUCCESS(0)
        return

bit 2 of CONTROL on?
yes:
        flush any queued TRANSMITS

pass SIGNAL Command to service.
```

SIGNAL Response (from host)

A SIGNAL Response is generated by the host when it wishes to acknowledge a SIGNAL Command.

Logic

```
FINDCHAN return zero?
yes:    log error and discard
        return

call FLOWCTL

pass SIGNAL Response to service
```

SIGNAL (from service)

A SIGNAL Command is generated by the service when it wishes to modify the state of a logical channel or the process at the opposite end of the channel.

Logic

```
FINDCHAN return zero?
yes:    log error and discard
        return

bit 1 of CONTROL on?
yes:
        flush any queued TRANSMIT Commands

bit 3 of CONTROL on?
yes:
        any TRANSMITS waiting to be sent?
        yes:
                queue SIGNAL at end of queue.  It will
                go out after the last TRANSMIT goes out.
                return

call HOSTSEND
```

SIGNAL Response (from service)

A SIGNAL Response is generated by a service when it wishes to acknowledge a SIGNAL Command.

Logic

```
FINDCHAN return zero?
yes:    log error and discard
        return

call FLOWSEND
```

EXEC (from host)

An EXECUTE Command is generated by a process when it

wants to request a special function from the service.

Logic

```
FINDCHAN return zero?
yes:
        return EXECUTE Response
        w/status <- CHAN_NOT_FOUND(1)
        return

call FLOWCTL

pass EXECUTE on to service.
```

EXEC Response (from host)

An EXECUTE Response is generated by a process when it wishes to acknowledge an EXECUTE Command.

Logic

```
FINDCHAN return zero?
yes:    log error and discard
        return

call FLOWCTL

status = CHAN_NOT_FOUND(1)?
yes:    log error
        return

pass EXECUTE Response on to service
```

EXEC and EXEC Response (from service)

An EXECUTE Command is generated by a service when it wishes to request a special action from the process at the other end of the logical channel.

An EXECUTE Response is generated by a service when it wishes to acknowledge an EXECUTE Command.

Logic

```
FINDCHAN return zero?
yes:    log error and discard
        return

call FLOWSEND
```

FINDCHAN

FINDCHAN is called by the various command/response routines to locate a specific channel data structure.

Logic

```
Set up search thru channel data structure list

while entries in list
        current channel group = list channel group
                         and
        current channel member = list channel member
        yes:
                return  address of channel data structure

        get next channel list member

(all entries searched!)
return zero
```

GETCHAN

GETCHAN is called by the command/response routines to get an unused channel data structure from the channel free list.

Logic

```
any entries in free list?
no:     return zero

delink entry from list
copy current channel group and member into data structure

search active channel list for correct entry point
link new entry into active channel list
return address of channel data structure
```

FREECHAN

FREECHAN is called by the command/response routines to release a channel data structure.

Logic

search active channel list for channel entry

delink it from active list

link structure into free channel list

FLOWCTL

FLOWCTL is called by command/response routines that receive HFP messages from the host. Its principal function is to update channel acknowledgement and flow control variables. If any TRANSMIT Commands have been acknowledged, the associated IPC message segments may be released to the system-free pool. If the number of outstanding TRANSMITS is less than eight and TRANSMITS are queued to be sent to the host, send as many TRANSMITS as possible.

Logic

```
copy credit to hiscredit (save received host credit)
copy yourseq to myseq (save last acknowledged TRANSMIT)

release any acknowledged TRANSMIT Commands

number of queued TRANSMITS = MAXQUEUEABLE?
yes:
        set XONFLAG

while any queued TRANSMIT Commands

        map first queued TRANSMIT
        (structure myseq + hiscredit) mod 16
                    >=
                TRANSMIT myseq?
```

```
yes:
        delink TRANSMIT
        copy his_seq to yourseq
        copy mycredit to credit
        call HOSTSEND
        unmap queued TRANSMIT
no:
        XONFLAG lit?
        yes:
                number of queued TRANSMITS
                                     <
                                MIN_QUEUED
                yes:
                            send XON event to service
        return
```

FLOWSEND

FLOWSEND is called by the command/response routines to copy the current flow control and acknowledgement information into the header of HFP Messages going to the host.

Logic

```
copy currseq to myseq
copy mycredit to credit
copy his_seq to yourseq
call HOSTSEND
```

HOSTSEND

HOSTSEND is called by many command/response routines to pass messages to the output side of the H6000-11/70 hardware device driver for ultimate delivery to the H6000.

APPENDIX II

ARPANET HOST-HOST SERVICE MODULE

ARPANET Host-Host Service (HHS) Module

Current status. The detailed description of the module which follows is a preliminary version and is subject to change.

Function. The ARPANET Host-Host Service (HHS) module will enable programs running in the H6000 to use the ARPANET NCP in the front end. It will implement the ARPANET Host-Host process-to-service protocol described in CAC Technical Memorandum No. 80. The HHS module will perform several functions, using the ARPANET NCP in the front end.

1. It will open and close ARPANET connections to hosts on the network.
2. It will pass data between the H6000 and hosts on the network.
3. It will maintain connection status information.

The HHS module will communicate with programs in the H6000 via the CPM.

Structure. The HHS module will be implemented as a finite state machine which relays information flowing in two directions:

1. from the CPM (and thus from the H6000) to the NCP and
2. from the NCP to the CPM (and thus to the H6000).

The HHS module will communicate with the CPM via the IPC mechanism. It will communicate with the NCP via the non-blocking I/O mechanism.

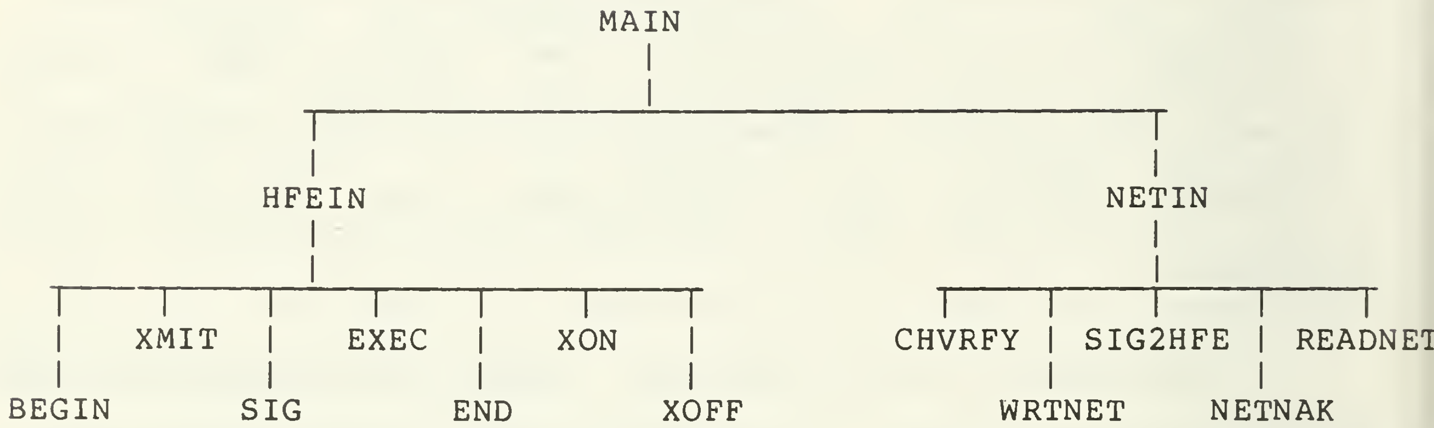
Operation. The HHS module will wait for IPC messages from the CPM and IPC events from the NCP.

The IPC messages from the CPM will have the form of HFP Commands and Responses. As the HHS module receives each message, it will call a routine appropriate to the message type. These routines will perform Command-specific functions, handle error situations, initiate state transitions, and generate HFP Responses.

IPC events from the NCP will signal the completion of network I/O operations. As the HHS module receives each event, it will call an appropriate routine. These routines will generate the necessary HFP Message sequences.

Software Architecture

The procedure calling structure of HHS is relatively simple.



State Transition Table

The following table depicts HHS logical channel states, actions, and state transitions.

STATE	EVENT(input)	ACTION(output)	NEXT STATE
NULL	BEGIN Command	open net channel	PEND
PEND	net open success	notify host	ESTAB
	net open fail	notify host	NULL
	END Command	close net channel free resources	NULL
	SIGNAL Command	error	PEND
	EXECUTE Command	error	PEND
ESTAB	net error	notify user free resources	NULL
	TRANSMIT (partial)	data to net	BUSY
	TRANSMIT (full)	data to net	ESTAB
	data from net	send to host	ESTAB
	END	close channel free resources	NULL
	SIGNAL	do it	ESTAB
	EXECUTE	do it	ESTAB
BUSY	neterror	notify host flush buffers free resources	NULL
	TRANSMIT	buffer data	BUSY
	data from net	send to host	BUSY
	data to net gone		ESTAB
	END	let data drain	TERM
	SIGNAL	do it	BUSY
	EXECUTE	do it	BUSY
TERM	data drained	notify host	NULL
	SIGNAL	do it	TERM
	EXECUTE	do it	TERM
	net error	notify host	NULL

Channel Data Structure

Channel information is kept in a singly linked list of data structures. At any one time, each of these structures is linked into either an active or free list. The following fields are necessary to hold channel information (the numbers in parens are the field widths in bits):

```
link    (16)     - address of new channel list element
group   (16)     - channel group ID
member  (16)     - channel member ID
state   (8)      - channel state
flag    (8)      - channel flag bits
size    (16)     - number bytes waiting to be read from NCP
currseg(16)      - ID of TRANSMIT being sent to the NCP
fid     (8)      - NCP file ID for this channel
sindex  (8)      - next queued TRANSMIT to send
segs[N*8]        - list of HFP Messages being output
```

Open structure

When HHS performs a network open on a given channel, it must pass parameters with the request. This is done with a structure containing the following fields:

o op(8) - used internally by the NCP.

o type(8) - connection type:

```
bit 0:  init/listen
bit 1:  icp/direct
bit 2:  duplex/simplex
```

o id(16) - internal to NCP.

o lskt(16) - host's local socket for this connection.

o fskt(32) - socket in foreign host to which connection is to be attempted.

o frnhost(8) - foreign host identifier.

o bsize(8) - sizes of bytes used on the connection.

o nomall(16) - nominal allocation in bytes.

o timeo(16) - number of seconds to wait before timing out an attempt.

o relid(16) - internal to NCP.

Program Logic

HFP Response status codes and their names are defined in the HFP Specification. In the software logic descriptions, each status code is represented by its name followed by its value in parentheses.

MAIN

The MAIN subroutine provides the driving loop for the program. MAIN waits for IPC events and messages from the CPM and NCP. As these IPC communications are received, the event source and type are used to call lower level routines in the hierarchy. Each of these routines implements state transition operations.

MAIN sets up the necessary resources and falls into a loop where it waits for an event to occur. For security purposes, any event that does not originate from the CPM or the NCP is logged as an error and ignored.

Logic

```
Link Channel structures into the free list
Initialize IPC variables
loop:
        wait for IPC event

        event source = CPM?
        yes:
                call HFEIN

                return

        event source = NET?
        yes:
                call NETIN

                return
```

```
( if we got here, event is from bad source )
log event as an error and discard
```

HFEIN

HFEIN obtains the IPC messages from the CPM and attempts to determine if the Command is implemented. If the message is an HFP Response, it is discarded, since Responses are not currently used.

Logic

```
Get IPC message from CPM
error getting message?
yes:
        log error and return

message an HFP Response?
yes:
        discard
        return

implemented Command?
no:
        send Command Response
        w/status <- COMM_NOT_IMPLEMENTED(3)

        return

call FINDCHAN

call Command procedure
```

Following are the five routines immediately subordinate to HFEIN that handle the HFP Commands from the host.

BEGIN

A BEGIN Command is received when a host process wants an

ARPANET connection opened. The Command TEXT contains parameters for that open request.

Logic

```
Channel structure found?
yes:
        return BEGIN Response
        w/status <- ILLEGAL_STATE(2)
        return

fill in
        connection type
        foreign host
        foreign socket
        local socket
        nominal allocation
        timeout
        bytesize

initiate network file OPEN request

error in OPEN request?
yes:
        send BEGIN Response
        w/status <- ACTION_FAILED(66)

        return

call MAKCHAN
MAKCHAN fail?
yes:
        send BEGIN Response
        w/status <- NO_RESOURCES(34)

        return

set channel state <- PEND
```

XMIT

XMIT transfers data to the NCP when it arrives from the CPM. A channel is considered busy when a previous non-blocking I/O request could not accept all data, or HHS is awaiting

confirmation of transfer to the foreign host.

Logic

```
Channel found?
no:
        return TRANSMIT Response
                w/status <- CHAN_NOT_FOUND(1)
        return

channel state not <- ESTAB or BUSY?
yes:
        return TRANSMIT Response
                w/status <- ILLEGAL_STATE(2)
        return

set up transfer

channel state = BUSY?
yes:
        queue TRANSMIT to be sent to network
no:
        write TRANSMIT data to network
        error in network write?
        yes:
                return TRANSMIT Response
                w/status <- ACTION_FAILED(66)

                return

                call KILLCHAN
                return

        all bytes transferred?
        yes:
                return TRANSMIT Response
                w/status <- SUCCESS(Ø)

                return
        no:
                set channel state <- BUSY
                save TRANSMIT information
```

SIGNAL

As data is received from the network, it is passed on to the CPM for transmission to the host. Therefore, signals asking to

flush data in transit to the host are ignored here and handled by the CPM.

Logic

```
bit 1 of CONTROL on?
yes:
        release any queued segments

bit 2 of CONTROL on?
yes:
        ignore.  data going to the host is not
        buffered here.


bit 3 of CONTROL on?
no:
        bit 1 of CONTROL on?
        yes:
                send ARPA INS for this connection.

        send SIGNAL Response w/status <- SUCCESS(0)

yes:
        bit 1 of CONTROL on?
        yes:
                mark signal so that INS is sent
                later.

        room to queue signal?
        no:
                send SIGNAL Response
                w/status <- ACTION_FAILED(66)

                return
        yes:
                queue SIGNAL Command
                SIGNAL Response will be returned
                when all TRANSMITS have been
                sent to the network.
```

EXEC

At this time, the Network change-allocation mechanism is not available. Any requests for this feature will be given an 'unimplemented' error.

Logic

```
find channel data structure?
no:
        return EXECUTE Response
                w/status <- CHAN_NOT_FOUND(1)
        return

request channel state information?
yes:
        get network status information from system

        copy information into an EXECUTE Response

        send EXECUTE Response w/status <- SUCCESS(0)

        return

(option wasnt found, tell host)
send EXECUTE Response
w/status <- NOT_IMPLEMENTED(4)
```

END

If there is no queued output when an END is received, the channel is immediately closed; otherwise, its state is set to TERM and the data is allowed to drain to the network. The END Response is successful as long as the channel exists.

Logic

```
channel data structure found?
no:
        return END Response
                w/status <- CHAN_NOT_FOUND(1)
        return

flush requested?
yes:
        flush queued TRANSMIT Commands

channel busy?
no:
        call KILLCHAN
        return END Response w/status <- SUCCESS(0)
yes:
        set channel state <- TERM
```

CPM-SERVICE Flow Control

XON and XOFF events are used by the CPM to flow control TRANSMIT Commands from the services. When a service receives an XOFF event from the CPM, it should not send TRANSMIT Commands until an XON event for the associated channel arrives.

XON

XON is called in response to an XON event received from the CPM.

Logic

```
clear XOFF bit in channel state
data to read?
yes:
        call READNET
```

XOFF

XOFF is called in response to receiving an XOFF event from the CPM.

Logic

```
set XOFF bit in channel state
clear channel size
```

Network Section of HHS

NETIN

When an event from the NCP comes in, NETIN is called to determine the next state. If the file descriptor cannot be found in the channel list, an error is logged and the event is discarded. The event source determines which subordinate routine is called.

Logic

```
find network file ID?
no:
        log error and return.

known event source?
no:
        log error and return.

call event routine
```

CHVRFY

CHVRFY is called when a network file OPEN request completes. It sends a BEGIN Response previously saved by the BEGIN routine.

Logic

```
OPEN request successful?
yes:
        channel state <- ESTAB

        return BEGIN Response
                w/status <- SUCCESS(0)
no:
        call KILLCHAN (don't send END Command)
```

```
send BEGIN Response
        w/status <- ACTION_FAILED(66)
```

READNET

NETIN or CHVRFY calls READNET when data is available on a channel.

Logic

```
data available?
yes:
        call BLDMSG (to get an IPC data segment)

        set up network read

        read data into IPC segment

        call SNDMSG to transfer data to host.
no:
        (channel is dead)
        call KILLCHAN
```

WRTNET

When a write to a channel completes, WRTNET is called. Terminating channels are handled here.

Logic

```
channel dead?
yes:
        call KILLCHAN
        return

get current segment

more data in current TRANSMIT Command
to send to the network?
no:
        call NEWSEG

more TRANSMITS to send?
```

```
        no:
                channel state = TERM?
                yes:
                        call KILLCHAN

                return

        set up network write

        write data to network

        error in network write?
        yes:
                call KILLCHAN
                return

        save bytes remaining to be written
```

NEWSEG

NEWSEG is called when data from a TRANSMIT Command has been sent to the network and its successful transfer has been confirmed. Wait-for-drain SIGNAL Commands are handled here.

Logic

```
        return TRANSMIT Response w/status <- SUCCESS(0)

        loop:
        more TRANSMITS queued to send?
        yes:
                get first queue element
                element a SIGNAL Command?
                yes:
                        send ARPA INS
                        return SIGNAL Response
                                w/status <- SUCCESS(0)
```

NETNAK

When a network transmission to a foreign host fails, NETNAK is called to resend the data.

Logic

```
Set up network write
write as many bytes as possible to network
error in writing?
yes:
        call KILLCHAN
        return

save write information
```

SIG2HFE

SIG2HFE is called when an event from the NCP is received stating that an ARPA network INS has been received.

Logic

```
find network file ID?
no:
        log error and discard event
        return

send SIGNAL Command to host
        w/control specifying interrupt for
        correct data flow direction
```

Auxiliary Routines

MAKCHAN

MAKCHAN builds a new channel node and inserts it into the channel list, returning a pointer to it.

Logic

```
any free channel data structures?
no:
        log error and return zero

delink channel structure from channel free list

link structure into active list

copy channel group and member into structure field

initialize rest of channel data structure

return address of structure
```

KILLCHAN

This routine destroys a channel and deallocates all the resources associated with it.

Logic

```
free any queued IPC segments

send END Command?
yes:
        send END Command without flush

delink channel data structure from active list

link channel data structure into channel free list
```

APPENDIX III

PROGRAM ACCESS SERVICE MODULE

Program Access Service (PAS) Module

Current status. The detailed description of the module which follows is a preliminary version and is subject to change.

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 will perform several functions using the Unix pseudo-Teletype (PTY) mechanism.

1. It will enable programs on the H6000 to log in to and log out of the Unix system.
2. It will enable programs on the H6000 to run programs under Unix (for example, User Telnet).
3. It will pass data between programs on the H6000 and programs running under Unix.

The PAS module will communicate with programs on the H6000 via the CPM.

Structure. The PAS module will be implemented as a finite state machine which relays information flowing in two directions:

1. from the CPM (and thus from the H6000) to the pseudo-Teletypes, and
2. from the pseudo-Teletypes to the CPM (and thus to the H6000).

The PAS module will communicate with the CPM via the IPC mechanism. It will use the pseudo-Teletypes via the non-blocking I/O

mechanism. The PAS module will be a user-level program.

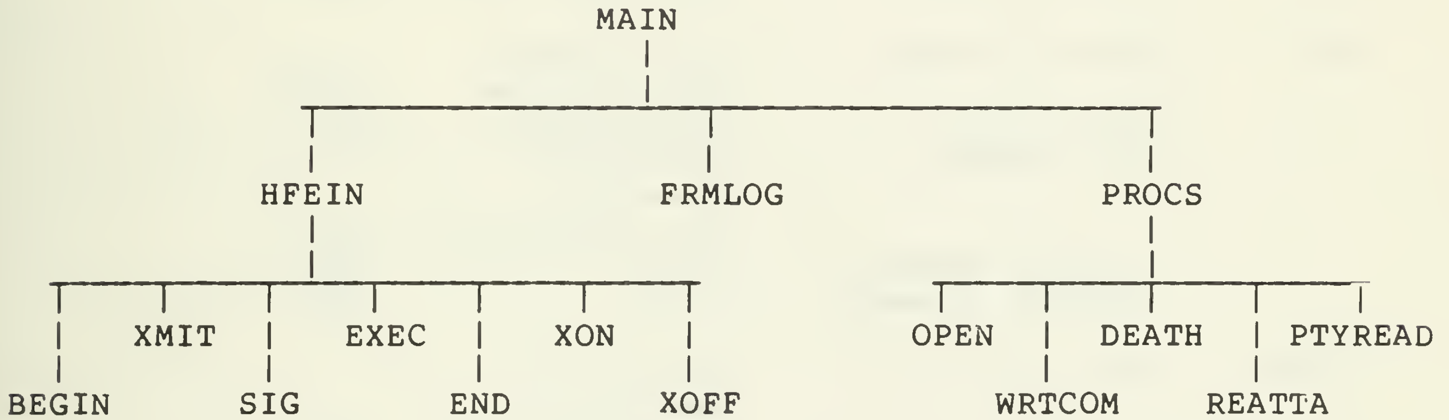
<u>Operation</u>. The PAS module will wait for IPC messages from the CPM and IPC events from the pseudo-Teletypes.

The IPC messages from the CPM will have the form of HFP Commands and Responses. As the PAS module receives each message, it will call a routine appropriate to the message type. These routines will perform Command-specific functions, handle error situations, initiate state transitions, and generate HFP Responses.

IPC events from the pseudo-Teletypes will signal the completion of pseudo-Teletype I/O operations. As the PAS module receives each event, it will call an appropriate routine. These routines will generate the necessary HFP Message sequences.

Software Architecture

The call structure, shown below, is a tree with three branches. The right branch communicates with front-end processes. The center branch handles login responses. The left branch communicates with the H6000 through the CPM.



State Transition Table

The following table defines states, actions, and state transitions.

STATE	EVENT(input)	ACTION(output)	NEXT STATE
NULL	BEGIN Command	send login msg	
		open PTY	PEND
PEND	login success	notify Host	
		fork process	ESTAB
	login fail	notify Host	NULL
	END Command	send kill	
		close PTY	
		free resources	NULL
	SIGNAL Command	error	PEND
	EXECUTE Command	error	PEND
ESTAB	PTY error	notify Host	
		send kill	
		free resources	NULL
	TRANSMIT (partial)	data to PTY	BUSY
	TRANSMIT (full)	data to PTY	ESTAB
	data from PTY	send to Host	ESTAB
	END	close PTY	
		send kill	
		free resources	NULL
	SIGNAL	do it	ESTAB
	EXECUTE	do it	ESTAB
	child death	notify Host	
		free resources	
		close PTY	NULL
BUSY	PTY error	notify Host	
		flush queue	
		send kill	
		close PTY	
		free resources	NULL
	TRANSMIT	buffer data	BUSY
	data from PTY	send to Host	BUSY
	data to PTY gone		ESTAB
	END	let data drain	TERM
	SIGNAL	do it	BUSY
	EXECUTE	do it	BUSY
	child death	close PTY	
		free resources	
		notify Host	NULL
TERM	data drained	notify Host	

	send kill	NULL
SIGNAL	do it	TERM
EXECUTE	do it	TERM
PTY error	notify Host	NULL
child death	notify Host	
	free resources	
	close PTY	NULL

Channel list

Channel information is kept in a linked list of structures. The following fields hold channel information (numbers in parens are the number of bits in the field):

- link (16) - address of new channel list element
- group (16) - channel group ID
- member (16) - channel member ID
- state (8) - channel state
- flag (8) - channel flag bits
- size (16) - number bytes waiting to be read from NCP
- currseg (16) - ID of TRANSMIT being sent to the NCP
- fid (8) - NCP file ID for this channel
- sindex (8) - next queued TRANSMIT to send
- segs[N*8] - list of HFP Messages being output

PTY Table

The PAS module communicates with front-end processes by means of pseudo-Teletypes (PTY). These are software terminals which appear to the process as a hardware terminal, thus enabling one process to control another interactively.

A table of suffixes of known PTY's is kept to speed the search for an available one. The two ends of the pseudo-Teletype are known by different names: PTY<n> (the master), and TTY<m> (the slave). Two characters for each entry are necessary.

Each table entry has the following form:

master(8) - The suffix of a PTY; the end which acts as the keyboard and printer of a terminal.

slave(8) - The suffix of a TTY; the slave end which is used by a program as a normal terminal.

Program Logic

HFP Response status codes and their names are defined in the HFP Specification. In the software logic descriptions, each status code is represented by its name followed by its value in parentheses.

MAIN

MAIN provides the driving loop for the service. MAIN procures IPC resources, links channel data structures into a free list, and falls into a loop. MAIN loops waiting for an event. When an event arrives, MAIN determines whether the the source is the CPM or the pseudo-Teletype software. Based on the event source, HFEIN is called (event from CPM) or PROCS is called (event from pseudo-Teletype software).

Logic

```
Link channel structures into the free list
Initialize IPC variables
Loop:
        wait for IPC event

        event source = CPM?
        yes:
                call HFEIN
                return

        event source = login service?
        yes:
                call FRMLOG
                return

        event source = front-end process?
        yes:
                call PROCS
                return

        ( if we got here, event is from bad source )
        log event as error and discard
```

HFEIN

HFEIN reads the message from the CPM. If the message is an HFP Response it is not processed. The service does not use Responses in its present form. If the Command is implemented, the Command procedure is called.

Logic

```
Get IPC message from CPM
error getting message?
yes:
        log error and return

message an HFP Response?
yes:
        discard
        return

legal Command?
no:
        send Command Response
        w/status <- COMM_NOT_IMPLEMENTED(3)

        return

call FINDCHAN

call Command Procedure
```

Following are routines immediately subordinate to HFEIN. These are the handlers of the HFP Commands.

BEGIN

A BEGIN Command is sent when a host process requests the execution of a front-end program.

Logic

```
Channel data structure found?
yes:
```

```
                return BEGIN Response
                w/status <- CHAN_IN_USE(32)

                return

        Get a message segment

        Can't get it?
        yes:
                return BEGIN Response
                w/status <- ACTION_FAILED(66)

                return

        Copy security field to IPC message

        Send it to login service

        error?
        yes:
                return BEGIN Response
                w/status <- ACTION_FAILED(66)

                return

        Call MAKCHAN

        MAKCHAN fail?
        yes:
                return BEGIN Response
                w/status <- NO_RESOURCES(34)

                return

        set channel state <- PEND
```

END

An END Command is received when the host process wants to terminate communications.

Logic

```
        Channel data structure found?
        no:
                return END Response
                w/status <- CHAN_NOT_FOUND(1)

                return
```

```
flush requested?
yes:
        flush queued TRANSMITS

channel BUSY?
no:
        call KILLCHAN

        return END Response
        w/status <- SUCCESS(0)
yes:
        set channel state <- TERM
```

EXEC

At this time, no functions have been assigned to the EXECUTE Command. An "option not implemented" Response will be sent back to the sender with no other action taken.

Logic

```
Return EXECUTE Response
w/status <- OPTN_NOT_IMPLEMENTED(4)
```

XMIT

XMIT is called when a TRANSMIT Command is received from the CPM.

Logic

```
Channel found?
no:
        return TRANSMIT Response
        w/status <- CHAN_NOT_FOUND(1)

        return

Channel state not ESTAB or BUSY?
yes:
        return TRANSMIT Response
        w/status <- ILLEGAL_STATE(2)

        return
```

```
set up transfer

Channel busy?
yes:
        queue TRANSMIT to be sent to PTY
        return
no:
        write TRANSMIT data to PTY
        error in write?
        yes:
                return TRANSMIT Response
                w/status <- ACTION_FAILED(66)

                call KILLCHAN
                return


all bytes transferred?
yes:
        return TRANSMIT Response
        w/status <- SUCCESS(0)
no:
        set channel state BUSY
        save TRANSMIT information
```

<u>SIG</u>

SIG is called when a SIGNAL Command is received from the CPM.

<u>Logic</u>

```
Default case:
        (Command not implemented)

        return SIGNAL Response
        w/status <- NOT_IMPLEMENTED(4)

        return

KILL:
        return SIGNAL Response
        w/status <- SUCCESS(0)

        call KILLCHAN
        return

INTERRUPT:
        send INTERRUPT to process
```

```
QUIT:
        send QUIT to process

return SIGNAL Response
w/status &lt;- SUCCESS(0)
```

CPM-Service Flow Control

XON and XOFF events are used by the CPM to flow control TRANSMIT Commands from the services. When a service receives an XOFF event from the CPM, it should not send any TRANSMIT Commands until an XON event for the associated channel arrives.

XON

XON is called in response to an XON event received from the CPM.

Logic

```
Clear XOFF bit in channel state
data to read?
yes:
        call PTYREAD
```

XOFF

XOFF is called in response to receiving an XOFF event from the CPM.

Logic

```
Set XOFF bit in channel state
clear channel size
```

Login Handler

FRMLOG

A message from login for this channel tells whether or not the security field sent in a BEGIN is valid.

If the user is correctly logged, a PTY is opened for the channel.

Logic

```
Login failure?
yes:
        send BEGIN Response
        w/status <- ACTION_FAILED(66)

        free channel data structure
        return

Find a free PTY

Cannot find one?
yes:
        send BEGIN Response
        w/status <- NO_RESOURCES(34)

        free channel data structure
        return

Execute a non-blocking OPEN on the PTY
error?
yes:
        send BEGIN Response
        w/status <- NO_RESOURCES(34)

        free channel data structure
        return
```

Process Section of PAS

The process side of PAS handles write completion events, child deaths, read events, PTY reattaches, and PTY open completions. The PTY reattaches are made possible by a special mechanism that allows open files to be passed between processes.

PROCS

PROCS calls FINDFID to seek the channel node. If it exists, the event source is validated as a system source and the event-specific routine is invoked; otherwise the event is logged and discarded.

Logic

```
Call FINDFID

channel found?
no:
        log error and return

event source legal?
yes:
        call event specific routine

        return

(if we get here, it's an illegal source)
log as error and return
```

WRTCOM

If all of the data in a TRANSMIT Command could not be accepted by the kernel, an event will arrive when more can be accepted. That type of event is handled here. Data queued will be

written until no more is accepted or the queue is empty.

When a channel's queue empties and it is in state TERM, the PTY will be closed, a kill will be sent to the process, and the state will be set to NULL.

Logic

```
Did the event return a channel error?
yes:
        call KILLCHAN
        return

set up write

WRITE as much as possible.
error?
yes:
        call KILLCHAN
        return

Did all the data go out?
yes:
        return TRANSMIT Response
        w/status <- SUCCESS(Ø)
no:
        save what didn't transfer
        set channel state to BUSY
        return

Loop:
        Is all data gone?
        yes:
                Is state <- TERM?
                yes:
                        return END Response
                        w/status <- SUCCESS(Ø)
                        return

        Get next element in queue

        Write as much to PTY as possible
        error in Write?
        yes:
                call KILLCHAN
                return

        All bytes transferred?
        no:
                save where transfer stopped
```

```
                    channel state to BUSY

                    return

            send TRANSMIT Response
            w/status <- SUCCESS(Ø)
```

OPEN

OPEN is called when an OPEN issued on a PTY completes.

If the non-blocking OPEN done by FRMLOG failed, channel set up cannot complete. In this case all resources are released to the system and a BEGIN Response with an error indication is returned.

Logic

```
Did the PTY OPEN fail?
yes:
        return BEGIN Response
        w/status <- ACTION_FAILED(66)
        free channel data structure
        return

execute a FORK system call
FORK fail?
yes:
        return BEGIN Response
        w/status <- NO_RESOURCES(34)

        free channel data structure
        close PTY's
        return

FORK return Ø?
yes:    (This is the child process)

        Change user ID
        Change group ID
        Change PTY owner to user ID
        Change to user's working directory

        close all files except PTY

        Setup standard input and outputs
```

Close PTY

Setup arguments for execute

execute desired program

Execution of program failed?

Call EXIT with status = CHILD_FAILED(34)

no: (parent process)

return BEGIN Response
w/status <- SUCCESS(Ø)

close PTY

set state to ESTAB

DEATH

DEATH is called when a child process dies.

Logic

call KILLCHAN

REATTA

REATTA is called in response to a "reattach" event from the system.

Logic

Close PTY file.

Call KILLCHAN

PTYREAD

Called in response to a READ event, PTYREAD builds a message large enough to hold the number of bytes indicated by the READ

event and reads the data into it. The message is then sent to CPM. If there is an error on the channel, the channel is destroyed.

Logic

```
Error on channel?
yes:
        call KILLCHAN
        return

Call BLDMSG (to get IPC segment)

Read data into text field
error on read?
yes:
        Call KILLCHAN
        free message segment
        return

send TRANSMIT to CPM
error?
yes:
        free message segment
        log error and return
```

Auxiliary Routines

MAKCHAN

MAKCHAN builds a new channel node and inserts it into the channel list, returning a pointer to it.

Logic

```
Any free channel data structures?
no:
        log error and return zero

delink channel data structure from channel free list

link data structure into active list

copy channel group & member to data structure fields

initialize rest of channel data structure

return address of data structure
```

KILLCHAN

This routine destroys a channel and deallocates all the resources associated with it.

Logic

```
Close PTY

free any IPC resources

child still executing?
yes:
        send KILL to process

send END Command?
yes:
        send END Command without flush
```

delink channel data structure from active list

link channel data structure into channel free list

APPENDIX IV

ARPANET SERVER VIRTUAL TERMINAL SERVICE MODULE

ARPANET Server Virtual Terminal Service (SVTS) Module

Current status. The detailed description of the module which follows is a preliminary version and is subject to change.

Function. The ARPANET Server Virtual Terminal Service (SVTS) module will enable programs in the H6000 to be accessed by terminals on the ARPANET. It will implement the ARPANET Server Virtual Terminal process-to-service protocol described in CAC Technical Memorandum No. 82. It will also implement the ARPANET Telnet protocol described in NIC Document No. 15372. The SVTS module will perform several functions, using the ARPANET NCP in the front end.

1. It will open and close ARPANET connections to hosts on the network.
2. It will pass data between the H6000 and hosts on the network, transforming the data in accordance with Telnet protocol.
3. It will maintain connection status information.
4. It will perform Telnet option negotiation.

The SVTS module will communicate with programs in the H6000 via the CPM.

Structure. The SVTS module will be implemented as a finite state machine which relays and transforms information flowing in two directions:

1. from the CPM (and thus from the H6000) to the NCP and
2. from the NCP to the CPM (and thus to the H6000).

The SVTS module will communicate with the CPM via the IPC mechanism. It will communicate with the NCP via the non-blocking I/O mechanism.

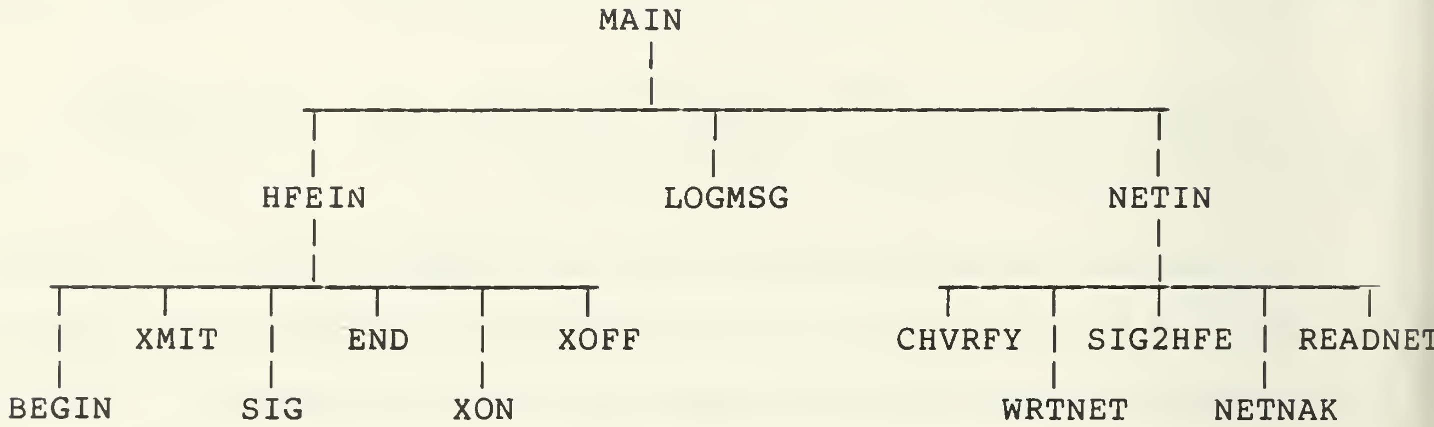
Operation. The SVTS module will wait for IPC messages from the CPM and IPC events from the NCP.

The IPC messages from the CPM will have the form of HFP Commands and Responses. As the SVTS module receives each message, it will call a routine appropriate to the message type. These routines will perform Command-specific functions, transform data, handle error situations, initiate state transitions, and generate HFP Responses.

IPC events from the NCP will signal the completion of network I/O operations. As the SVTS module receives each event, it will call an appropriate routine. These routines will generate the necessary HFP Message sequences and perform Telnet option negotiation.

Software Architecture

The SVTS procedure calling structure is similar to PAS.



State Transition Table

The following table shows states, actions, and state transitions.

STATE	EVENT(input)	ACTION(output)	NEXT STATE
NULL	BEGIN Command	open net chan	PEND
PEND	net open success	notify Host	ESTAB
	net open fail	notify Host	NULL
	END Command	close net chan free resources	NULL
	SIGNAL Command	error	PEND
	EXECUTE Command	error	PEND
ESTAB	net error	notify user free resources	NULL
	TRANSMIT (partial)	data to net	BUSY
	TRANSMIT (full)	data to net	ESTAB
	data from net	send to Host	ESTAB
	END	close channel free resources	NULL
	SIGNAL	do it	ESTAB
	EXECUTE	do it	ESTAB
BUSY	neterror	notify Host flush buffers free resources	NULL
	TRANSMIT	buffer data	BUSY
	data from net	send to Host	BUSY
	data to net gone		ESTAB
	END	let data drain	TERM
	SIGNAL	do it	BUSY
	EXECUTE	do it	BUSY
TERM	data drained	notify Host	NULL
	SIGNAL	do it	TERM
	EXECUTE	do it	TERM
	net error	notify Host	NULL

SVTS Data Structures

Channel list

Channel information is kept in a linked list of structures. The following fields are necessary to hold channel information (numbers in parens are the number of bits in the field):

```
link    (16)     - address of new channel list element
group   (16)     - channel group ID
member  (16)     - channel member ID
state   (8)      - channel state
flag    (8)      - channel flag bits
size    (16)     - number bytes waiting to be read from NCP
currseg(16)      - ID of TRANSMIT being sent to the NCP
fid     (8)      - NCP file ID for this channel
sindex  (8)      - next queued TRANSMIT to send
segs[N*8]        - list of HFP Messages being output
```

Open structure

When SVTS performs a network OPEN on a given channel, it must pass a larger amount of information to Unix than is normal for an OPEN. This is done with a structure containing the following fields:

o_op(8) - used internally by the NCP.

o_type(8) - connection type:

```
bit Ø:  init/listen
bit 1:  icp/direct
bit 2:  duplex/simplex
```

o_id(16) - internal to NCP.

o_lskt(16) - host's local socket for this connection.

o_fskt(32) - socket in foreign host to which connection is to be attempted.

o frnhost(8) - foreign host identifier.

o bsize(8) - sizes of bytes used on the connection.

o nomall(16) - nominal allocation in bytes.

o timeo(16) - number of seconds to wait before timing out on attempt.

o relid(16) - internal to NCP.

Program Logic

HFP Response status codes and their names are defined in the HFP Specification. In the software logic descriptions, each status code is represented by its name followed by its value in parentheses.

MAIN

MAIN sets up all of the necessary resources and falls into a loop where it waits for an event to occur. For security purposes, any event that does not originate from the CPM module or the NCP is ignored.

Logic

```
Link Channel structures into the free list
Initialize IPC variables
loop:
        wait for IPC event

        event source = CPM?
        yes:
                call HFEIN
                return

        event source = NET?
        yes:
                call NETIN
                return

        (if we got here, event is from bad source)
        log event as an error and drop on floor
```

HFEIN

HFEIN reads the message from the CPM and attempts to determine if the Command is a legal one.

Logic

```
Get IPC message from CPM
error getting message?
yes:
        log error and return

legal Command?
no:
        send Command Response
        w/status <- COMM_NOT_IMPLEMENTED(3)

        return

call FINDCHAN

call Command procedure
```

Following are the five routines immediately subordinate to HFEIN that handle the HFP Commands.

BEGIN

A BEGIN Command is received when a host process wishes to "listen" on an ARPANET socket. The Command TEXT contains parameters for that "listen" request.

Logic

```
Channel structure found?
yes:
        return BEGIN Response
        w/status <- CHANNEL_IN_USE(32)

        return

Get a message segment
Can't get it?
yes:
        return BEGIN Response
        w/status <- ACTION_FAILED(66)

        return

Copy security field to message
```

```
Send it to login service
error sending to service?
yes:
        return BEGIN Response
        w/status <- ACTION_FAILED(66)

        return

Call MAKCHAN
MAKCHAN fail?
yes:
        return BEGIN Response
        w/status <- NO_RESOURCES(34)

        return

set channel state <- PEND
```

XMIT

XMIT transfers data to the network when it arrives from the CPM. A channel is considered BUSY when a previous non-blocking WRITE did not accept all of the data.

Logic

```
Channel found?
no:
        return TRANSMIT Response
        w/status <- CHAN_NOT_FOUND(1)

        return

Channel status not <- ESTAB or BUSY?
yes:
        return TRANSMIT Response
        w/status <- ILLEGAL_STATE(2)

        return

set up transfer

Channel state = BUSY?
yes:
        queue TRANSMIT to be sent to network
no:
```

```
write TRANSMIT data to network
error in network write?
yes:
        return TRANSMIT Response
        w/status <- ACTION_FAILED(66)

        call KILLCHAN
        return

all bytes transferred?
yes:
        return TRANSMIT Response
        w/status <- SUCCESS(0)
no:
        set channel state <- BUSY
        save TRANSMIT information
```

SIGNAL

As data is received from the network, it is passed on to the CPM for transmission to the Host. Therefore, signals asking to flush data in transit to the Host are ignored here and handled by the CPM.

Logic

```
bit 1 of CONTROL on?
yes:
        release any queued segments

bit 2 of CONTROL on?
yes:
        ignore. data going to the Host is
        not buffered here.

bit 3 of CONTROL on?
no:
        bit 1 of CONTROL on?
        yes:
                send ARPA INS for this connection.

        send SIGNAL Response
        w/status <- SUCCESS(0)
yes:
        bit 1 of CONTROL on?
        yes:
```

```
                mark signal so that INS is sent
                later.

        room to queue signal?
        no:
                send SIGNAL Response
                w/status <- ACTION_FAILED(66)
                return
        yes:
                queue SIGNAL Command
                SIGNAL Response will be returned
                when all TRANSMITS have been
                sent to the network.
```

END

If there is no queued output when an END is received, the channel is immediately closed; otherwise, its state is set to TERM and the data is allowed to drain to the net. At this time, the response is always successful as long as the channel exits.

Logic

```
Channel data structure found?
no:
        return END Response
        w/status <- CHAN_NOT_FOUND(1)
        return

flush requested?
yes:
        flush queued TRANSMITS

channel busy?
no:
        call KILLCHAN
        return END Response
        w/status <- SUCCESS(Ø)
yes:
        set channel state <- TERM
        save END Command.
```

CPM-Service Flow Control

XON and XOFF events are used by the CPM to flow control

TRANSMIT Commands from the services. When a service receives an XOFF event from the CPM, it should not send any TRANSMIT Commands until an XON event for the associated channel arrives.

XON

XON is called in response to an XON event received from the CPM.

Logic

```
clear XOFF bit in channel state
data to be read?
yes:
        call READNET.
```

XOFF

XOFF is called in response to receiving an XOFF event from the CPM.

Logic

```
set XOFF bit in channel state
clear channel size
```

Login Handler

LOGMSG

LOGMSG is called when the login service returns the success or failure of a user verification request. A success is indicated by a message event. A failure is indicated by the absence of a message accompanying the event.

Logic

```
Login failure?
yes:
        send BEGIN Response
        w/status <- ACTION_DENIED(2)

        free channel structure
        return

fill in
        connection type
        foreign host
        foreign socket
        local socket

initiate network file OPEN request

error in OPEN request?
yes:
        send BEGIN Response
        w/status <- ACTION_FAILED(66)
```

Network Section of SVTS

NETIN

When an event from the NCP arrives, NETIN is called to determine the next state. If the file ID cannot be found in the channel list, the event is discarded. Otherwise the source determines which subordinate routine is called.

Logic

```
Find network file ID?
no:
        log error and return

Known event source?
no:
        log error and return

call event routine
```

The following are the routines called by NETIN.

CHVRFY

CHVRFY is called when a foreign host asks for a connection.

Logic

```
Get saved BEGIN Command.

error returned by NCP?
yes:
        return BEGIN Response
        w/status <- ACTION_FAILED(66)
        call KILLCHAN
        return

set channel state <- ESTAB
return BEGIN Response
```

w/status <- SUCCESS(0)

READNET

NETIN or CHVRFY call READNET when data is available on a channel.

Logic

```
data available?
yes:
        call BLDMSG (to get an IPC data segment)
        set up network read
        read data into IPC segment
        call SNDMSG to transfer data to Host.
no:
        (channel is dead)
        call KILLCHAN
```

WRTNET

When a write to a channel completes, WRTNET is called. Terminating channels and wait-for-drain signals must be handled here.

Logic

```
Channel dead?
yes:
        call KILLCHAN
        return

get current segment

more data in current TRANSMIT Command
to send to the network?
no:
        call NEWSEG

more TRANSMITS to send?
no:
        Channel state = TERM?
        yes:
```

```
                    call KILLCHAN

            return

        set up network write
        write data to net
        error in network write?
        yes:
                call KILLCHAN
                return

        save bytes remaining to be written
```

<u>NEWSEG</u>

NEWSEG is called when the data from a segment has been sent to the network and its successful transfer has been confirmed.

<u>Logic</u>

```
        return TRANSMIT Response
        w/status <- SUCCESS(Ø)

        Loop:
                more TRANSMITS queued to send?
                yes:
                        get first queue element
                        element a SIGNAL Command?
                        yes:
                                send ARPANET INS
                                return SIGNAL Response
                                w/status <- SUCCESS(Ø)
```

<u>NETNAK</u>

When a transmission to a foreign host cannot reach that host for some reason, NETNAK is called to resend the data.

<u>Logic</u>

```
        Set up network write
        write as many bytes as possible to network
        error in writing?
        yes:
```

```
            call KILLCHAN
            return

        save write information
```

SIG2HFE

SIG2HFE is called when an event for the NCP is received stating that an ARPANET INS has been received.

Logic

```
        find net file ID?
        no:
                log error and discard event

        send SIGNAL Command to Host
        w/Control specifying Interrupt
```

Auxiliary Routines

MAKCHAN

MAKCHAN builds a new channel node and inserts it into the channel list, returning a pointer to it.

Logic

```
Any free channel data structures?
no:
        log error and return Ø

delink channel structure from free list

link structure into active list

copy channel group and member into structure field

initialize rest of channel data structure

return address of structure
```

KILLCHAN

This routine destroys a channel and deallocates all the resources associated with it.

Logic

```
Free any queued IPC segments

send END Command?
yes:
        send END Command without flush

delink channel data structure from active list

link channel data structure into channel free list
```

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CAC Document Number 221 CCTC-WAD Document Number 7502	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Network Research in Front Ending and Intelligent Terminals, Experimental Network Front End Functional Description		5. TYPE OF REPORT & PERIOD COVERED Research
		6. PERFORMING ORG. REPORT NUMBER CAC Document No. 221
7. AUTHOR(s) S. F. Holmgren, P. A. Alsberg, G.R. Grossman and P. B. Jones		8. CONTRACT OR GRANT NUMBER(s) DCA-100-76-C-0088
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Advanced Computation University of Illinois Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Command and Control Technical Center WWMCCS ADP Directorate 11440 Isaac Newton Sq. N. Reston, VA 22090		12. REPORT DATE January 15, 1977
		13. NUMBER OF PAGES 101
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Copies may be requested from the address given in (11) above.

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

No restriction on distribution

18. SUPPLEMENTARY NOTES

None

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

network front end
network protocol

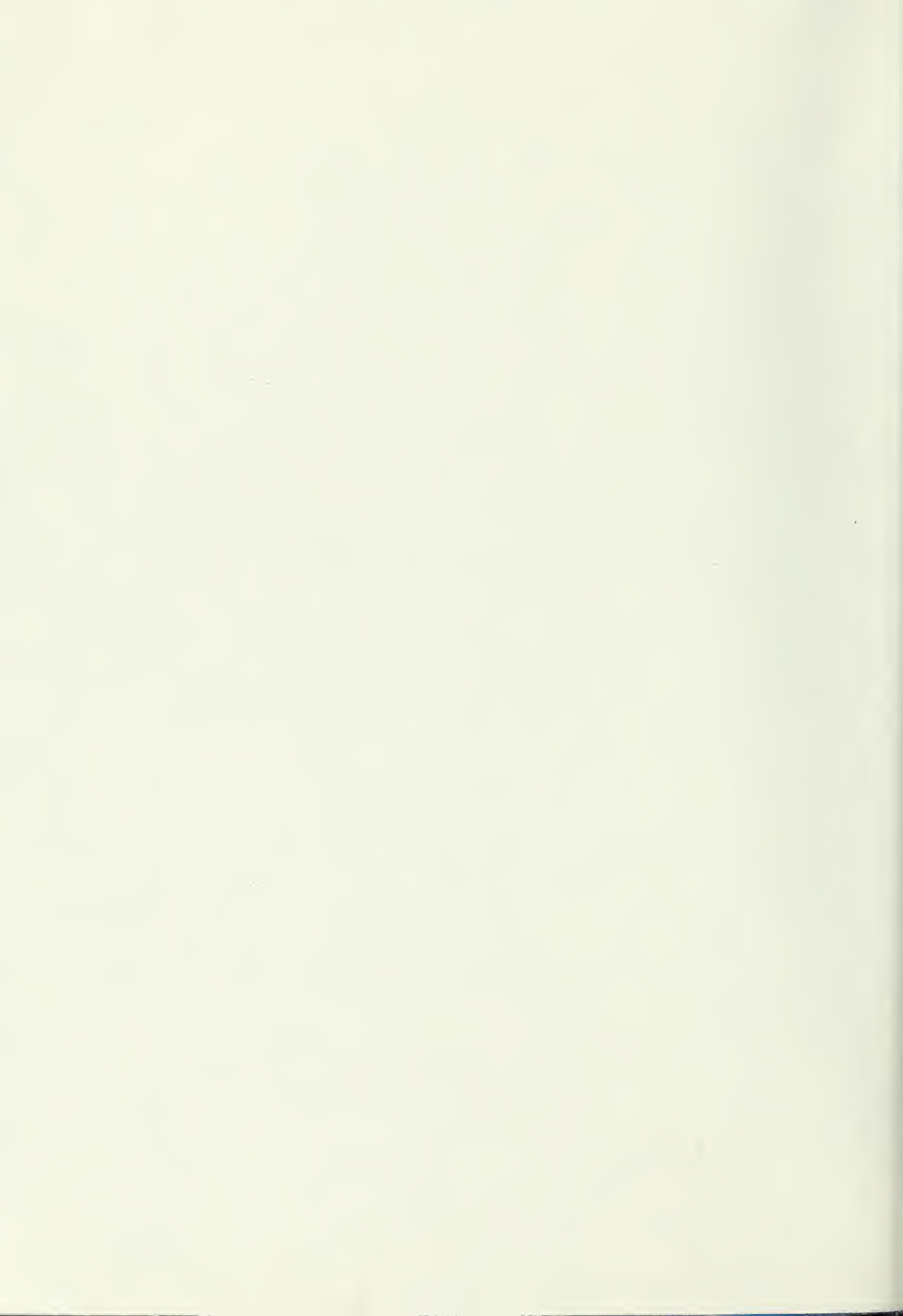
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The CAC is engaged in an investigation of the benefits to be gained by employing a network front end. A DEC PDP-11/70 is being used as front end for connecting a Honeywell 6000 host to the ARPANET. This document presents a functional description of the software required in the PDP-11/70 for allowing H6000 access to the ARPANET. Some implementation - specific details not supplied in other documents are presented.

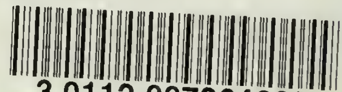
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



BOUND BY
HNM


UNIVERSITY OF ILLINOIS-URBANA
510.84IL63C C001
CAC DOCUMENT$URBANA
220-221 1977
3 0112 007264002