LEVEL II

(12)

RADC-TR-81-164, Vol I (of two)
Final Technical Report
June 1981

AD A105035

UNIX NSW FRONT END ENHANCEMENTS

Bolt Beranek and Newman, Inc.

Robert H. Thomas
Henrik O. Lind
Stephen G. Toner

UNIX NSW FRONT END ENHANCEMENTS, Vol I

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

81 10 5 024

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-164, Volume I (of two) has been reviewed and approved for publication.

APPROVED: [illegible]

PATRICIA J. BASKINGER
Project Engineer

APPROVED: [illegible]

JOHN J. MARCINIAK, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(11) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-164-Vol-1 (of two)	2. GOVT ACCESSION NO. AD-A105 035	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) UNIX NSW FRONT END ENHANCEMENTS. Vol I.		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report, Jan 80 — Oct 80
		6. PERFORMING ORG. REPORT NUMBER BBN-4571-Vol-1
7. AUTHOR(s) Robert H. Thomas Henrik O. Lind Stephen G. Toner		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0062
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman, Inc. 50 Moulton St. Cambridge MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55812124
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISCP) Griffiss AFB NY 13441		12. REPORT DATE June 1981
		13. NUMBER OF PAGES 212
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same (12) 209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Patricia J. Baskinger (ISCP)

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

Computer Networks
Software Systems
Network Operating Systems

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

The effort to develop a UNIX NSW Front End is part of the National Software Works (NSW) program sponsored jointly by the Air Force and ARPA. The goal of the NSW program is to develop a network operating system (called the NSW system or NSW for short) that provides an effective environment for software production, software configuration control and software maintenance.

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 107

Users access the National Software Works system by means of an NSW Front End. The objective of the UNIX NSW Front End project was to develop an NSW Front End that runs on a DEC PDP-11 computer under the UNIX operating system.

UNIX NSW FRONT END ENHANCEMENTS

Robert H. Thomas

TABLE OF CONTENTS

1. INTRODUCTION

This report is the final report for Rome Air Development Center contract F30602-C-80-0062 titled "Unix NSW Front End Enhancements".

The effort to develop a Unix NSW Front End is part of the National Software Works (NSW) program sponsored jointly by the Air Force and ARPA. The goal of the NSW program is to develop a network operating system (called the NSW system or NSW for short) that provides an effective environment for software production, software configuration control, and software maintenance.

Users access the National Software Works system by means of an NSW Front End. The objective of the Unix NSW Front End project was to develop an NSW Front End that runs on a DEC PDP-11 computer under the Unix operating system.

Work to develop a Unix NSW Front End had been begun under another contract (F30602-78-C-0242). The purpose of this contract was to continue the software development that was started under the previous contract. More specifically, the objectives of this contract were threefold:

o To enhance the capabilities of the Unix Front End developed by BBN under contract F30602-78-C-0242. This included completing implementation of the Front End design as documented in RADC-TR-80-44 (also BBN Report 4242), as well as designing and implementing several new

Front End features.

- o To install the Unix operating system and the Unix NSW Front End on a PDP-11/45 at Warner-Robins Air Force Base.

- o To support and maintain the software that implements the Unix NSW Front End.

Development of the Unix NSW Front End involved software design and implementation work in three areas. These were:

- o The Unix operating system.

  A number of enhancements to Unix were necessary to enable it to support an NSW Front End. In particular, modifications to the terminal handler, the ARPANET network control program (NCP), and the implementation of the TELNET protocol were required.

  At the beginning of this contract the Unix modifications required to support the Front End were complete. These modifications had also been integrated into the "standard" version of Unix that is used and maintained by BBN.

- o Unix MSG.

  MSG is the NSW interprocess communication mechanism. It is used to support communication among components of the NSW system (See BBN Report No. 4702). An implementation of MSG was required for Unix.

  At the start of this contract the Unix MSG implementation was partially complete. Two of the three communication modes supported by MSG, messages and alarms, had been implemented. The third mode of communication, direct connections, is required by the Front End but had not been implemented.

- o Unix Front End Software.

  This is the software that actually implements the Front End functions. These functions fall into two categories: the user interface, which provides the user with a convenient environment for interacting with the NSW by means of NSW commands; and a network interface which communicates with NSW software modules on other hosts by means of MSG in order to carry out user commands.

At the start of this contract an initial implementation of the Front End software was partially complete. The Front End implemented most of the NSW user commands not requiring use of MSG direct connections. More specifically, most commands except those that start and stop tools were operational. In terms of the two aspects of the Front End functionality (user interface and network interface), initial implementation of the user interface functions was nearly complete and initial implementation of the network interface functions was partially complete.

For this contract the following was accomplished:

- o MSG direct connections were implemented for Unix MSG. In addition, a number of other enhancements were made to Unix MSG (See Section 2.1).
- o The Front End software required to implement the Front End design (as documented by RADC-TR-80-44) was completed. The major accomplishment here was to implement the software necessary to control interactive tools. In addition, a number of other features were added to the Front End (See Section 2.2).
- o The Unix operating system was installed on a PDP-11/45 computer at Warner-Robins Air Force Base.
- o The Unix Front End software was installed on the Warner-Robins PDP-11/45.
- o The Unix Front End software was installed on a PDP-11 computer at RADC.
- o The source programs for the Unix MSG and Unix Front End software were "delivered" to RADC as text files stored on the RADC PDP-11 computer in the form of C language programs.
- o A user manual for the Unix NSW Front End was prepared

(See Volume II.)

- A program maintenance manual for the Unix NSW Front End software was prepared (See Appendix A).
- A program maintenance manual for the Unix MSG implementation was prepared (See Appendix B).

The rest of this report is organized as follows. Section 2 describes the improvements made to the Unix MSG implementation and to the Unix NSW Front End software under this contract. Volume II is the user manual for the Front End, Appendix A is the program maintenance manual for the Front End software, and Appendix B is the program maintenance manual for the Unix MSG implementation.

2. UNIX FRONT END IMPLEMENTATION

The Unix NSW Front End is designed to run under the Unix operating system on a DEC PDP-11/45 or PDP-11/70 connected to the ARPANET.

As noted in Section 1, in addition to modifications to Unix, implementation of the Front End involved software development activity in two areas:

- o Unix MSG.

  Section 2.1 describes the improvements that were made to the Unix MSG implementation under this contract.

- o Front End software.

  Section 2.2 describes the improvements that were made to the Front End software under this contract.

2.1 Unix MSG

Improvements were made to the Unix implementation of MSG in the following areas.

2.1.1 Direct Connections

The MSG OpenConn and CloseConn operations were implemented. These enable a process to establish and break a direct, stream oriented communication path with another process. The implementation supports both binary and TELNET connections.

Direct connections are used by the Front End to provide communication paths between its user and interactive tools.

2.1.2 Initial Connection Protocol (ICP) responder

MSG implementations on different hosts initially establish MSG-to-MSG communication paths to support inter-host MSG communication by means of the standard ARPANET initial connection protocol. The ICP is an asymmetric protocol for which one party acts as the "initiator" and the other as the "responder". When an MSG configuration is started, no prior communication has occurred between the MSGs on different hosts. When a process on a host (H1) attempts to communicate with a process on another host (H2), if no MSG-to-MSG communication path exists between H1 and H2 the MSG on H1 initiates an ICP exchange with the MSG on H2 in order to establish the path.

At the start of this contract the Unix MSG was capable of initiating ICP exchanges but could not respond to them. This was adequate since in most cases the transactions involving Front End processes are two party transactions which are initiated by the Front End processes. In fact, all transactions the Front End was capable of participating in when the contract started were initiated by the Front End.

However, the tool start up transaction is a three party protocol, involving processes for the Front End, a Works Manager and a tool Foreman, and in the general case it involves three hosts. Furthermore, the transaction calls for the Foreman process to send a message to the Front End process to initiate communication between the Foreman/tool and the Front End/user. It is possible (when the Front End, Works Manager and tool are all on different hosts) that the Front End host will not have yet communicated with the Foreman host. In this case the Foreman host's MSG must establish an MSG-to-MST communication path to carry the message from the Foreman to the Front End. It does this by initiating an ICP exchange with the Front End (Unix) host's MSG. Thus, to properly support interactive tools the Unix MSG must be capable of responding to ICP exchanges initiated by remote hosts.

The Unix MSG was upgraded under this contract to include an ICP responder.

2.1.3 Extended leader addressing

The MSG specification defines a format for a process address which includes a 16 bit field for the host address of the process. At the time MSG was designed all ARPANET hosts could be addressed by only 8 bits, and so 16 bits for host addressing were adequate for addressing processes on all ARPANET hosts. However, since the MSG design was completed, ARPANET addresses have been expanded to 24 bits (the so-called extended leader addressing that makes it possible for the network to accommodate more that 63 IMPs), and internetwork addresses have been defined to be 32 bits (the additional 8 bits are for a network address).

For MSG the 16 bit host address field in a process address was specified to hold the 8 bit ARPANET host address. Since the ARPANET and most hosts support both old style (8 bit) and new style (24 bit) addresses, this permitted MSG implementations to communicate with one another as long as no NSW configuration included a host that required extended leader addressing.

The Warner-Robins PDP-11 host requires extended leader addressing. In order to permit such hosts to be part of NSW configurations the MSG host addressing conventions had to be redefined. The approach taken to this redefinition was to define a mapping between (some) 32 bit internet addresses and the 16 bit MSG host address field. (Details of and motivation for the approach taken may be found in "MSG Host Addressing", BBN NSW Working Note No. 31, August 28, 1980.)

The Unix MSG was modified to support this new host address convention for MSG.

2.1.4 Receipt of generically addressed messages

At the start of this contract processes on Unix were able to send generically addressed messages, but were not able to receive them. The MSG ReceiveGeneric operation was implemented making it possible for processes on Unix to receive generically addressed messages. Although it is not required for any currently defined Front End protocol scenarios, the implementation of the ReceiveGeneric operation will be useful when (if) NSW file movement and tool operations are developed for the Unix operating system. In addition, it makes the Unix MSG implementation more nearly conform to the MSG design specification.

As part of the effort to implement ReceiveGeneric the "newproc" creation specification, the "restartproc" termination specification, and the "initgm" and "startproc" creation modifiers were also implemented. (A creation specification is a declaration that specifies how processes of a particular generic class are to be created when a generically addressed message for the class arrives and there are no outstanding unsatisfied ReceiveGeneric operations to match it with. A termination specification is a declaration that specifies how a process is to be destroyed when it executes the MSG StopMe operation. A creation modifier is a declaration that provides additional specification information for process creation. See BBN Report

No. 4701 for more details.)

2.1.5 StopMe

The MSG StopMe operation was implemented. StopMe is executed by a process when it has finished its task and no longer has reason to exist. It enables MSG to deallocate the resources dedicated to the process in an orderly fashion.

The Front End executes the StopMe operation when the user session is terminated.

2.1.6 Improved robustness

The ability of the Unix MSG to *recover from* "failures" of various types has been greatly improved. For example, the Unix MSG is now able to continue operation when an MSG-to-MSG communication path breaks unexpectedly. Previously when such a failure occurred the entire MSG configuration would halt. Now when it occurs, the Unix MSG cleans up the internal data structures used to manage communication across the failed communication path and continues operation, maintaining normal communication with other hosts.

2.1.7 Major code reorganization

The Unix MSG implementation was started several years ago under ARPA and Navy support, and was continued and upgraded under the previous (F30602-78-C-0242) and present RADC contracts. Throughout this development the code grew significantly in size and complexity but retained its original organization. As the implementation become more complete it became relatively stable. Given the increasing stability of the code, we felt that the time was right for an effort to clean up the implementation. Under this contract we completed a major reorganization of the Unix MSG implementation which (we feel) has greatly improved its maintainability by making it easier to understand, change and extend.

2.2 Front End Software

Improvements were made to the Unix Front End software in the following areas.

2.2.1 Starting and stopping tools

The Front End commands for starting, stopping, and using interactive tools were implemented. This includes the "use", "resume", "quit terminate", and "quit abort" commands, and the CNTL-N "return from tool to Front End" control character.

As noted previously these tool features required use of MSG direct connections and the MSG capability to respond to ICP exchanges.

2.2.2 Rerun tool command

One of the NSW reliability mechanisms provides a means for a user to recover the results of a partially completed tool session that has been interrupted by any of a wide range of tool host, Front End, Works Manager host, and network communication failures. The mechanism enables the user to instruct his Front End to re-establish communication with the interrupted tool session after the failed component(s) has (have) been restored. Once communication has been established, the user may save partially completed work represented by files that were "trapped" in the workspace of the interrupted tool session by having the files he wishes to retain be "delivered" into NSW file space.

This mechanism requires support from:

- The Foreman to save the workspace contents after a failure (this requires that the Foreman maintain the workspace in a "savable" fashion), to notify the Works Manager that the workspace has been saved, and to restore the workspace contents when (if) communication with the user is re-established;
- The Works Manager to remember that the tool workspace have been saved and to inform the user that it has been;
- The user's Front End to establish communication with a tool Foreman that can restore the workspace.

The "rerun" command is the means by which a user can initiate re-connection with the interrupted tool workspace. The protocol scenario that supports "rerun" is similar to the scenario for starting a tool.

The "rerun" command was implemented for the Unix Front End.

2.2.3 Extended logout commands

The NSW commands "fastlogout" and "logout move" and the Front End "autologout" function were implemented for the Front End.

The "fastlogout" command provides a quick way for a user to log out of the NSW system even if the user has active tool sessions when the command is issued. Any tool sessions that are active when the command is issued are automatically aborted prior to actually logging the user out.

The "logout move" command is equivalent to a "logout" command followed by a "login" command. It provides a convenient way for a user to change his login identity by means of a single command.

The "autologout" function is a failure recovery function invoked by the Front End should the user's connection to the Front End be broken before the user has logged out. It terminates the user's session in an orderly way by requesting tool Foreman processes for any active tool sessions to initiate their workspace saving procedures (see discussion above) prior to logging the user out of NSW. This permits the user to "rerun" any active tools sessions at some later time (see Section 2.2.2).

2.2.4 Access to Unix shell

The standard Unix command interpreter is called the "shell". It provides users access to all of the standard Unix features.

A "shell" command was implemented for the Unix Front End which permits a user to use Unix as a normal local Unix user. The "shell" command is implemented by creating a Unix process "inferior" to the Front End which runs the program for the Unix shell. Once a Unix shell has been started in this fashion, a user can treat it like an NSW interactive tool in the sense that he can use CNTL-N and the "resume" command to switch between it and the Front End command interpreter (see Section 2.2.5).

2.2.5 Uniform treatment of NSW tools, TELNET connections, Unix shells

The Unix Front End provides access to three types of interactive computing services: NSW interactive tools, native host services on ARPANET hosts, and local Unix services. Access to tools is established by the "use" command, to native host services by the "telnet" command, and to Unix services by the "shell" command.

Once established these services bear many similarities, and we feel that they should be portrayed to users in a uniform fashion. The Unix Front End tries to provide this uniformity in a number of ways:

- o Each service instance, regardless of its type, has an "instance" name which the user can use to refer to the service instance.
- o The user can switch back and forth between the Front End command interpreter and a service instance, regardless of its type, by means fo the "resume" command and the CNTL-N control character.
- o The "terminate" command can be used to terminate a service instance regardless of its type. For tools "terminate" is equivalent to the "quit terminate" command. For shells it destroys the process created to run the shell (see Section 2.2.4), and for TELNET connections it simply closes the connections.

2.2.6 Improved Front End print out and status query features

A significant amount of effort was spent under this contract improving the quality of the print out produced by the Front End. This effort included making the messages printed more informative, better formatted, and, in many cases, more timely.

The last point (more "timely" print out) may require some explanation. The Unix Front End is designed to be capable of performing many operations concurrently. This was done, in part, in recognition of the fact that many of the operations supported by the Front End require a relatively long time to complete (e.g., minutes to terminate a tool when there are large files to deliver) and that while these long operations are being performed users should be able to initiate other operations or interact with services, and, in part, because the Front End inherently involves a moderate degree of concurrency (e.g., a user can have simultaneous active tool sessions). Because of this potential for concurrent operation events requiring user notification or solicitation of user response do not always occur when the user is ready or would like to be notified or to respond. For example, a tool start operation might complete while a user is interacting with another tool. In situations like these, the Front End should postpone user notification until an "appropriate" time. Choosing an "appropriate" time for user notification or solicitation of a user response required careful analysis and a moderate amount of experimentation and trial and error to get it "right".

While improving the Front End print out did not add to the Front End capabilities in a quantitative way, we feel that it was time well spent since, in our opinion, it resulted in a significant qualitative improvement in the Front End by making it substantially easier and more pleasant to use.

A CNTL-T capability was added to the Front End which provides a means for the user to obtain a short print out of the Front End status without interrupting any transaction processing that may be in progress by the Front End. In addition, the variants of the NSW "show" command that provide information on the status of the Front End ("show active tools", "show active commands", and "show status") were improved.

Appendix A

UNIX NSW FRONT END PROGRAMMER´S MAINTENANCE MANUAL

Henrik O. Lind

TABLE OF CONTENTS

LIST OF FIGURES

1. GENERAL DESCRIPTION

1.1 Purpose of Front End Program Maintenance Manual

The purpose of this Program Maintenance Manual (PMM) for the UNIX[1] implementation of the Front End (FE) is to provide maintenance programmer personnel with sufficient information to maintain that implementation. The reader is assumed to be familiar with the UNIX FE User Manual [3].

1.2 Front End Application

The Front End was designed to be the user interface for the National Software Works (NSW) system. The NSW system is an operating system for a collection of heterogeneous computers (called hosts) connected to a computer network. NSW itself is implemented by a collection of modules which execute as processes on the various host computers. The ARPA computer network (ARPANET) supports inter-host communication for the current NSW implementation.

The UNIX Front End supports: (1) user communication with other NSW components by interfacing to MSG, the inter-process communication facility for NSW; and (2) user communication with

[1] UNIX is a trademark of Bell Laboratories.

Conversational Partners (NSW interactive tools, ARPANET hosts, and UNIX subshells).

FE communication with other NSW components is accomplished by calling MSG communication primitives to (1) send specifically addressed messages, (2) send generically addressed messages, (3) receive specifically addressed messages, (4) send alarms, and (5) open and close direct connections to NSW tools. The Front End manages the execution of NSW protocol scenarios which are defined in terms of the MSG primitives referred to above. User communication with other NSW components occurs almost entirely by the means of executing NSW protocol scenarios.

FE communication with Conversational Partners occurs in a way dependent on each kind of partner. Communication with NSW interactive tools is accomplished by supporting direct binary duplex ARPANET connections between the Front End and the NSW Foreman component associated with the NSW tool. The Front End opens and closes the connections in conjunction with NSW tool-activation and tool-ending scenarios.

Communication with ARPANET hosts is accomplished by supporting TELNET connections to these hosts. The Front End manages the connections in conjunction with user-level commands to initiate and close the connections.

Communication with UNIX subshells is accomplished by forking a UNIX process (inferior to the FE) and communicating with it by means of UNIX pipes. The Front End manages the subshells in conjunction with user-level commands to create and kill the subshells.

The Front End communicates with the user by means of the user's terminal.

1.3 Equipment Environment for the Front End

The UNIX implementation of the Front End requires a hardware base capable of supporting the UNIX operating system as modified and maintained by BBN. This hardware is a DEC PDP-11 model 45 or higher processor (capable of supporting separate Instruction and Data space) with at least 128KBytes of main memory and typically 20MBytes or more of secondary storage. In addition, a host interface to an ARPANET IMP (Interface Message Processor) is required.

1.4 Program Environment of the Front End

The UNIX implementation of the Front End runs under the UNIX operating system as modified by BBN. This is a version of the Bell Labs version 6 UNIX system and contains the following enhancements:

ARPANET NCP and network-related system calls.
Ports
The _awtenb_, _awtdis_, _await_, and _capac_ system calls.
The _itime_ system call.

1.5 Conventions

This section documents the principal programming conventions used in the UNIX FE implementation.

1.5.1 C Language Implementation

The UNIX FE is implemented in the C programming language, using an enhanced form of the version 6 C compiler. See [1]; this refers to some features not supported by the compiler used.

1.5.2 Machine-Dependent Code

The UNIX FE source code is machine-dependent in sections that require taking account of PDP-11 addressing conventions (i.e., the low-order byte of a 16-bit word has a lower address than the high-order byte of the word) and word length (an _int_ on a PDP-11 is 16 bits long, but would be 32 bits long on a VAX, for example; in some sections of the source code, it is assumed that an _int_ is 16 bits long).

The symbol L_INDIAN is defined and used with compiler control lines to enable operations that must be present because of PDP-11 addressing conventions (e.g., swapping the bytes of 16-bit word). This is incompletely implemented.

At present there is no delineation of sections of code which make assumptions about the word size of C variables.

1.5.3 Function Commenting Conventions

Each C function begins on the top of a page. Preceding the function source code is a <u>function header</u> which contains the following items:

Function name, with formal parameters (if any)
Explanation of parameters (if any)
A note explaining the purposes of the function
Item(s) modified by the function
What the function returns
UNIX system calls made in the function
Other C functions called

1.5.4 Naming Conventions

1.5.4.1 Naming of Variables

C variable names are formed from lower-case alphabetics, numerics, and the underline character. All C compilers limit variable names to a maximum length of 7 characters for global symbols and 8 characters for local symbols. No variable names used exceed these limits.

1.5.4.2 Naming of Defined Constants

Defined constants are formed from upper-case alphabetics, numerics, and the underline character. The C macro preprocessor does not reliably recognize defined constants longer than 8 characters. No defined constant exceeds this length.

If a defined constant is a flag name used in a status word of a structure, it will be formed by the structure tag, followed by underline and an alphanumeric string. For example, 'AS_BLOCK' is a flag in the status word of an active scenario table entry in the Protocol Process.

1.5.4.3 Naming of Functions

C function names are formed from upper- and lower-case alphabetics, numerics, and the underline character. Each function name begins with an upper-case alphabetic character. Since function names are considered to be external, they do not exceed seven characters in length.

UNIX system calls have names formed from lower-case alphabetics, numerics, and the underline character. Each function header (see Section 1.5.3 above) has a category called ´System calls made:´ from which it is readily apparent which UNIX system calls are made in the function.

1.5.4.4 Naming of Textual Macros

Textual macros (functions defined using the C macro preprocessor) have names formed from lower-case alphabetic characters.

1.5.4.5 Naming of Structures

C structures are named according to a rigid naming scheme. The structure tag is a two-character string, and each structure member is formed by appending ´_<string>´ to the structure tag, where ´<string>´ is a one- to four-character string. For example, the structure ´as´ in the Protocol Process stores active scenario information and is defined to be:

```
struct as *as_pent;      /* ptr to previous entry */
struct as *as_nent;      /* ptr to next entry */
int as_sid;              /* scenario identifier */
int as_stid;             /* stream id */
struct as *as_prev;      /* ptr to previous scenario */
struct as *as_next;      /* ptr to next scenario */
int as_flag;             /* status flags */
int as_scen;             /* FE ptcl scenario number */
struct pt *as_ptpt;      /* ptr to current ptcl step */
struct tc *as_tcpt;      /* ptr to current t-block */
int *as_uspt;            /* ptr active session entry in UP */
int as_attp;             /* file ptr of associated tool */
int as_pcod;             /* value of us_pcod for scen */
long as_tmis;            /* time last RcvSpec issued */
```

Each structure member has a comment following which describes it. In most cases but not all, if the structure points to another structure, the <string> portion of the structure member name is made up of the structure tag name of the structure which is pointed to, followed by ´pt´; e.g., ´struct tc *as_tcpt;´.

1.5.4.6 Naming of Structure Pointers

If a structure pointer is not a member of a structure, the first two characters of the name are the structure tag of the structure to which it points. For example, ´tcptr´ points to a structure whose structure tag is ´tc´.

1.5.5 Global Data Declarations

For each FE process, there is a module exclusively devoted to global data declarations. There is, in addition, a corresponding .h file which is included by all modules which reference the global data.

Some modules also have module-specific global data, which is found on the first page of the module (following the #include lines - see below).

1.5.6 Declaration Files

Declaration files have extension .h and a name that indicates its domain of reference. These files are used to define constants and structures and are included on page one of each module using C compiler-control lines of the form:

```
#include "filename.h"
```

The Front End makes reference to structures and constants defined outside itself; these necessitate including declaration files for the UNIX terminal driver and User TELNET package, both of which are employed by the Front End.

1.6 Status of Implementation

The implementation is complete. The following gives recommended implementational improvements:

- o Delineate code which depends upon PDP-11 addressing conventions (see Section 1.5.2) by the defined constant L_INDIAN in ifdef compiler-control lines.
- o The code which reads the Front End initialization file is very slow and resource-consuming and ought to be recoded with buffered I/O, preferably with stdio using the Phototypesetter License (pcc) compiler. This will also obviate the need for function PrtLong and will simplify functions ReadWrd, SkipWrd, and SkipLin.

- Delineate code which assumes a 16-bit word size (see Section 1.5.2).
- Modules fntool.c, mpars.c, mprt.c, mutil.c, and mwrite.c should be libraries.

2. SYSTEM DESCRIPTION

This chapter documents the principal program modules of the UNIX FE implementation. It describes the structure, operation, and composition of the Front End implementation.

2.1 General Description

2.1.1 Front End Processes

A Front end configuration on UNIX is implemented as a collection of UNIX processes. Each configuration consists of three processes:

1. The Protocol Process
2. The Tool Process
3. The User Process

The Protocol Process (PP) is responsible for managing the NSW protocol scenarios by which the Front End communicates with other NSW components. It issues MSG communication primitives to do this; these primitives are part of UNIX MSG and are loaded with the PP.

The Tool Process (TP) handles the Front End interface to Conversational Partners. This includes opening and closing ARPANET connections to NSW tools, and managing TELNET connections and pipes to UNIX subshells. It reads from the user's terminal and directs the data to the appropriate network connection or

UNIX pipe. Similarly, it reads data from network connections or UNIX pipes and either displays it on the user's terminal or buffers it until the user requests to view it.

The User Process (UP) is the controlling process of the Front End and performs the user exec interface, which includes command interpretation, command dispatch, and output of results.

The Front End initialization process (see Section 2.2.12 below) is present only at start-up. Its only function is to read a default or explicit initialization file to determine the pathname of the UP and overlay itself with that UP using the *execl* system call.

A Front End configuration can be started manually from the UNIX shell or automatically at login to UNIX if the user's account password file entry contains a pointer to the Front End initialization process. Part of the PP initialization procedure is to create a local MSG for the Front End. If the local MSG finds no central MSG ready, it will cause one to be created. Refer to the UNIX MSG and FE User Manuals for further details concerning initialization.

2.1.2 Process Structure of a Front End Configuration

The process structure of a Front End configuration is shown in Figure 1. The figure shows the associated local MSG. Note also that user input and output is done by two processes: the

UP, when the user is issuing NSW commands; and the TP, when the user is communicating with Conversational Partners.

2.1.3 Communication Between Front End Processes

The processes which implement a Front End configuration must communicate with each other in order to perform their functions. This inter-process communication is achieved through the use of UNIX pipes.

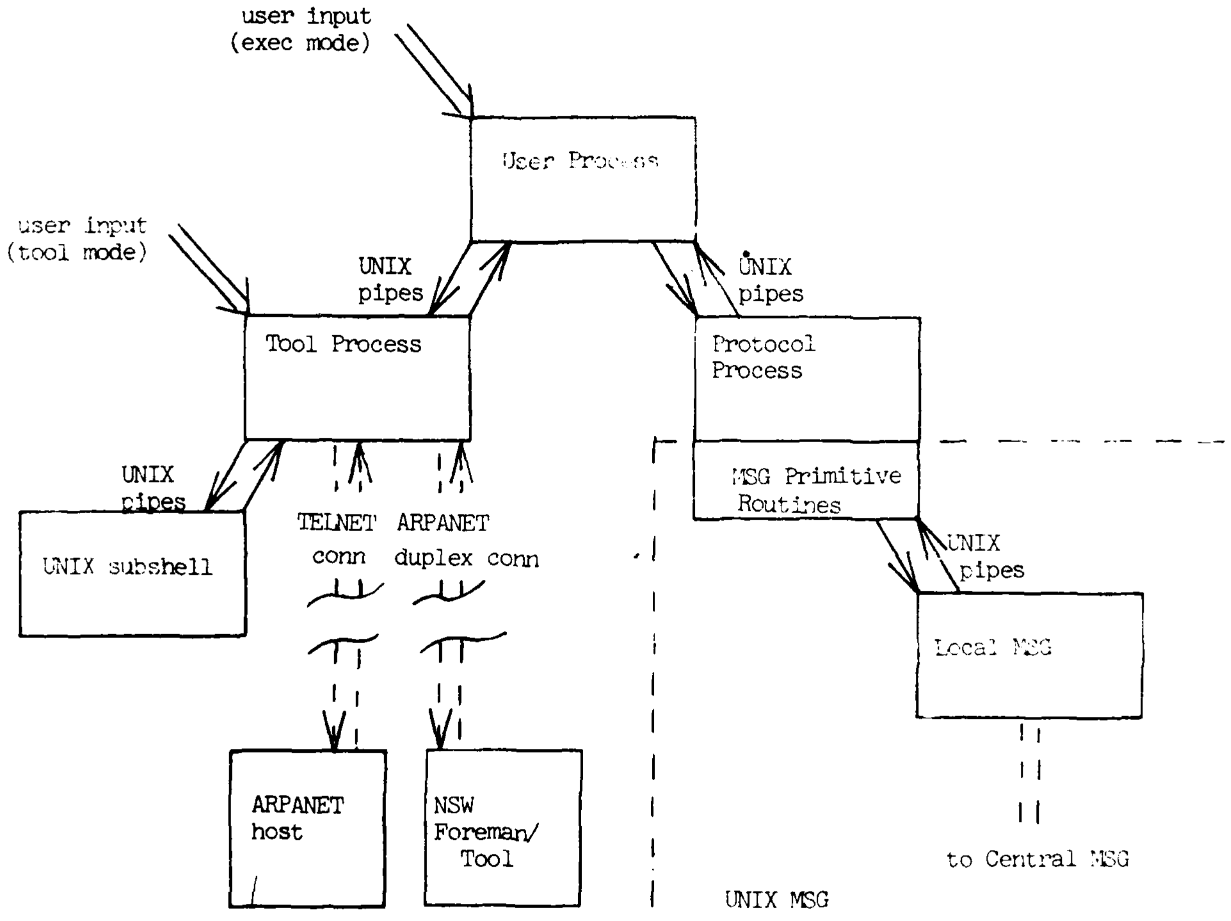
Communication with NSW tools or TELNET server processes on remote hosts is achieved by using the standard ARPANET communication functions provided by the UNIX Network Control Program (NCP).

2.1.4 General Front End Flow of Control

The three Front End processes are event-driven, using the BBN-developed _await_ and _capac_ mechanism (see Section 1.4 above). Each process waits until an event occurs, at which time the process awakened executes one pass through its central loop in an attempt to process the event.

The central loops of the Front End processes are given next in a pseudo-English form:

FIG. 1. FRONT END CONFIGURATION PROCESS STRUCTURE



```
/* main loop of Protocol Process */

while (Front End is running)
{
   compute time to next wakeup;

   await (until event occurs or time is up);

   process input from User Process;

   process input from MSG;

   advance all protocol scenarios where possible;

   check ReceiveSpecific timeouts;
}

/* main loop of Tool Process */

while (Front End is running)
{
   await (until event occurs or await times out);

   if (user talking to Conversational Partner)
      process input from user´s terminal;

   cycle thru each Conversational Partner
   {
      if (network connection) process net input;

      if (shell) process pipe input;
   }
   process input from User Process;
}
```

```
/* main loop of User Process */

while (Front End is running)
{
   await (until event occurs or await times out);

   if (user not talking to Conversational Partner)
      process input from user's terminal;

   if (there is a PP) process input from it;

   if (there is a TP) process input from it;

   if (there is terminal output ready)
      announce its presence to the user;
}
```

The following paragraphs describe in general terms how the Front End (1) handles user commands, (2) interfaces to MSG, and (3) handles Conversational Partners.

2.1.4.1 User-Issued Commands

When the Front End is in exec mode, the Front End accepts user input from the user's terminal and tries to parse it according to the NSW command language. For each user token, a node of the parse structure (PS - see Section 3.2.4.1 below) is created. Each token is matched against the set of possibilities defined by the current position of the parse in the grammar tree (GT - see Section 3.2.4.5 below). Thus, a one-to-one correspondence between PS entries and GT entries is created until the end of a complete command is reached. From the grammar tree, a command code is found which determines what to do to "execute" the command.

Executing the command means either assembling a NSWTP message to send to another NSW component or performing an action locally in the Front End. In the case of assembling a message to be sent to another NSW component, usually the Works Manager, the message is assembled in the UP and sent by pipe to the PP. In the PP, an active scenario (AS - see Section 3.2.2.1 below) entry is created, and the message is buffered until it is ready to be sent.

In the case of a "local" command, the UP will either perform the requested action or send a pipe message to the TP requesting it to take some action.

Because of the asynchronous nature of Front End operation, the UP buffers output from user commands and notifies the user that the command has completed. User output from NSW comes to the UP via pipe from the PP. The output comes in groups of one or more pipe messages which are buffered and managed as successive terminal buffer header (TB - see Section 3.2.4.3 below) entries. When the final message arrives, the user is notified that his command is complete. Viewing the command output is performed by invoking a local command to display the output.

2.1.4.2 MSG Interface

The Front End interface to MSG is accomplished by calling MSG communication primitives to send and receive messages, send alarms, and manipulate connections. All possible interactions are categorized in terms of entries in the protocol scenario definition table (PT - see Section 3.2.2.4 below). Executing NSW protocol scenarios occurs by stepping through the entries in PT; each entry typically invokes issuance of an MSG primitive. For each protocol scenario in progress there is an active scenario entry, and each such entry has a PT pointer associated with it.

The UNIX pipe from local MSG to process MSG, i.e., that part of MSG loaded with the PP, is await-enabled. The PP thereby knows when pipe transmissions from the local MSG arrive. If a pipe transmission has come in, the PP issues a RequestSignal MSG primitive to determine which pending event, if any, completed. The Front End uses the MSG Request signal when issuing MSG primitives which create pending events in order to permit the susequent use of the RequestSignal primitive. If a transmission from the local MSG does not signal the completion of a pending event, then some background functions in process MSG are performed. If the transmission signals the completion of a pending event, the active scenario entry corresponding to the pending event is identified. If the active scenario entry corresponds to one initiated by the Front End, a test is made to see that the received message has the expected type, transaction id, and procedure name. If this fails, the scenario is aborted. If the active scenario entry does not correspond to one initiated by the Front End, a test is made to see that the received message validly initiates some protocol scenario known to the Front End.

If the test fails, the message is deleted and the active scenario entry is freed; otherwise, the message is processed.

The Front End always has one, and only one, ReceiveSpecific primitive outstanding. When a message is received, a new ReceiveSpecific is immediately issued and then the received message is processed. Processing a message usually means either sending its fourth argument to the UP or using it in the PP to build a subsequent message in the protocol scenario.

Associated with each step of a protocol scenario is a transaction control block (TC - see Section 3.2.2.2 below). A transaction control block, or transaction block, contains all information needed to issue an MSG primitive; in the case of the ReceiveSpecific primitive, the transaction block contains information obtained by parsing the incoming message. Transaction control blocks are stored for the life of a protocol scenario; messages are retained only for the life of a transaction, i.e., protocol scenario step.

2.1.4.3 Handling of Conversational Partners

Front End Conversational Partners are handled by one process, the TP. For each Conversational Partner, there is an entry in the active tool table (TT - see Section 3.2.5.1 below), a file accessed by all three Front End processes. In the startup sequence for a Conversational Partner, the following happens in the UP: If there is no TP, the UP forks one (the TP exists only when there is at least one Conversational Partner) and then adds an entry to TT with data for the Conversational Partner being started. If the Conversational Partner is an NSW tool, a NSW tool startup (WM-RUNTOOL or WM-RERUNTOOL) protocol scenario is initiated. Otherwise (the Conversational Partner is a TELNET connection or UNIX subshell), an ´activate´ (P_ACTIV) pipe message is sent to the TP. The TELNET connection is opened, or the UNIX subshell is created, and a positive acknowledgement is sent to the UP. A negative acknowledgement indicates failure.

For NSW tools, tool-activation is slightly different. Owing to the fact that UNIX file descriptors are process-relative, it is not possible for MSG to open the direct MSG connection to the NSW tool and pass the file descriptor to the TP. The OpenConn MSG primitive in the UNIX implementation has its functionality effectively distributed between MSG and its user calling process, in this case the Front End. When the protocol scenario comes to the point that an OpenConn must be issued, the following sequence occurs: the PP sends a pipe message to the UP, which is forwarded to the TP, ordering a "dummy open" to be performed by the NCP (the dummy open reserves a block of 8 sockets on UNIX for subsequent actual use by connections); in the TP, a tool buffer header entry (TI - see Section 3.2.3.1 below) is created for the tool; if no block of 8 sockets exists or has a free socket pair, a tool connection information block (TO - see Section 3.2.3.2 below) is created and a "dummy open" is performed; and a positive acknowledgement is sent via pipe to the PP; at this point, the MSG OpenConn primitive is issued; when it completes, an ´activate´ (P_ACTIV) pipe message is sent to the TP from the PP; when the TP receives this message, it actually opens the ARPANET connection to the NSW tool and sends a positive acknowledgement to the PP. This positive acknowledgement, as it passes through the UP, causes the user to be notified that his tool is ready to use.

Ending the use of Conversational Partners is accomplished in a manner also dependent on the kind of Conversational Partner in question. If the Conversational Partner is an NSW tool, an ´endtool´ (P_FINIS) pipe message is sent to the TP from the PP at the end of a tool-terminate or tool-abort scenario. In the TP, this causes the connection to be closed (if still open); a positive acknowledgement to the pipe message indicates the end of the protocol scenario and causes the user to be notified.

For TELNET connections and UNIX subshells, there is no protocol scenario associated with ending use of them. The use of them is ended by user command in exec mode or (in the case of a TELNET connection) by the connection closing.

2.2 Detailed Description

This section describes in detail the principal program modules and routines of the UNIX Front End implementation.

2.2.1 Macros

A number of macros are used extensively in the Front End. They are defined in module fe.h and are included here for reference:

```
/* textual macros used in the Front End */

#define and               &&
#define or                ||
                                  /* decides whether to prepend '0'
                                     to octal number */
#define zero(x)           ((x) == 0? "": "0")
#define setflag(x,y)      (x =| (y))
#define clrflag(x,y)      (x =& (~(y)))
#define entsize(t,n)      (sizeof(t)/2)/(n)
#define freebuf(x)        if ((x) != NULL) free(x)
```

2.2.2 Global Declaration Files

There are two global declaration files which define constants and structures used in all or most of the Front End; they are nsw.h and fe.h.

The module nsw.h contains defined names for NSWTP types and arguments, NSWB8 types and values, values of NSW fault classes and component id's, and values for the FM-BEGINTOOL entry vector variable entvec.

The module fe.h contains definitions global over the Front End, but not specifically related to NSW. Included are common defined constants; names for UNIX system call parameters; definitions related to file-manipulation, error-handling, event-logging, and diagnostic typeouts; Front End protocol scenario numbers and timeouts; structure definitions for parsing NSWTP messages and the process control block (PB - see Section 3.2.5.2 below); and definitions of macros used in the Front End (see Section 2.2.1 above).

2.2.3 Protocol Process

This section describes the program modules which, except for utility modules, comprise the Protocol Process.

2.2.3.1 PTCL

The module ptcl.c contains the following principal C functions:

- o main is the top-level function of the PP. It initializes the PP, forks local MSG and contains the main loop of the process. In the main loop (shown in Section 2.1.4 above), it services input from all the possible sources of input to the PP: namely, MSG and the UP; in addition, it drives the protocol scenario engine AdvanSc and computes the time for the next ReceiveSpecific timeout. It is necessary to compute the timeout since there is no corresponding MSG event which would awaken the PP from waiting. main calls the MSG function Seconds to ascertain the next wakeup time as seen by MSG. The actual time to wakeup will be the minimum of that returned by seconds and by ChRCTmo. At the end of main, there is code to "clean up" before terminating.

- o Hup_PP intercepts the UNIX ´Hangup´ signal when it is sent to the PP. The Hangup signal is sent when the user loses contact with his UNIX processes, e.g., when a TELNET connection to the Front End breaks. This function initiates the Autologout scenario and sets appropriate state variables.

- o ChRCTmo calculates the time at which the next ReceiveSpecific primitive times out.

- o RcvSpTm checks to see if any ReceiveSpecific primitives have timed out. If any have, the corresponding protocol scenarios are aborted, and the user is notified.

Module ptcl.h contains definitions of structures and constants global to the PP.

2.2.3.2 PTCLV

The module ptclv.c contains the global data declarations for the Protocol Process. Module ptclv.h contains external references to all variables declared in ptclv.c.

2.2.3.3 PDC_MB

The module pdc_mb.c contains C functions AddMB and RemovMB which, respectively, add and remove message buffer headers (MB - see Section 3.2.2.3 below) to/from their data class (see Section 3.2.1 below).

2.2.3.4 PISSUE

The module pissue.c contains the C functions which issue MSG primitives. IssuePr is called with a pointer to an active scenario entry as an argument. Based on the PT pointer in the corresponding transaction block, the appropriate issuing routine is called: namely, IsRcvSp issues a ReceiveSpecific primitive; IsSndSp issues a SendSpecific primitive; IsSndGn issues a SendGeneric primitive; PrOpnCn initiates the OpenConn sequence (described in Section 2.1.4.3 above), which includes at a later time issuing an OpenConn; IsClsCn issues a CloseConn primitive; and IsSndAl issues a SendAlarm primitive. These are the only MSG primitives which create pending events that the Front End issues.

The arguments for the primitives are extracted by these routines from the corresponding transaction block and, in some cases, from the appropriate active tool table entry.

The function IsDRcSp issues the so-called "dummy" ReceiveSpecific primitives, of which the Front End always has one outstanding.

2.2.3.5 PPIPE

The module ppipe.c contains the C functions that read the pipe from the User Process and perform actions depending on the instruction field of the pipe message. Module UPInput performs the read and dispatches based on the instruction field. The following describes the principal modules which may be invoked by the dispatch:

AHelpM	user Help message answered
NHelpM	user Help message aborted
IIniScM	user initiates protocol scenario
ADumOpM	TP acknowledges "dummy open"; issues OpenConn primitive
NDumOpM	"dummy open" failed; aborts scenario
AActivM	NSW tool-activation succeeded
NActivM	NSW tool-activation failed; aborts scenario
AfinisM	NSW tool-ending succeeded
NFinisM	NSW tool-ending failed; aborts scenario
IFastLM	user issues Fastlogout

2.2.3.6 PPRIM

The module pprim.c contains C functions which are called in response to NSWTP messages received. The functions include:

- o RutlRep is called when a NSWTP Reply message for a WM-RUNTOOL message is received. In conjunction with FEOpnCn (see below), RutlRep initializes the active tool table entry for the NSW tool being started. If there is already is a TT entry having the MSG process name in the Reply message, then an FE-OPENCONN has already been received. The MSG process name is copied into the actual TT entry; the contents of the temporary entry (created by the prior FE-OPENCONN) are added to the actual TT entry; and the temporary entry is removed (i.e., its tool id is set to null).

- o FEOpnCn is called when a FE-OPENCONN message is received. If there already is a tool table entry having the MSG process name found in the FE-OPENCONN messages, the other arguments of the message are added to the TT entry. If there is no such entry, a temporary entry is added until such time as a tool-initiation Reply message arrives (see under RutlRep above).

- o FEToolH is called when a FE-TOOLHALTED message arrives. Since the contents of the fourth argument of this message must be saved and since messages are deleted very soon after receipt, the fourth argument of the message is copied into a temporary buffer pointed to by the transaction block. If the message is part of a remotely-initiated tool-termination scenario, the file pointer of the NSW tool's TT entry is determined (the PP deals with tools solely on the basis of their file pointers).

- o SndMsDn is called when a MSG Send primitive completes. It is currently a no-op.

- o OpnCnDn is called when a MSG OpenConn primitive completes. It sends an 'activate' (P_ACTIV) pipe message to the TP, which instructs the TP to open the tool's connection and make the tool available to the user.

- o ClsCnDn is called when a MSG CloseConn primitive completes. If the protocol scenario for which the CloseConn completed is part of a Fastlogout sequence, the arguments returned by the preceding FE-TOOLHALTED message are parsed, and the tool id and NSW accounting information are appended to the Fastlogout intermediate file, and the scenario is then flushed. [The Fastlogout intermediate file becomes the fourth argument of the WM-FASTLOGOUT message, once all the active NSW tools have been aborted.] For protocol scenarios which are not part of the Fastlogout sequence, ClsCnDn is a no-op.

- o SndAlDn is called when a MSG SendAlarm primitive completes. It is currently a no-op.
- o FELNDSv is called when a FE-LND-SAVED message has been received. If an Autologout sequence is in progress, it is a no-op. Otherwise, it adds a TT entry to the tool table with the information contained in the message. Finally, it informs the UP of the new LND-saved tool.

2.2.3.7 PRECV

The module precv.c contains C functions related to processing received NSWTP messages. They include the following:

- o RcvSpDn is called when an MSG ReceiveSpecific primitive completes. It checks the message List count and extracts the message type, tid, and procedure name (if there is one) from the message (using ExtrTyp, ExtrTid, and ExtrPnm, respectively). It identifies the message as part of an existing protocol scenario and then calls routines to process the message.
- o MatchSc attempts to match the message received with an outstanding ReceiveSpecific primitive for an existing protocol scenario. The match involves scanning all AS entries and their associated PT pointer to search for scenarios expecting messages. MatchSc performs the conditional branch function in the PT table (described below in Section 3.2.2.4) which is done when a message is received. If the received message is not a part of an existing protocol scenario, RemotSc is called. RemotSc tries to match the received scenario with all possible kinds of messages which validly may initiate a protocol scenario (by calling IdentSc) and then create an active scenario entry for the scenario. Scenarios created in this way are called ´remotely-initiated´.
- o IdentSc is the function which performs the actual match of received message information with PT information for remotely-initiated scenarios.

o LinkTC is called when the received message is part of an existing protocol scenario. In this case, the transaction block associated with the received message is linked with the previous transaction block for the existing scenario, and the "dummy" AS entry associated with the outstanding ReceiveSpecific primitive is deallocated.

o NewSc is called when the message received validly initiates a scenario. It sends a pipe message to the UP notifying it that a new scenario has been initiated and directing it to possibly add corresponding table entries.

2.2.3.8 PSCEN

The module pscen.c contains the following principal C functions:

o CreatSc adds a new active scenario entry to the active scenario table. If the AS entry to be added has the same stream id as an existing scenario, the new scenario is sequenced after the existing scenario. The function performed by CreatSc is the most basic function in initiating a protocol scenario.

o IniStep creates a transaction block for the step of the protocol scenario given as its argument. Transaction block fields are filled, including MSG signal, timeout, and PT pointer. The data structures for the associated message (for Send primitives) are initialized. The block is linked to the corresponding AS entry.

o PrepMes performs functions needed to prepare NSWTP messages to be sent. If there is no message, one is assembled. The proper MSG process name is found and loaded into the transaction block.

o AssemMs assembles NSWTP messages, based on the value of the PT pointer. Most NSWTP messages directly associated with user commands are assembled in the UP and sent by pipe to the PP. Other messages, e.g., intermediate messages required by intermediate steps of the protocol, are assembled by AssemMs. It assembles the following: WM-AUTOLOGOUT, WM-FASTLOGOUT, FM-SAVE-LND, FM-ENDTOOL, WM-TOOLABORT, WM-TOOLHALTED, Null Reply, and Help Reply messages.

- AdvanSc is the "engine" that drives the protocol scenarios forward. It cycles through all AS entries and, for each entry, performs the following: if the scenario is a "dummy" ReceiveSpecific scenario, it is ignored; if, there is a previous scenario linked to the current one, it is ignored; if the PT pointer has not been initialized for the scenario, IniStep is called; if the PT pointer points to end of scenario (op code PT_ENDSC - refer to Section 3.2.2.4 below), mark scenario as done and go to next AS entry; if the current protocol scenario step has completed (flag AS_COMPL is set) move to the next protocol step and go to next AS entry; if the PT pointer points to "go to" (op code PT_GOTO see below) perform the "go to" by advancing the scenario to the indicated location; if the current PT op code is for a ReceiveSpecific (not a "dummy" ReceiveSpecific, but one associated with a known protocol scenario), ignore it; if the scenario is temporarily halted waiting for some other Front End activity to occur (flag AS_BLOCK is set), ignore it; if the protocol scenario step has no associated transaction block, call IniStep; and, finally, issue the MSG primitive. The operation of AdvanSc may be summarized by saying that it advances each protocol scenario forward as far as it may go, subject to the constraints of waiting for remote or local (FE) events to occur.

- DoGoTo executes the "go to" (PT_GOTO) operation in the PT table.

- EndSc sends an "end of scenario" (P_SCDON) pipe message to the UP. This condition indicates (1) that, as far as the PP is concerned, the scenario is complete, and (2) the UP should be aware of that fact.

- FlushSc deletes all table entries associated with a scenario instance from the PP. For sequenced scenarios, it clears the following scenario for execution.

- ParsFTH parses the fourth argument of a FE-TOOLHALTED message and places the results into a parse result structure. If the scenario is part of a Fastlogout sequence, the tool id and NSW accounting information are appended to the Fastlogout intermediate file.

- ToolSc is called when a Fastlogout or Autologout sequence is initiated. It scans the tool table for NSW entries and, for each tool found, initiates a tool abort or FM-SAVE-LND scenario, respectively.

- FastLog is called when all tool abort scenarios in the

Fastlogout sequence have completed; it initiates a protocol scenario to handle the WM-FASTLOGOUT message.

- AutoLog is called when all FM-SAVE-LND scenarios in the Autologout sequence have completed; it initiates a protocol scenario to handle the WM-AUTOLOGOUT message.

2.2.3.9 PSCENTBL

The module pscentbl.c contains the protocol scenario definition table (PT). Module pscentbl.h defines the structure of PT (refer to Section 3.2.2.4 below) and defines constants used in the table.

2.2.3.10 PUTIL

The module putil.c contains utility routines for the PP. These routines are primarily used to send pipe messages to the UP, handle error conditions in protocol scenarios, and manipulate MSG process names.

2.2.4 Tool Process

This section describes the program modules which, except for utility modules, comprise the Tool Process.

2.2.4.1 TOOL

The module tool.c contains the following principal C functions:

- main is the top-level function of the TP. It initializes the TP and contains the main loop of the process. In the main loop (shown in Section 2.1.4 above), it services input from all the possible sources of input to the TP: namely, user's terminal (when the user is in tool mode), the UP, and all network connections and UNIX subshells currently open or present. At the end of main, there is a code to "clean up" before terminating.

- o ToolInt performs the read interface to the ARPANET (NSW tools and TELNET connections). Input from the ARPANET is either displayed on the user's terminal (if he is in tool mode) or is buffered in a core buffer for later display. ToolInt takes as an argument a pointer to a tool buffer header (TI); if the argument equals the contents of <u>tiuse</u>, then the data is displayed on the terminal - otherwise, it is buffered. ToolInt returns its argument, or (if the connection is closed) the pointer to the previous TI entry. Using the <u>capac</u> system call, ToolInt is guaranteed to know how much network data there is to read.

- o ShelInt performs the read interface to UNIX shells running as inferior processes to the TP. Other than the fact that it reads from UNIX pipes, its basic operation is the same as that of ToolInt.

- o TermInt performs the read interface to the user's terminal. When the user is in exec mode, terminal I/O is performed by the UP. When the user transfers to tool mode, the function of handling terminal I/O is taken over by the TP, the read interface by function TermInt. Characters are read from the terminal one at a time until there are no more to read or until control-N (CHGEXEC) is read; control-N causes a transfer to exec mode. TermInt takes the characters as they are read and directs them to the proper destination. This destination may be either a network connection (defined by <u>connp</u>) or a UNIX pipe (defined by <u>ufd</u>). The TP uses three different terminal mode settings depending on the destination of the characters; in each TI entry, there is a pointer to a terminal modes block, and these terminal modes are placed into effect when the user begins to communicate with the connection or pipe in question. For UNIX pipes, linefeed and bell are principal break characters [as well as control-N, of course], and the UNIX terminal driver's facilities for line-editing are used. With the exception of linefeed and bell, no break characters are echoed; linefeed and bell are echoed by the TP. The default terminal mode settings for a network connection are local echoing and linefeed as the primary break character. Local UNIX line-editing features are in effect. Linefeed is echoed as CR/LF. With hosts that negotiate the TELNET Remote Echo option, the break set becomes all Ascii characters and local line-editing capabilities are disabled. For all writes to the network, a linefeed is appended to every carriage-return written.

- o Hup_TP intercepts the UNIX 'Hangup' signal when it is sent to the TP. The Hangup signal is sent when the user loses contact with his UNIX processes, e.g., when a TELNET connection to the FE breaks. This function causes an automatic transfer to exec mode if the user was in tool mode and sets appropriate state variables.

- o UPInput reads the pipe from the User Process and dispatches to routines to handle the message based on the content of the message instruction field.

- o ToolSig intercepts UNIX signal 17 (defined to have name TOOLSIG); TOOLSIG is a user-defined signal and is sent by the UP whenever the user invokes an abort operation. Intercepting the signal has the effect causing an error exit from network opens which may not have completed or timed out.

- o ConnErr processes error conditions on network connections and pipes to UNIX subshells. The connection is closed, and an error message is sent to the UP. UNIX subshells are killed, and an error message is sent to the UP.

Module tool.h contains definitions of structures and constants global to the TP.

2.2.4.2 TOOLV

The module toolv.c contains the global data declarations for the Tool Process. Module toolv.h contains external references to all variables declared in toolv.c.

2.2.4.3 TELNIO

The module telnio.c contains C functions required by the User TELNET library, which manages the Front End network connections. For each TELNET option which may be negotiated, a test function and an action function must be provided by the user program which makes calls on the TELNET library. The test function determines whether the option is in effect, and the action function turns the option on or off. For the Front End, the only TELNET option that may be negotiated is the Remote Echo option; the test function is TsRmEch, and the action function is DoRmEch. These functions comprise telnio.c. Refer to Volume II for further information about the user TELNET library.

2.2.4.4 TPIPE

The module tpipe.c contains the C functions which may be invoked by the dispatch in UPInput in tool.c.

These are:

IActivM	activate an NSW tool, connection is opened.
IShellM	Fork a subshell.
IAcTelM	Activate a TELNET connection; connection is opened.
IDumOpM	Perform a "dummy open".
IResumM	Resume communication with a Conversational Partner; process mode changed to tool mode.
IFinisM	Terminate the use of an NSW tool; connection is closed.
IFnTelM	Terminate the use of a TELNET connection; connection is closed.
IFClosM	Force a connection to close; extraordinary condition in effect.

2.2.4.5 TTOOL

The module ttool.c contains the following principal C functions:

- o OpnTool opens a connection to an NSW tool, based on the arguments passed to it.

- o ClrTool clears the TI and TO entries for a tool or TELNET connection, and frees any buffers which have been allocated for the tool or connection.

- o ClsTool is called when the use of a Conversational Partner is being ended. Depending on the kind of conversational Partner, it will either close a network connection or kill a UNIX subshell. Under some circumstances a 'closed' (P_CLOSE) pipe message will be sent to the UP, and in some cases there will be a process mode change (from tool mode to exec mode). If there is no buffered output for the Conversational Partner, ClrTool (see above) is called. Otherwise, the table entries are left intact so the user may view his buffered output [at that time, the table entries will be removed].

- o T_ChgET is called to effect a process mode change from exec mode to tool mode. Among other things, the terminal is await-disabled, and the function may, depending on its arguments, go through a 'suspend' sequence, send a 'suspend' (P_SUSP) pipe message, or send a 'closed' (P_CLOSE) pipe message. <u>tiuse</u> is cleared.

- o SuspTl performs actions related to suspending the use of a Conversational Partner. This consists of allocating a buffer to hold buffered output for the Conversational partner and setting related variables.

- o WCloseM sends a 'closed' (P_CLOSE) pipe message to the UP.

2.2.4.6 TUTIL

The module tutil.c contains the following principal C functions:

- o IniMBLK initializes the terminal modes block for a new TI entry; it differentiates between network connections and subshells.

- o SetTMod loads and sets the current terminal modes to be those of the terminal modes block for the TI entry argument.

- o **SndErMs** sends a pipe error message to the UP and may terminate the TP. It is a utility routine.

- o **MakShel** initializes a UNIX subshell inferior to the TP. It makes the pipes, forks, sets up the pipes to operate as standard input and output for the shell, and overlays a shell with the execl system call.

2.2.5 User Process

This section describes the program modules which, except for utility modules, comprise the User Process.

2.2.5.1 USER

The module user.c contains the following principal C functions:

- o **main** is the top-level function of the UP. It initializes the UP, forks the PP, and contains the main loop of the process. In the main loop (shown in Section 2.1.4 above), it services input from all the possible sources of input to the UP: namely, the user's terminal (when the user is in exec mode), the PP, and the TP; in addition, it outputs announcements dealing with output ready for the user. This is done because output is, in general, not displayed to the user but is buffered, and the user is then notified that it is ready. To view the output, the user requests it with a command. At the end of main, there is a code to "clean up" before terminating.

- o **U_PDead** is called when the UP determines that the PP has died or stopped itself for some reason. Variables related to the PP are reset, and if the TP has also died or stopped, the 'run' flag (ST_RUN in pb_stat of the process control block) is cleared, which causes the Front End to stop. There is currently no provision for restarting the PP should it terminate abnormally.

- o **U_TDead** is called when the UP determines that the TP has died or stopped itself. If the PP has also died or stopped, the 'run' flag (ST_RUN) is cleared, which causes the Front End to stop. Unlike the PP, the TP may validly be stopped in the course of an Front End session: this occurs whenever the active Conversational Partner count goes to zero. The TP may also die from errors. In any case, variables related to TP are reset and the tool table is scanned, and each TT entry is modified so that a request to reconnect from an NSW Foreman (the Foreman reconnect scenario - PSC_FMRC) will find all tool table entries in the proper state. [If the TP dies because of errors, any TELNET connections and UNIX subshells will be permanently lost].

- U_UDone is called when the Front End is being stopped. Among other "clean up" functions, it resets the terminal modes to those prior to entry to the Front End, removes the active tool table, and prints a departing herald to the user.

- PPInput reads the pipe from the Protocol Process and dispatches to routines to handle the message based or the content of the instruction field.

Module user.h contains definitions of structure ancd constants global to the UP.

2.2.5.2 USERV

The module userv.c contains global data declarations for the User Process. Module userv.h contains external references to all variables declared in userv.c.

2.2.5.3 UCHAR

The module uchar.c implements the special-function characters of the Front End command language and does high-level parse functions, such as gathering a token.

CmdPars reads one character at a time from the character buffer. It ignores null characters, implements the special functionality for linefeed (linefeed means ´display next completed output´ if there is any), or if there is no output to be displayed, converts linefeed to carriage-return. Other characters are handled by dispatching to routines for each class of character.

The C functions which handle each class of character are:

AbortCh	handles control-X, the "abort" character.
RetypCh	handles control-R, the "retype command" character.
ErswdCh	handles control-W, the "delete field" character.
ErschCh	handles DEL, the "delete character" character.
HlprqCh	handles ?, the "help request" character.
EscapCh	handles ESCAPE, the "recognize and complete" or "prompt request" character.
TcCh	handles SPACE and TAB, the "terminate field" characters.
CcCh	handles CR, the "command complete" character.
McCh	handle , (comma), the "begin command modifications" character.
OtherCh	handles all other characters.

2.2.5.4 UCMD

The module ucmd.c contains the following principal C functions:

- o DescCmd implements the DESCRIBE command. The DESCRIBE command accesses the Front End text file <u>fe-text</u> by means of an index file (<u>fe-txtindex</u>) and copies the relevant part of the text file to the user's terminal. The first call of DescCmd causes the index file to be read in and stored in a buffer. All subsequent calls to DescCmd use the in-core index to find the proper place in the text file. The UNIX pathnames of the FE test file and text index file are parameters in the Front End initialization file.

- PrintUS causes the text contents of a user session table entry (US - see Section 3.2.4.2 below) to be displayed to the user. This requires that the associated terminal buffer headers (TB - see Section 3.2.4.3 below) be scanned to determine the display characteristics of the information buffered. In particular, the contents of a terminal buffer associated with TB entry may be the parameters or arguments of NSWTP Null reply messages, FE-PREDISPLAY messages, Help reply messages, or NSW Error messages. All or part of the fourth arguments of these NSWTP messages are transmitted by pipe from the PP to the UP, where they are buffered, each in conjunction with a TB entry, in their unparsed NSWTP form. This eliminates unnecessary message parsing in both the PP and UP.
- WrtCSstr prints to the user's terminal a character string stored in NSWB8 form.
- CvCmdNo is a general routine that accepts a Front End command number and returns the pointer to the associated US entry. It implements handling of the default case, i.e., a negative command number returns the first US entry on the list.
- CNumCmd implements several user commands which take an Front End command number as their argument. The commands are the DISPLAY, DISCARD, and ABORT commands (though ABORT is not implemented).

2.2.5.5 UDC_TB

The module udc_tb.c contains C functions that manage the terminal buffer header data class (TB). The functions are AddTB, AllocTB, and RemovTB.

2.2.5.6 UDC_TQ

The module udc_tq.c contains the C function AddTQ, which adds an entry to the TP message queue. The queue is filled when a TP instance is in the process of being started; pipe message are buffered in the queue until the TP is ready.

2.2.5.7 UDC_UM

The module udc_um.c contains the C functions which manage the user message buffer (UM), an unstructured buffer which holds user messages not related to US entries (i.e., not related to currently active commands). AddToUM appends its argument to the present contents of UM, and PrintUM prints the contents of UM and empties it.

2.2.5.8 UDC_US

The module udc_us.c contains functions which manage the user session table data class (US - see Section 3.2.4.2 below).

- o AddUS allocates and partially fills a new US entry.
- o RemovUS removes a US entry.
- o StateUS is used to generate user notification messages based upon the contents of the two status words us stat and us umes. In some cases, such as short command output or error output, it displays the command output by calling PrintUS.
- o ShowUS scans all US entries and prints to the user the current state of all active commands. It implements the SHOW ACTIVE COMMANDS command.
- o DumpUS is called by ShowUS and decodes the contents of the two US status words in a form suitable for the SHOW ACTIVE COMMANDS command.

2.2.5.9 UHELP

The module uhelp.c deals with processing the Help reply terminal context, which is entered to answer NSW Help calls. It contains C functions:

- o ProcHlp takes a completed Help reply supplied by the user, assembles it into NSWTP message form, sends it to the PP, and (if the Help message was answered out of tool mode) returns the user to the Conversational Partner he was talking to. The user's completed Help reply is accessed through the parse structure (PS - see Section 3.2.4.1 below).

- IniHelp initializes the UP to accept the user's Help reply. It is invoked by PrintUS after printing out the Help request message to the user. IniHelp, using table ghelp, sets the parser to use the proper section of the "help tree" (see Section 3.2.4.6 below) to parse the user's Help reply. The terminal context is set to 'Help reply' context.

2.2.5.10 UINIT

The module uinit.c contains C functions which perform initialization functions for the UP and entire Front End. The functions are:

- IniTMod saves the terminal modes in effect upon entering the Front End and sets the Front End terminal modes.
- RdIniFl reads the Front End initialization file and converts numeric strings into numbers. The entire initialization file is read in, and UP event-logging flags and typeout flags are set.
- RdIniLn reads in a process initialization line of the Front end initialization file; it is called by RdIniFl.
- CreatPP creates the PP (i.e., it forks and overlays the child with the PP). It does not block. Communication with the PP is confirmed by WaitPP.
- WaitPP is called later in the UP initialization procedure (after CreatPP) has been called. It waits for the PP to set a 'name of tid file' (P_TIDFL) pipe message to the UP, which constitutes confirmation that the PP has been initialized. Until this pipe message arrives, WaitPP blocks. When the message arrives, a state variable is modified and a 'name of tool table file' (P_ATTFL) pipe message is sent to the PP.
- CreatTP creates the TP by forking and overlaying the child process with the TP. It does not block.

Communication with the TP is confirmed upon receipt of an 'init' (P_INIT) message from the TP.

- InitSig initializes the interrupt handlers for the UNIX Interrupt and Quit signals (sent by control-X and control-T, respectively). The other Front End processes ignore these signals.

2.2.5.11 ULKP

The module ulkp.c contains C functions which perform lookup functions in the grammar tree when parsing user input. These functions are:

- TLookup searches through the PY entries (parse lookup entries) for the given GT node and tries to match the user's token with all or part of a string in a PY entry. If a syntactic item supports run-time recognition and completion, a table in the py list form is created for the item and a pointer to it is placed into the grammar tree PY entry before it is scanned. TLookup returns a pointer to a completion string for the item, if it is found. This completion string is typed to the user and is buffered if the user has requested the FE complete his token.
- PrtLst functions in much the same way as TLookup, except that all strings in the PY entries whose initial strings match the user's token are typed out. This function largely implements the "?" (Help Request) character functionality. It supports run-time recognition and completion, but does not return a pointer to a completion string.
- SameStr is the basic string comparison function used in parse table lookup. It terminates on null characters and returns the number of matching characters as well as a boolean which indicates whether the whole string was matched or not.

2.2.5.12 UMESG

The module umesg.c contains the C function to "execute" the command issued by the user, once they have been parsed. These functions are:

- ProcCmd determines the command code for the user's command (which is stored in the grammar tree), executes the command directly if it is "local" (by calling LoclCmd), or otherwise adds a US entry for the command, sets its stream id, and then calls functions to assemble and send the proper NSWTP message(s). Logout with the 'move' modification is implemented as a WM-LOGOUT message, followed by a WM-LOGIN message. Fastlogout is implemented as a 'do Fastlogout' (P_FASTL) pipe message to the PP. All other cases are handled by MakRmMs.

- LoclCmd executes "local" commands, i.e., those directly executable by the Front End. It returns the boolean remot whose value indicates whether a remote action was initiated or not. The value of remot is used when the Front End is in deferred return mode to determine whether control should be returned to the user or not.

2.2.5.13 UMUTIL

The module umutil.c includes the following principal C functions dealing with assembling NSWTP messages and processing user commands dealing with NSW tools:

- AdInLst copies and translates into NSWTP form the list of integers in the parse structure entry pointed to by an input argument.

- AdCSLst does the corresponding operation for character strings (they are placed into NSWTP Character String form).

- AdPCDPs, based on a pointer to the parse structure, copies and translates the following five user tokens into a Physical Copy Descriptor (PCD), followed by a password. This is used by the NET command.

- IntrpPS was referred to under ProcCmd above. It scans the PS table until a PS entry is found which points to the command code in the grammar tree, which the user has entered. If there are PS entries corresponding to subcommands, these are scanned to verify that the command code is correct, or to find a new command code based on the subcommands entered

- ReruCmd assembles a WM-RERUNTOOL message and modifies the active tool table accordingly.

- StdInvM is a utility function to add the procedure name, argument List count, and sessid (session id) to a standard NSWTP type 1 message.

- RerunTl processes the RESUME command. It either causes a switch from exec mode to tool mode and communication with the specified Conversational Partner or, by calling ReruCmd, causes a WM-RERUNTOOL message to be assembled.

- AbortTl process the QUIT ABORT command. It sends TOOLSIG to the TP. If the Conversational Partner to be aborted is an NSW tool, a tool abort scenario is initiated; otherwise, the Conversational Partner (TELNET connection or UNIX subshell) is terminated.

- TermTl processes the QUIT TERMINATE, TERMINATE, QUIT SHELL, and QUIT TELNET commands. If the Conversational Partner to be terminated is an NSW tool, a tool terminate scenario is initiated; otherwise, the Conversational Partner (TELNET connection or UNIX subshell) is terminated.

2.2.5.14 UPARAM

The module uparam.c contains C functions used in parsing parameters used in the command language.

Param accepts the parameter token and calls routines to parse it depending on its expected type. The Front End currently supports the following kinds of parameters: the null parameter, integers (to 32 bits), ARPANET host name or number, NSW filespec or filespec number, and arbitrary character strings. ParsInt parses integers, and Parsfil parses NSW file specifiers. NSWFile parses NSW file specifiers, which ParsFil uses to determine if a string is a plausible NSW file specifier. NSWStr verifies that the argument string contains only upper- or lower-case alphabetics, numerics, the underline, or the hyphen character; this is the character set, plus ´.´, to which NSW file specifiers are limited.

2.2.5.15 UPARS

The module upars.c, in conjunction with uchar.c, contains the main parser functionality of the Front End. The principal C functions are:

- o ParsWrd attempts to parse the token to which the PS pointer points. If the token is to be looked up in the grammar tree, TLookup is called to do it; otherwise, the token is parsed as a parameter. This routine also verifies that the command being entered may legally be entered at that time, e.g., LOGOUT may not be issued unless the user is already logged in.

- o AdvPars moves the GT pointer to the next GT node in the grammar tree, once the current token has been parsed. It also adjusts terminal modes for the new token to be entered.

- o PrtProm prints prompts to the user when the user requests them or circumstances require it.

- o IniPars (re-)initializes the Front End to begin a completely new parse of user input. It is called, for example, when the user types control-X.

- o TypeBuf implements the control-R "retype command" function. It scans the PS table and outputs all tokens and buffered prompts to the user.

2.2.5.16 UPIPE

The module upipe.c contains the C functions which may be invoked by the dispatch in PPInput in user.c. These are:

IHelpM	handle NSW Help message.
IErDsM	handle NSW Error message.
INullM	handle Null reply message.
IPdspM	handle FE-PREDISPLAY message.
IScDonM	protocol scenario is done.

INewScM	new protocol scenario initiated remotely.

This module also reads the pipe from the Tool Process and dispatches to routines to handle the message based on the content of the message instruction field. There routines are:

ISuspM	User has suspended use of a Conversational Partner; control-N was typed to TP.
NAcTelM	A TELNET or subshell activation failed.
ICloseM	A Conversational Partner closed.
AAcTelM	A TELNET or subshell activation succeeded.
AFinisM	An NSW tool-ending sequence succeeded.
AFnTelM	A TELNET or subshell termination succeeded.
IInitM	The TP has initialized and is ready; causes contents of TQ to be dumped to the pipe to the TP.
NResumM	A Conversational Partner cannot be resumed; buffered output, if any, was displayed to the user by the TP.
NActivM	A NSW tool-activation failed.
AActivM	A NSW tool-activation succeeded.
ANHelpM	Answer Help message for current NSW tool.

IErrorM handles pipe error messages from both the PP and TP.

2.2.5.17 UPROC

The module uproc.c contains C functions which display or create status information to or for the user. For example, the function which processes the control-T Status Query character is in this module.

2.2.5.18 UPUTIL

The module uputil.c contains utility routines for the Front End parser. The principal C functions are:

- o AddStr adds one or more characters to the user's terminal buffer and may also display them on the user's terminal.

- o AddPS adds a new parse node to the PS table. If it is a node for the first token of a subcommand, it is linked to the first token of the previous subcommand, if there is one, and otherwise to the PS entry for the command verb (the very first token); see Section 3.2.4.1 for detailed description.

- o RemWord implements the control-W "remove field" character. RemovPS is called by RemWord. It removes a PS node from the PS table. If there are links to command modification PS nodes, the links are broken.

- o AdjTMod adjusts the FE terminal break classes, depending on the current value of the GT pointer. The Front End runs with two terminal break classes (in exec mode): (1) every Ascii character is a break; (2) alphabetics, numerics, balanced delimiters, and some others are nonbreaks. The former set is in effect when the command buffer is empty and after a complete command has been accepted by the Front End (but before the final confirming CR has been typed). The latter set is in effect at all other times.

- o MakTabl and RemTabl, respectively, create and remove run-time parse lookup lists to support run-time recognition and completion [for syntactic items whose possible number of strings are small at any one time, but which cannot be predicted in advance; e.g. the names of Conversational Partner].

2.2.5.19 URMES

The module urmes.c contains the following C modules which handle NSWTP messages.

- o MakRmMs assembles an NSWTP message, based on its input arguments and the contents of the PS table.

- SndRmMs sends a 'initiate protocol scenario' (P_INISC) pipe message to the PP, followed by the NSWTP message.

- AbtRmMs resets the appropriate data structures and removes the message buffer if the message is to be aborted.

2.2.5.20 USIG

The module usig.c contains C functions which intercept UNIX signals in the UP:

- Sig_Qit intercepts the 'Quit' signal, which is sent by typing control-T. This function implements calling the Status Query typeout and allows for quoting control-T.

- Sig_Itr intercepts the 'Interrupt' signal, which is sent by typing control-X. Invoked in deferred return mode, this function causes all commands issued in deferred mode to be treated as if issued in immediate return mode; this is done by clearing US_DEFER in the command's US entry.

- Hup_UP intercepts the 'Hangup' signal when it is sent to the UP. The Hangup signal is sent when the user loses contact with his UNIX processes, e.g., when a TELNET connection to the Front End breaks. If the user is logged in, all outstanding Help calls are aborted; otherwise, the Front End is stopped. A state variable is also modified.

2.2.5.21 UTERM

The module uterm.c contains C functions which implement terminal I/O in the UP. These include:

- TermInt reads from the terminal, based on the number of characters to be read, as determined from the capac system call. Characters are read into an intermediate buffer.

- ChInput takes the characters from the intermediate buffer, calls CmdPars to parse them, and if they define a complete command, executes the command.

- o TermOut performs unsolicited terminal output to the user, i.e., command notification and user messages. Output is performed only if the user is not typing in data to the terminal and only if the information for a given US entry is ready (US_ANNOU and US_NOANN are off).

2.2.5.22 UTOOL

The module utool.c contains C functions which handle

Conversational Partners:

- o TelnCmd implements the TELNET command. It adds an entry to the US table (since the command is considered to initiate a remote action), adds an entry to the tool table, and sends an ´activate TELNET´ (P_ACTEL) pipe message to the TP. If the TP has not yet been initialized, the pipe message is buffered in TQ.

- o QTelCmd implements the TERMINATE (et. al) command for TELNET connections and UNIX subshells. It modifies a state variable in the TT entry of the Conversational Partner and sends a ´end use of conn/shell´ (P_FNTEL) pipe message.

- o ShowTT implements the SHOW ACTIVE TOOLS command. It scans the tool table and prints out a listing of the names and status of all active Conversational Partners.

- o LkpTool looks up in the active tool table a Conversational Partner specified by the user. The Conversational Partner is guaranteed to exist, since it was previously recognized by the parser (see RcgTool below).

- o RcgTool scans the active tool table and creates an in-core table of the names of Conversational Partners in a format compatible with that of the parse table lookup routines (i.e., TLookup and PrtLst - see Section 2.2.5.11 above). This function implements run-time recognition and completion for the names of Conversational Partners.

- o U_ChgTE effects a change of process mode from tool mode to exec mode. It performs the inverse functions of those done by U_ChgET and resets terminal modes.

- o SlewTl invokes a change from exec mode to tool mode, for the Conversational Partner specified by the user. The procedure is: lookup the Conversational Partner in the active tool table; if there is an outstanding Help call for this Conversational Partner, go process it; otherwise, change to tool mode and send a ´resume´ (P_RESUM) pipe message to the TP.

- o CntTool scans the active tool table and counts the number of entries of the same type as that type specified by the argument to CntTool.

2.2.5.23 UUTIL

The module uutil.c contains C functions which are utilities used by the UP. Notably, StopTP and FEStop, respectively, stop the TP and entire Front End.

2.2.5.24 GTREE

The module gtree.c contains the grammar tree (GT) which defines the UNIX FE command language. Module gtree.h defines the structure of GT (refer to Section 3.2.4.5 below) and defines constants used in the table.

2.2.5.25 HTREE

The module htree.c contains the "help tree", which is a table of structure GT, but which defines the acceptable Front End user inputs when the user is answering a Help call. Module htree.h defines names for NSW help codes.

2.2.6 Management of Conversational Partners

This section describes program modules which specifically deal with handling Conversational Partners. All of the modules access and/or modify the active tool table, and all but AddTT are used by all three Front End processes.

2.2.6.1 ATT

The module att.c contains C functions which access the active tool table, a file accessed by all three Front End processes. These functions are:

- OpenTT opens the active tool table. All tool table access is done in exclusive-access read-write mode. OpenTT makes five attempts, separated by one-second sleep calls, before giving up. If the open succeeds, the file pointer is set to that specified by the argument. If the argument is negative, the file pointer is set to 2, the beginning of the first TT entry.
- ReadTT reads a tool table entry into the location pointed by its argument. The file pointer is not modified beforehand.
- WriteTT writes the tool table entry pointed to by its second argument to the file position specified by the first argument. If the tool table is not open, it is opened first.
- CloseTT closes the active tool table.

The module att.h contains the definition of a TT entry and defined constants used in referring to an entry. Refer to Section 3.2.5.1 below for details.

2.2.6.2 ADTOOL

The module adtool.c contains C function AddTT, which adds an entry to the active tool table. The entry will be added in the first vacant position of the table, or appended to the end of the file if no entries are vacant.

2.2.6.3 FNTOOL

The module fntool.c contains C functions that search for particular TT entries or delete particular TT entries. Deleting an entry consists of setting the entry's toolid field to zero. Those C functions that search for TT entries leave the table open on exit and return the entry found in the location pointed to by an input argument. Those C functions that remove TT entries close the table on exit.

The functions are:

MATTPtr	find the entry at the specified file position
MToolID	find the entry with the specified tool id
MToolNm	find the entry with the specified tool name
MProcNm	find the entry with the specified MSG process name
RATTPtr	remove the entry at the specified file position
RToolId	remove the entry with the specified tool id
RToolNm	remove the entry with the specified tool name
RProcNm	remove the entry with the specified MSG process name

2.2.7 Message Handling

This section describes program modules which specifically manipulate NSWTP messages and NSWB8 elements. Except for mwrite.c (which only the PP uses), the program modules are used by both the PP and UP.

2.2.7.1 MPARS

The module mpars.c contains C functions that parse NSWTP messages on an element-by-element basis. The parse results are placed into a structure mp:

```
/* structure to parse incoming NSWTP messages */

struct mp
{
   int mp_val;  /* value of argument, or ptr to it */
   int mp_cnt;  /* size of argument, where relevant */
};
```

Below we give with each C function what they return i mp val and mp cnt. Each function returns a pointer to the slot in the message buffer following that of the NSWB8 element just parsed.

GetBool	parses NSWB8 boolean; return boolean value in mp val
GetIndx	parses NSWB8 index; return index value in mp val
GetIntg	parses NSWB8 integer; return pointer to first byte of integer in mp val; must call CnvIntg afterwards
GetBStr	parses NSWB8 bit string; return pointer to string in mp val, string count in mp cnt
GetCStr	parses NSWB8 character string; return pointer to string in mp val, string count in mp cnt
GetList	parses NSWB8 List header; return List found in mp val

2.2.7.2 MPRT

The module mprt.c contains C functions that print out NSWB8 elements in human-readable form and functions that manipulate NSWTP messages.

The functions that print NSWB8 elements utilize a global stack of size MSTKSIZ and a global stock pointer. The value of the stackpointer (msub, an array subscript) specifies the depth of nesting of the current List, and the value of the stack element pointed to is the remaining List count for the current List. The functions in question are:

TypeMes	is the top-level message scanner and does some formatting of the output. It calls the Prt... routines.
Pop	decrements the List count for each NSWB8 element printed. When it reaches zero, it decrements the stack pointer.
PrtBool	prints a NSWB8 boolean.
PrtIndx	prints a NSWB8 index.
PrtIntg	prints a NSWB8 integer.
PrtBStr	prints a NSWB8 bit string.
PrtCStr	prints a NSWB8 character string.
PrtList	prints a NSWB8 List header. The stack pointer is incremented, and the stack element is loaded with the List count.

The remaining functions are:

CopyLst	copies an NSWB8 List from the position of origin argument to that of the destination argument.
SkipLst	skips a NSWB8 List beginning at its argument.

2.2.7.3 MUTIL

The module mutil.c contains C functions that add NSWTP message elements or NSWB8 elements to a target buffer.

Generally, the functions return a pointer to the location following the section of the target buffer loaded. NSWB8 elements are assembled in byte-reversed order relative to PDP-11 addressing conventions. The functions are:

AddTid	adds a transaction identifier to the message buffer.
AddType	adds a NSWTP type to the message buffer.
AddPNam	adds a procedure name to the message buffer.
AdNxPar	adds a NSWB8 element to the message buffer; it calls the following functions.
PutBool	adds a NSWB8 boolean.
PutIndx	adds a NSWB8 index.
PutIntg	adds a NSWB8 integer.
PutBStr	adds a bit string.
PutCstr	add a NSWB8 character string.
BegList	adds a NSWB8 list header

2.2.7.4 MWRITE

The module mwrite.c contains C functions that write NSWB8 elements to an output file. These functions are used by the PP to assemble the arguments for WM-FASTLOGOUT and WM-AUTOLOGOUT in an intermediate file. The functions are:

WrtBool	writes a NSWB8 boolean.
WrtIndx	writes a NSWB8 integer.
WrtBStr	writes a NSWB8 bit string.
WrtCStr	writes a NSWB8 character string.
WrtList	writes a NSWB8 List header.

WrtAny   writes a NSWB8 element; it calls the preceding.

2.2.8 Data Storage Management

As described in Section 3.2.1, a data class is the name used to refer to doubly-linked lists of dynamically allocatable table entries. This section describes the program modules which manage the Front End data classes.

In the PP and TP, the table entries are dynamically allocated in the sense of requesting space from the UNIX memory space allocator by making alloc system calls. In the UP, the table entries consists of moving the entries defined at compile-time from the free list to the end of the ´in use´ list.

The state of each data class is represented by a class header; this header is altered by the management routines to reflect the current state of the data class.

2.2.8.1 ALLOC

The module alloc.c contains the following principal C functions that perform data storage management in the UP:

- Initlze initializes the class header for a data class and links together the table entries to form the free list. The ´in use´ list is empty.
- Allocat allocates an entry of the data class. It does so by moving the entry from the free list to the end of the ´in use´ list.
- Dealloc deallocates an entry of the data class. The entry is removed from the ´in use´ list and placed at the end of the free list.

- o NextEnt returns a pointer to the next entry of the data class, given a pointer to the current entry. A null argument causes it to return the pointer to first entry in use.
- o ChkEntr is called by Dealloc and NextEnt and verifies that the argument is a valid pointer to an entry of the data class.

2.2.8.2 ALLOCN

The module allocn.c contains C functions that perform data storage management for the TP and PP. It uses the same function names as those in alloc.c, but their operation is different; the principal functions are:

- o Initlze initializes the class header for a data class. Any entries not already removed are freed (using the UNIX free system call).
- o Allocat allocates an entry for a data class. It does so by calling the UNIX alloc system call and linking the returned block to existing entries.
- o Dealloc deallocates an entry of a data class. It does so by using the UNIX free system call.
- o NextEnt returns a pointer to the next entry of the data class, given a pointer to the current entry, as in alloc.c.
- o ChkEntr, as in alloc.c, verifies that the current pointer argument is a valid one.

2.2.9 Logging Functions

This section describes the program module which implements the logging features of the Front End described below in Section 4.4.

WrtAny   writes a NSWB8 element; it calls the preceding.

2.2.8 Data Storage Management

As described in Section 3.2.1, a data class is the name used to refer to doubly-linked lists of dynamically allocatable table entries. This section describes the program modules which manage the Front End data classes.

In the PP and TP, the table entries are dynamically allocated in the sense of requesting space from the UNIX memory space allocator by making alloc system calls. In the UP, the table entries consists of moving the entries defined at compile-time from the free list to the end of the 'in use' list.

The state of each data class is represented by a class header; this header is altered by the management routines to reflect the current state of the data class.

2.2.8.1 ALLOC

The module alloc.c contains the following principal C functions that perform data storage management in the UP:

- o Initlze initializes the class header for a data class and links together the table entries to form the free list. The 'in use' list is empty.
- o Allocat allocates an entry of the data class. It does so by moving the entry from the free list to the end of the 'in use' list.
- o Dealloc deallocates an entry of the data class. The entry is removed from the 'in use' list and placed at the end of the free list.

- o NextEnt returns a pointer to the next entry of the data class, given a pointer to the current entry. A null argument causes it to return the pointer to first entry in use.
- o ChkEntr is called by Dealloc and NextEnt and verifies that the argument is a valid pointer to an entry of the data class.

2.2.8.2 ALLOCN

The module allocn.c contains C functions that perform data storage management for the TP and PP. It uses the same function names as those in alloc.c, but their operation is different; the principal functions are:

- o Initlze initializes the class header for a data class. Any entries not already removed are freed (using the UNIX free system call).
- o Allocat allocates an entry for a data class. It does so by calling the UNIX alloc system call and linking the returned block to existing entries.
- o Dealloc deallocates an entry of a data class. It does so by using the UNIX free system call.
- o NextEnt returns a pointer to the next entry of the data class, given a pointer to the current entry, as in alloc.c.
- o ChkEntr, as in alloc.c, verifies that the current pointer argument is a valid one.

2.2.9 Logging Functions

This section describes the program module which implements the logging features of the Front End described below in Section 4.4.

2.2.9.1 LOG

The module log.c contains the following principal C functions:

- o InitLog initializes a process for event-logging. The event log file is created, and an initial record is written out to it. The file stays open throughout the Front End session.
- o LogEvnt is called to log a Front End process event. It writes out a record to the event log file based on the arguments given to it. The module log.h contains the structure of the non-Ascii part of each event log record.
- o OpnLgFl opens an error log file and sets the file pointer to the end of the file.
- o LogErr writes a line to the error log file. If LOG_TTY is set, the same line is written to the user's terminal also.

2.2.10 Inter-Process Communication

This section describes the program modules used to implement inter-process communication in the Front End.

2.2.10.1 IPC

The module ipc.h is a C header file in which the common structure of all pipe message transactions (IP - see Section 3.1.1.3 below) is defined. The auxiliary message block for 'initiate scenario' (P_INISC) pipe messages is defined, and the names for the contents of the instruction field are defined.

2.2.10.2 PIPE

The module pipe.c contains C functions which perform pipe I/O in all three Front End processes. The functions are:

WriteIP	write a pipe header (an IP block) to a pipe.
PrintIP	if VRB_IPC is turned on, PrintIP is called to print out the contents of the pipe header to the terminal; called from WriteIP and ReadIP.
LogIP	If LOG_IPC is turned on, log the pipe message in the event log file; called from WriteIP and ReadIP.
ReadIP	Read a pipe header from a pipe.

2.2.11 Utility Routines

The module libfe.a is a file in UNIX library format and contains the following C utility functions:

AdToBuf	copies bytes from origin to destination locations; arguments are addresses.
AToInt	performs Ascii to (16-bit) integer conversion, accepts a base argument.
ClrByte	clears bytes beginning at origin location, ending at destination location or with null byte count.
Cnvlntg	Converts NSWTP integer to UNIX <u>long</u> integer.
ConvLC	Converts upper-case alphabetics to lower-case.
ConvUC	Converts lower-case alphabetics to upper-case.
CopyFil	Copies contents of input file to output file; neither opens nor closes either file.
CopyByt	Copies bytes from origin to destination locations; one argument is a count.
DoAlloc	Attempt to allocate space; handle errors if failed.
ErrCond	Send an error message to the UP; then abort or exit from FE, or continue.
FlushFd	Read characters from a file descriptor, especially to keep pipes clear.
InitPB	Initialize process control block.

IntrpCh	Print graphic interpretation of special characters, octal representation of other characters.
LdVStr	Create space for string; copy string into new space; store pointer to new string.
MkErDes	Assemble an NSWTP error descriptor.
MkFMEnd	Make fourth argument of FM-ENDTOOL.
MkHstNm	Make a special host name string.
NToA	Convert an integer to Ascii; accepts a base argument.
ProcArg	Handle process arguments at initialization.
PrtLong	Print a UNIX _long_ in octal.
RdBytes	Read bytes into buffer; is uninterruptible.

2.2.12 Front End Initialization Process

The module nswfe.c is the Front End initialization process (see Section 2.1.1 above). It reads in the pathname of the UP from the Front End initialization file and overlays the UP by calling _execl_.

3. INPUT/OUTPUT DESCRIPTIONS

This chapter provides detailed information on the structure and composition of data used by the UNIX Front End implementation.

3.1 General Description

3.1.1 Input/Output

The Front End is a "system program" which functions more like an operating system than like a language processor, such as a compiler, or a scientific program, such as a statistics package. Therefore, the Front End implementation does not deal with input/output in the "traditional" sense of inputs and processing it to produce a set of outputs.

However, in another sense (see "FE Flow of Control" Section 2.1.4 above), the user input (UNIX Front End command language) may be viewed as an input which produces an output (an NSWTP message or some local action), and a received NSWTP message, or input received from a Conversational Partner, may be viewed as an input to produce output to the user. Refer to the UNIX Front End User Manual for a complete description of the Front End command language and patterns of interaction between the user and the Front End.

The Front End accepts input at startup time from the Front

End initialization file which defines the characteristics of the Front End configuration to be run.

The Front End text file is read to retrieve selected pieces of text to be displayed to the user; this file is indexed by the Front End text index file.

3.1.1.1 Front End Initialization File

The user supplies the UNIX pathname of the Front End initialization file as an argument to the Front End initialization process, or if not supplied, the initialization process will use <u>fe-init</u> as the default file name.

For each Front End process, the initialization file contains its UNIX pathname and values for the flags governing event-logging and diagnostic typeout. Turning on event-logging for a Front End process will cause an event log output file to be generated. Turning on diagnostic typeouts for a Front End process causes extra terminal output to be generated. In addition, the initialization file contains the UNIX pathnames for the Front End text file and text index file (see below). Refer to the UNIX Front End User Manual for a detailed description of the initialization file; see also Sections 4.3 and 4.4 below.

3.1.1.2 Front End Text File

The Front End text file is an Ascii file containing the name of the item to be referenced, followed by the text to be displayed to the user when that item is referenced. Items to be referenced are enclosed in double # signs, to delimit both the items and their text. For example:

```
##item1##
Here is text for item1.
##item2##
Text for item2.
```

and so on.

The Front End text index file has, for each item, the following format:

```
-------------------------------------------------------------
| file ptr | #chars in text of item | item name | null char |
-------------------------------------------------------------
      2                  2                N           1
```

It is not an Ascii file. The byte counts for each field are shown below each field.

3.1.1.3 Inter-Process Communication

Front End processes communicate with each other over UNIX pipes. Each message transaction consists of a pipe header, which is always sent, plus zero or more additional transmissions associated with the instruction field in the pipe header. The structure of a pipe header (structure ip) is described below. The synonyms for the arguments are the mnemonic names used in the source code for the arguments of the pipe header.

```
/* inter-process communication structures */

struct ip
{
   int ip_inst;      /* instruction field */
   int ip_arg1;      /* first argument */
   int ip_arg2;      /* second argument */
   int ip_arg3;      /* third argument */
   int ip_arg4;      /* fourth argument */
   int ip_arg5;      /* fifth argument */
};


/* synonyms for IPC fields */

#define ip_aspt   ip_arg1 /* ptr to AS table */
#define ip_attp   ip_arg2 /* ptr to tool table file */
#define ip_uspt   ip_arg3 /* ptr to user session table */
#define ip_scen   ip_arg3 /* FE ptcl scenario number */
#define ip_usnx   ip_arg4 /* ptr to US entry for next scen */
#define ip_pcod   ip_arg4 /* value of us_pcod for scen */
#define ip_erno   ip_arg4 /* value of errno */
#define ip_topt   ip_arg4 /* ptr TO entry */
#define ip_hnum   ip_arg4 /* host number (long) */
#define ip_fskt   ip_arg4 /* foreign socket # (long) */
#define ip_usid   ip_arg4 /* user id (long) */
#define ip_mlen   ip_arg5 /* length of following message */
#define ip_tipt   ip_arg5 /* ptr to tool buffer header */
```

The octal values of the instruction field are less than 0400. If a pipe message generates a positive acknowledgement from the receiving process, an octal 0400 is OR-ed into the instruction field of the returning message. If a pipe message generates a negative acknowledgement from the receiving process, an octal 01000 is OR-ed into the instruction field of the returning message.

For the 'initiate new scenario' (P_INISC) pipe message, the pipe header is followed by an auxiliary block of the following structure:

```
/* structure for auxiliary info for P_INISC messages */

struct pi
{
   int pi_sid;      /* sid */
   int pi_stid;     /* stream id for command */
   int pi_scen;     /* FE ptcl scenario number */
   int pi_pcod;     /* value of us_pcod in US entry */
   int pi_mlen;     /* length of following message */
};
```

In both structures, the fields are declared as <u>ints</u>; however, the fields of ip contain various kinds of pointers, as well as <u>ints</u>.

3.1.1.4 Signals

The Front End uses UNIX signals for communication among the Front End processes on an interrupt basis. The signals used by Front End processes are:

- o Hangup: caught by all three Front End processes. It signals the start of an Autologout condition, i.e., each process will take steps to facilitate the automatic logout of the Front End from NSW.
- o Interrupt: caught by the UP and sent by typing the "abort input" character control-X. The signal handler implements the functionality of control-X. The PP and TP ignore the interrupt signal.
- o Quit: caught by the UP and sent by typing the Status Query character control-T. The signal handler implements the functionality of control-T. The PP and TP ignore the Quit signal.
- o INR/INS: caught by the User TELNET package in the TP. The signal handler (invisible to the Front End) handles the signal, which marks the arrival of an NCP host-to-host interrupt. The PP and UP ignore the INR/INS signal.

- o LPDSIG Signal 16 (Local Process Dead): this user-defined signal is sent by the UP to the PP and TP to force them to terminate immediately. It is sent when a catastrophic error in the Front End is detected. The PP and TP die when the signal is received.
- o TOOLSIG Signal 17 (Tool): this user-defined signal is sent by the UP to the TP when the user issues a QUIT ABORT command. It is caught by the TP and has the effect of interrupting an <u>open</u> which may not have returned.

All other UNIX signals have their usual effect.

3.1.1.5 Terminal I/O

The terminal modes and break character sets used by the UP are defined in IniTMod in uinit.c and as ´FE_BREAK´; global variable <u>breaks</u> has the value ´FE_BREAKS´). The terminal modes and break character sets used by the TP are defined in IniMBlk in tutil.c and in DoRmEch in telnio.c. The <u>modtty</u> system call is used to control all FE terminal modes. Consult ´modtty (II)´ in [6] for further details.

Terminal input is done by using the <u>await</u> and <u>capac</u> system calls. The process reading from the terminal await-enables it using the <u>awtenb</u> system call; this makes terminal input trigger a wakeup on an <u>await</u>. This function is disabled by using <u>awtdis</u> system call.

When an <u>await</u> wakeup occurs, the <u>capac</u> system call is ready to be read from the terminal. <u>capac</u> always returns a count of the first set of nonbreak characters plus the following break character as its value. This is true even if there is more than one break character in the terminal input queue. Thus, any terminal read is guaranteed to terminate with a break character. If the terminal input queue contains nonbreak characters but no break characters, capac will return 0.

The M_ECHO modtty control order, which is executed by the C function EchoChr, is required because the Front End operates in deferred echo mode. Were it not present, the nonbreak characters typed by the user could echo prematurely. The M_ECHO function tells the UNIX terminal handler to echo the next set of nonbreak characters in the terminal input queue.

Terminal output is straightforward. If flag itrflag is FALSE, the user has typed control-X at some point; all terminal output until the next prompt is suppressed, and itrflag reset to TRUE.

The transfer of control of the terminal (in process mode changes: exec mode to tool mode, or vice versa) is accomplished by awtdising the terminal in the process losing control of it, and then awtenbing it in the process gaining control of it. Associated FE process variables are also changed.

3.1.2 Internal Front End Data

To support its operation the Front End maintains two kinds of internal data:

- o Static data items are those that do not change as a result of Front End execution after a process has been initialized. Included are such items as PT (Section 3.2.2.4) in the PP, and the grammar tree and Help tree (Sections 3.2.4.5 and 3.2.4.6) in the UP.

- Dynamic data items are those changed as a result of Front End executions. Included are all data classes of all Front End processes and all Front End state and context variables.

3.2 Data Structures

This section describes the structures and data used in the Front End implementation.

3.2.1 General

The term <u>data class</u>, or <u>class</u>, as used in this document and the Front End source code means a doubly-linked list of dynamically allocatable table entries. The forward and backward list pointers are, respectively, constrained to be the first two words of a table entry. The rest of the table entry may have an arbitrary structure. By referencing the list pointers in the first two words, the data management functions described in Section 2.2.8 above perform their tasks.

As has been noted, the data management functions retain status information about each class in a <u>class header</u>. The class header for PP and TP data classes follows:

```
/* (new) class header */

struct nc
{
   int *nc_beg;     /* ptr to first entry */
   int *nc_end;     /* ptr to last entry */
   int nc_nent;     /* # entries allocated */
   int nc_esiz;     /* size of each entry (in bytes) */
};
```

The structure member nc_esiz contains the size of each entry in bytes and is not altered in the course of process execution.

The class header for the UP is:

```
/* class header */

struct ch
{
   int *ch_fuse;     /* first entry in use */
   int *ch_luse;     /* last entry in use */
   int *ch_ffre;     /* first free entry */
   int *ch_lfre;     /* last free entry */
   int ch_nfre;      /* number of free entries */
   int *ch_limi;     /* limit of entries */
};
```

The first two structure members are pointers, respectively, to the beginning and end of the ´in use´ list. The second two members are pointers, respectively, to the beginning and end of the free list. ch_nfre is a count of the number of free entries left out of the total number pre-defined at compile time; the latter number is found in nument in userv.c.

AD-A105 035 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA F/G 9/2
UNIX NSW FRONT END ENHANCEMENTS. VOLUME I.(U)
JUN 81 R H THOMAS, H O LIND, S G TONER F30602-80-C-0062
UNCLASSIFIED BBN-4571-VOL-1 RADC-TR-81-164-VOL-1 NL

2 of 3
ADA
105 035

3.2.2 Protocol Process

This section describes the principal data structures of the PP.

3.2.2.1 AS - Active Scenario Table Class

For each active NSW protocol scenario, there is an associated AS entry. It contains all information global to the scenario. The structure of an AS entry is:

```
/* ACTIVE SCENARIO TABLE */

struct as
{
   struct as *as_pent;      /* ptr to previous entry */
   struct as *as_nent;      /* ptr to next entry */
   int as_sid;              /* scenario identifier */
   int as_stid;             /* stream id */
   struct as *as_prev;      /* ptr to previous scenario */
   struct as *as_next;      /* ptr to next scenario */
   int as_flag;             /* flag; see definitions below */
   int as_scen;             /* FE ptcl scenario number */
   struct pt *as_ptpt;      /* ptr to current ptcl step */
   struct tc *as_tcpt;      /* ptr to current t-block */
   int *as_uspt;            /* ptr to active session
                               entry in UP */
   int as_attp;             /* file ptr of associated tool */
   int as_pcod;             /* value of us_pcod for scen */
   long as_tmis;            /* time last RcvSpec issued */
};


/* flags used in as_flag */

#define AS_REMOT  01   /* on, if scen remotely initiated */
#define AS_PERR   02   /* primitive in scenario failed */
#define AS_COMPL  04   /* on, if current step has completed */
#define AS_PEND   010  /* on, if current step initiated */
#define AS_SCDON  020  /* scenario completion, waiting for
                          flushing */
#define AS_BLOCK  040  /* temporarily suspend execution
                          of scenario */
#define AS_FASTL  0100 /* scen is part of Fastlogout seq */
#define AS_AUTOL  0200 /* scen is part of Autologout seq */
```

3.2.2.2 TC - Transaction Control Block Class

For each step of a protocol scenario, there is an associated transaction control block. The TC entry contains information specific to the protocol step. Once created, however, a TC entry remains in existence until the protocol scenario has terminated. In many cases, the current TC entry must reference previous TC entries to obtain needed information. The structure of a TC entry, along with mnemonic names for the extra arguments is:

```
/* TRANSACTION BLOCKS */

struct tc
{
  struct tc *tc_pent;     /* ptr to previous entry */
  struct tc *tc_nent:     /* ptr to next entry */
  struct as *tc_aspt;     /* ptr to corresponding AS entry */
  struct pt *tc_ptpt;     /* ptr to previous transaction block;
  struct tc *tc_prev;     /* ptr to previous transaction block;
                             if = NULL, no previous block */
  struct tc *tc_next;     /* ptr to next transaction block;
                             if = NULL, no next block */
  int tc_evnt;            /* event handle returned by MSG */
  int tc_disp;            /* disposition returned by MSG */
  int tc_sig;             /* signal used */
  int tc_type;            /* if message, NSW type; if SendAlarm,
                             alarm code */
  int tc_id;              /* if message, tid/sid of message */
  int tc_timo;            /* timeout, where appropriate */
  struct mb *tc_mbpt;     /* ptr to message buffer hdr; null
                             for ops involving no message */
  int tc_ext1;            /* extra arg 1 */
  int tc_ext2;            /* extra arg 2 */
  int tc_ext3;            /* extra arg 3 */
  char *tc_vtbk;          /* ptr to variable format and variable
                             size t-block extension to hold any
                             other information */
  char tc_proc[MSGPSIZ];  /* MSG process name involved */
};
```

```
/* synonyms for TC fields */

#define tc_acod   tc_type /* alarm code for SendAlarm */
#define tc_fskt   tc_extl /* foreign socket number (long)
#define tc_hand   tc_ext3 /* for SenGen, qwait; for SendSpec
                             and RecvSpec, special handling
                             code */
```

3.2.2.3 MB - Message Buffer Header Class

Associated with each NSWTP message, whether sent or received, is a message buffer header. Each MB entry has the following structure:

```
/* MESSAGE BUFFER HEADERS */

struct mb
{
   struct mb *mb_pent;    /* ptr to previous entry */
   struct mb *mb_nent;    /* ptr to next entry */
   struct tc *mb_tcpt;    /* ptr to corresponding t-block */
   int mb_char;           /* # chars buffered */
   char *mb_mbuf;         /* ptr to message buffer */
};
```

3.2.2.4 PT - Protocol Scenario Definition Table

The protocol scenario definition table is a global data item and is not a data class. Each protocol scenario is defined by a number of protocol scenario steps, each of which is defined by:

```
struct pt
{
   int pt_op;          /* op code */
   int pt_timo;        /* timeout */
   int pt_arg;         /* argument */
};
```

The op code is either that of a MSG primitive or PT_GOTO or PT_ENDSC. If the op code is that of an MSG primitive, the timeout argument is the timeout to be used on the primitive invocation. For SendMessage primitives, the argument (pt arg) indicates the kind of message to be sent. For the SendAlarm primitive, the argument gives the alarm code.

For the ReceiveSpecific primitive, the argument is a pointer to a block of codes defining what messages are allowed to be received for the current step. If the received message matches an expected message, the ReceiveSpecific protocol step returns at an offset from itself determined by the position in the block of codes matching that of the received message. That is, for example, if the received message matches the second code in the argument block pointed to by pt arg, then the ReceiveSpecific protocol step will return two steps past the ReceiveSpecific step. This is called ´conditional branching´ and enables the structure of PT to be fairly simple.

The PT_GOTO op code means to move the PT pointer ahead or back the number of steps given in the argument; the sign of the argument determines the direction of motion.

The PT_ENDSC op code means that the scenario is done and causes a ´scenario done´ pipe message (P_SCDON) to be sent to the UP; also, AS_SCDON is sent in the associated AS entry.

3.2.3 Tool Process

This section describes the principal data structures of the TP.

3.2.3.1 TI - Tool Buffer Header Class

For each active Conversational Partner, there is a TI entry. After a Conversational Partner has closed or been terminated, the TI entry will remain if there is buffered output for the user; otherwise, it will be removed. The structure of each TI entry is:

```
/* TOOL BUFFER HEADERS */

struct ti
{
   struct ti *ti_pent:        /* ptr to previous entry */
   struct ti *ti_nent;        /* ptr to next entry */
   struct NetConn *ti_conp;   /* ptr to connection
                                 control blk */
   int ti_uifd;               /* fd to read from UNIX pipe */
   int ti_uofd;               /* fd to write to UNIX pipe */
   struct to *ti_topt;        /* ptr to corresponding
                                 TO entry */
   int ti_attp;               /* file ptr to corresponding
                                 TT entry in tool table */
   int ti_char;               /* # chars buffered */
   char *ti_tbuf;             /* ptr to tool buffer */
   int *ti_tmod;              /* ptr to terminal modes for conn */
};
```

For network I/O, the connection control block pointer returned by the UNIX User TELNET Package is used, and (ti uifd and ti uofd are ignored. One pointer handles both directions of data transfer.

For pipe I/O with inferior UNIX processes, ti uifd and (ti uofd hold, respectively, the read and write pipe file descriptors; the connection control block pointer is ignored.

3.2.3.2 TO - Tool Connection Information Class

Each TO entry corresponds to one NCP "dummy open" having been performed and keeps track of the Conversational Partners using the block of 8 sockets returned by the open. The structure of each TO entry is:

```
/* Tool Connection Information Blocks */

struct to
{
   struct to *to_pent;     /* ptr to previous entry */
   struct to *to_nent;     /* ptr to next entry */
   int to_bsfd;            /* base fd for block of 8 sockets */
   int to_lskt;            /* socket # of base socket */
   struct ti *to_tipt[4];  /* ptr to corresponding TI entries */
};
```

Refer to ´NCP(IV)´ in [6] for details concerning the NCP "dummy open."

3.2.4 User Process

This section describes the principal data structures of the UP.

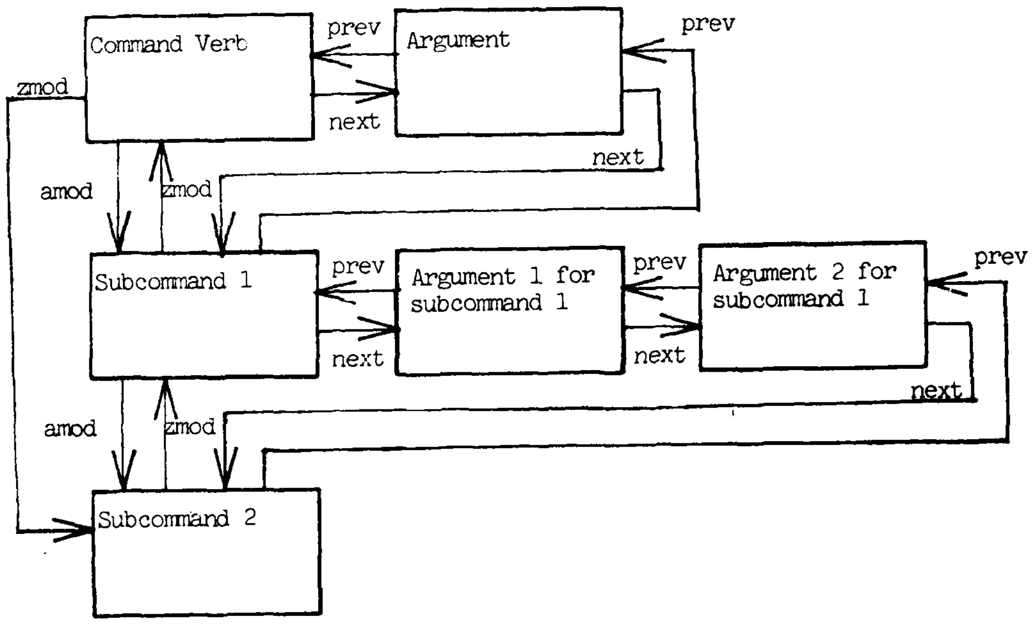
3.2.4.1 PS - Parse Structure Class

Each token partially or completed entered by the user causes a PS entry to be allocated. If a field of the command being entered by the user is removed, the corresponding PS entry is also removed; PS entries are added and removed sequentially, i.e., added and removed at the end of the PS table. In the case of subcommands, a special linkage of PS entries is supported (see below). The structure of each PS entry is:

```
/* PARSE STRUCTURE TABLE */

struct ps
{
   struct ps *ps_pent;  /* ptr to previous entry */
   struct ps *ps_nent;  /* ptr to next entry */
   int ps_type;         /* type of gt node */
   struct gt *ps_gtpt;  /* ptr to corresponding gt node */
   char *ps_btok;       /* ptr to beginning of token */
   char *ps_etok;       /* ptr to end of token */
   int ps_val;          /* "value" of parsed word */
   char *ps_prev;       /* ptr to associated string in gtree */
   int ps_flag;         /* flags (defined below) */
   struct ps *ps_prev;  /* ptr to prev parse node */
   struct ps *ps_next;  /* ptr to next parse node */
   struct ps *ps_amod;  /* ptr to first or next <mc> node */
   struct ps *ps_zmod;  /* ptr to last or prev <mc> node */
};

/* flags used in ps_flag */

#define PS_RECOG 01   /* word has been successfully parsed */
#define PS_COMPL 02   /* word ends a complete command */
#define PS_CC    010  /* have command for <mc>, expecting
                         CR LF */
#define PS_PROM  040  /* prompt has been printed */
#define PS_ABORT 0100 /* user has aborted Help */
```

The first PS entry of each subcommand is linked to the first PS entry of the main command or to the first entry of the previous subcommand. Thus, subcommands "hang" off the command verb. For example, the PS entries for a generalized command with subcommands would be linked as follows:

FIG. 2. PARSE STRUCTURE NODES FOR A GENERALIZED COMMAND



Note: pointers are labeled with appropriate structure member names.

3.2.4.2 US - User Session Table Class

For each active command which has generated remote action, there is a US entry. It contains all information necessary for the Front End to process the command; each US entry has the following structure:

```
/* USER SESSION TABLE */

struct us
{
   struct us *us_pent;  /* ptr to previous entry */
   struct us *us_nent;  /* ptr to next entry */
   int *us_aspt;        /* ptr to AS entry in PP */
   int us_cid;          /* command id */
   int_stid;            /* stream id */
   int us_stat;         /* command status word */
   struct tb *us_tbpt;  /* ptr to first TB entry for this
                           command */
   int us_scen;         /* FE ptcl scenario number */
   int us_pcod;         /* value from grammar tree */
   int us_attp;         /* file ptr for associated tool */
   char *us_cmd;        /* ptr to command string */
   char *us_str;        /* ptr to auxiliary string */
   struct us *us_prev;  /* ptr to prev command; null if
                           no sequencing */
   struct us *us_next;  /* ptr to next command; null if
                           no sequencing */
};


/* flags used in us_stat field */

#define US_ANNOU  01   /* user announcements have been made */
#define US_SCDON  02   /* scenario has completed */
#define US_HLSUP  04   /* Help has been supplied */
#define US_DEFER  010  /* command issued in deferred mode */
#define US_ABORT  020  /* user has aborted the command */
#define US_PRALL  040  /* print notification regardless of
                          terminal context */
#define US_REMOT  0100 /* scenario remotely-initiated */
#define US_NOANN  0200 /* inhibit user announcements */
#define US_TLHLP  0400 /* processing automatic slew from
                          tool mode */


/* flags used in us_umes field */

#define US_OURDY  01   /* non-Help output ready */
#define US_HLRDY  02   /* Help output ready */
#define US_ERRDY  04   /* error output ready; display
                          immediately */
#define US_COMPL  010  /* command has completed */
#define US_TLRDY  020  /* associated tool ready to use */
#define US_TLCOW  040  /* associated tool closed, there is
                          output waiting */
#define US_TLCLS  0100 /* associated tool closed, no output
                          waiting */
#define US_TLNRD  0200 /* associated tool not ready to use */
```

The output buffered for display to the user is accessed via us tbpt.

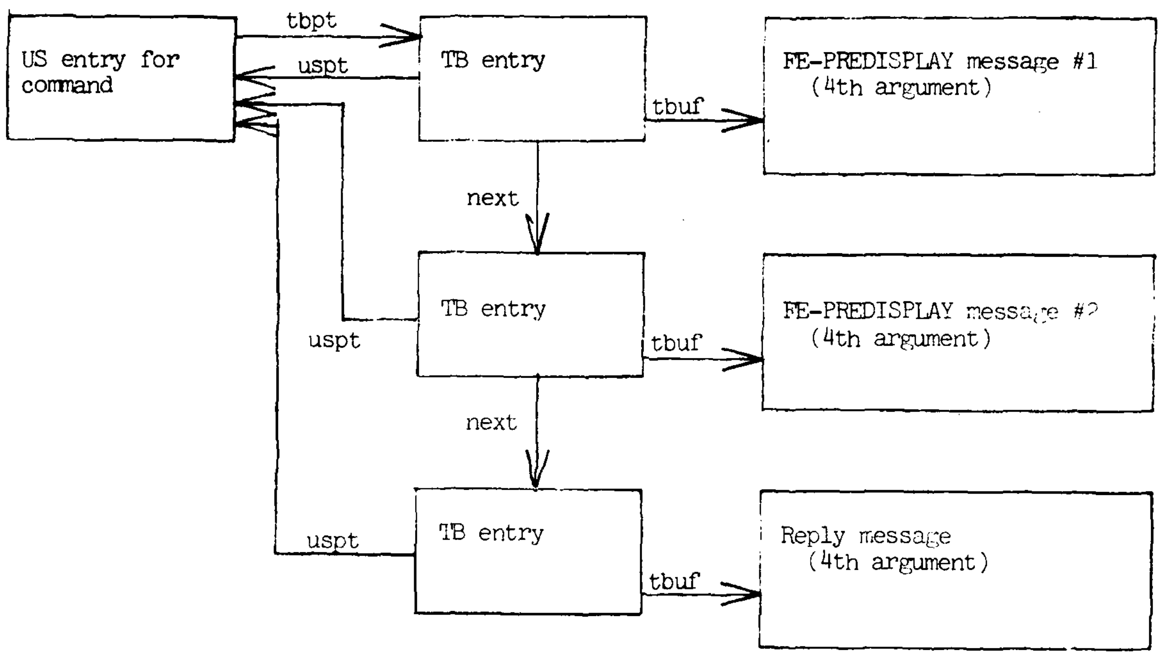
3.2.4.3 TB - Terminal Buffer Header Class

For each active command, one or more buffer of information may be displayed to the user. Each buffer is associated with a TB entry. Each TB entry points to the associated US entry and to the next TB entry. The structure of a TB entry is:

```
/* TERMINAL (Output) BUFFER HEADERS */

struct tb
{
   struct tb *tb_pent;   /* ptr to previous entry */
   struct tb *tb_nent;   /* ptr to next entry */
   struct us *tb_uspt;   /* ptr to corresponding US entry */
   struct tb *tb_next;   /* ptr to next TB entry for same US
                            entry */
   int tb_char;          /* # chars buffered */
   char *tb_tbuf;        /* ptr to buffer */
   int tb_inst;          /* code for kind of info stored
                            -- is ipptr->ip_inst */
   int tb_pcod;          /* value of us_pcod for this buffer */
};
```

It is necessary to store tb pcod in each TB entry because this value may change over the course of executing a command which has more than one protocol scenario associated with it.

Suppose the user issued a command which caused two FE-PREDISPLAY messages plus a Reply to be sent by the WM. The corresponding US and TB entries would be linked as follows:



US entry for command
tbpt
uspt
TB entry
tbuf
FE-PREDISPLAY message #1 (4th argument)
next
TB entry
uspt
tbuf
FE-PREDISPLAY message #2 (4th argument)
next
TB entry
uspt
tbuf
Reply message (4th argument)

Note: pointers are labeled with appropriate structure member names.

FIG. 3. HOW USER OUTPUT IS BUFFERED IN THE USER PROCESS

3.2.4.4 TQ - Tool Process Message Queue Class

Each TQ entry holds a pipe header to be transmitted to be TP when it has been initialized. The structure of each entry is:

```
/* Tool Process Message Queue */

struct tq
{
   struct tq *tq_pent;   /* ptr to previous entry */
   struct tq *tq_nent;   /* ptr to next entry */
   struct ip tq_Ipbk;    /* the IPC header to be sent */
};
```

3.2.4.5 Grammar Tree

The grammar tree is a global data item and is not a data class. Each GT node is linked to following PY or PZ entries. A PY entry is used when the syntactic item can be recognized. A PY entry can consist of one or more entries, each defining a valid string in the current position of the tree. A PZ entry follows the GT node if the syntactic item may not be recognized, i.e., it is a parameter.

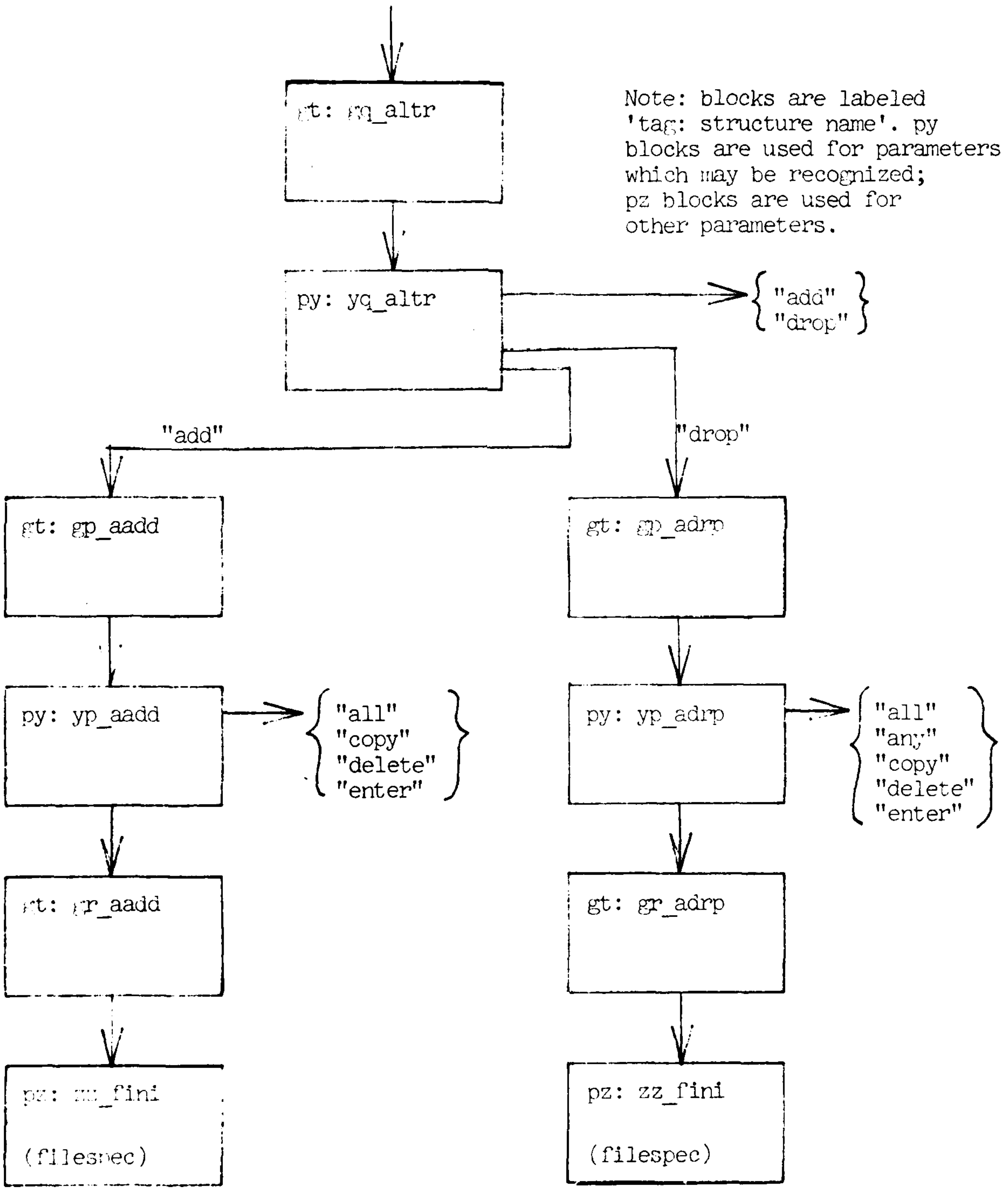
```
/* structure of grammar tree node */

struct gt
{
   int gt_type;     /* flags and type of node */
   char *gt_prom;   /* ptr to prompt string */
   char *gt_help;   /* ptr to help string */
   char *gt_guer;   /* ptr to query string */
   int *gt_flkp;    /* ptr to first PY/PZ entry */
};

/* structure of parse table entry (lookup) */

struct py
{
   char **py_list;        /* ptr to list of syntactic
                             items */
   struct gt *py_nxgt;    /* ptr to next gt node */
   struct gt *py_mcgt;    /* ptr to <mc> node in gt */
   int py_flag;           /* flags, defined below */
   int py_val;            /* value */
};
```

```
/* structure of parse table entry (no lookup) */

struct pz
{
   int pz_val;              /* default value of parm, or
                               ignored */
   struct gt *pz_nxgt;      /* next gt node */
   struct gt *pz_mcgt;      /* ptr to <mc> node in gt */
};


/* flags used in py_flag */

#define PY_UNLI  01   /* command may be used even if
                         user is not logged in */
#define PY_ULIP  02   /* command may be used if login
                         is pending */
#define PY_ULDI  04   /* command may be used if user
                         is logged in */
#define PY_ULOP  010  /* command may be used if logout
                         is pending */
#define PY_USMC  0400 /* command supports modification */
```

py mcgt and pz mcgt are non-NULL only if the command supports command modification. Only the mcgt entry in the command verb PY or PZ entry is used. If a command supports run-time recognition and completion, the C function address for the routine to assemble the list of syntactic items is placed into py val. py val is also used to hold the Front End protocol scenario number of the command being invoked, together with a command code specifying what action the command invokes. The flags in py flag specify so-called "user state" information, i.e., under what conditions a command may or may not be issued.

As an example, the structure of the grammar tree for the ALTER command is given in Figure 4.

FIG. 4. STRUCTURE OF GRAMMAR TREE FOR ALTER COMMAND



3.2.4.6 Help Tree

The Help tree is a global data item and is not a data class. It has the same structure and conventions as the grammar tree (above), but defines legal user input for the Help reply terminal context.

3.2.4.7 UC - User Context Block

The user context block is a global block giving global user context information. In the current version, uc ctx and uc gtrt are not used; their function is performed by global variables context and gtroot, respectively. The structure and associated flags of the block are:

```
/* User Context block */

struct uc
{
   int uc_flag;         /* flags; see below */
   int uc_ctx;          /* user terminal context */
   struct gt *uc_gtrt;  /* starting GT node for parsing */
   char *uc_tbuf;       /* ptr to terminal buf for context */
   char *uc_nxbf;       /* to location next char is to be
                           buffered into */
};

/* flags used in uc_flag */

#define UC_MCOK   04   /* command supports modification */
#define UC_BGMOD  010  /* about to begin command mod */
#define UC_QUOTE  020  /* next char is quoted */
```

3.2.5 Common To All Three Processes

This section describes the principal data structures common to all three Front End processes.

3.2.5.1 TT - Active Tool Table

The active tool table is in effect a piece of shared memory for the three Front End processes. It incorporates all globally-required information about active Conversational Partners. Each Front End process references and modifies TT at points in its execution. The structures of each TT entry is:

```
/* structure of tool table entry */

struct tt
{
   long tt_tlid;               /* tool id (WM-assigned) */
   int tt_flag;                /* flags - defined below */
   int *tt_aspt;               /* ptr to corresponding AS
                                  entry in Protocol Process */
   int *tt_tipt;               /* ptr to corresponding TI entry
                                  in User Process */
   int *tt_uspt;               /* ptr to corresponding US entry
                                  in User Process */
   int tt_cnid;                /* MSG conn id / UNIX shell pid */
   int tt_ctyp;                /* NSW connection type */
   int tt_lskt;                /* local socket number */
   long tt_fnum;               /* foreign socket/host number */
   char tt_gtnm[TNAMSIZ];      /* generic tool name */
   char tt_pnam[MSGPSIZ];      /* MSG process name */
};

/* synonyms for some TT fields */

#define tt_spid  tt_cnid /* UNIX shell pid */
#define tt_fskt  tt_fnum /* foreign socket number */
#define tt_hnum  tt_fnum /* foreign host number */
```

```
/* flags used in active tool table (tt_flag) */

#define TT_NSWT   01     /* tool is standard NSW tool */
#define TT_TELN   02     /* tool is a TELNET connection */
#define TT_LNDSV  04     /* set if LND has been saved */
#define TT_ABORT  010    /* tool is being aborted */
#define TT_TERM   020    /* tool is being terminated */
#define TT_ERROR  040    /* error has occurred */
#define TT_NRDY   0100   /* tool not ready to use */
#define TT_CLOSE  0200   /* connection has been closed */
#define TT_START  0400   /* tool has been started/restarted */
#define TT_FINIS  01000  /* TP has received P_FINIS for tool */
#define TT_HCALL  02000  /* there is an outstanding Help call
                            for this tool */
#define TT_SHELL  04000  /* tool is a UNIX subshell */
#define TT_ALL    -1     /* every possible flag */
```

For NSW tools, the 'generic tool name' is a tool name as found in the user's WM node entry, e.g., 'teco-r2'; the 'tool instance name' consists of a FE-supplied numeric prefix, followed by the generic tool name.

In the current version, TNAMSIZ is 20 bytes, and MSGPSIZ is 32 bytes.

3.2.5.2 PB - Process Control Block

The process control block is a global data item specifying process status. Most of it is initialized at startup and not altered subsequently. The structure of the block is:

```
/* Process Control Block */

struct pb
{
   int pb_spid;      /* pid of self */
   int pb_fdin;      /* fd of user input */
   int pb_fdou;      /* fd of user output */
   char pb_mode;     /* mode of process */
   char pb_proc;     /* identify process to self */
   int pb_stat;      /* state of process */
   int pb_ttfd;      /* fd of tool table */
   int pb_erfd;      /* fd of error log file */
   int pb_lgfd;      /* fd of event log file */
};
```

PRECEDING PAGE BLANK-NOT FILMED

4. PROGRAM COMPILING, LOADING, AND MAINTENANCE PROCEDURES

4.1 Support Software Requirements

The source programs for the Front End are a collection of program modules, written in the programming language C, as implemented by the ncc compiler. This compiler is an enhanced version of the Bell released version 6 C compiler and is less powerful than the pcc [Phototypesetter License] compiler.

To generate a new version of the Front End, the modules must be compiles and loaded with ncc. Shell files are provided for this purpose, and these are listed in Section 4.2.

The modules which comprise the Front End source programs are kept on-line in the UNIX file system. Each module is stored in a separate UNIX file. The constituent modules are:

alloc.h	defines for alloc.c
allocn.h	defines for allocn.c
att.h	structure and flags for TT
fe.h	defines global to FE
gtree.h	defines for gtree.c
htree.h	defines for htree.c
ipc.h	defines for inter-process communication
nsw.h	global FE defines for NSW items
open.h	ARPANET NCP open parameter block
pscentbl.h	defines for pscentbl.c
ptcl.h	defines global to PP
ptclv.h	external declarations global to PP
tool.h	defines global to TP
toolv.h	external declarations global to TP
user.h	defines global to UP
userv.h	external declaration global to UP
adtobf.c	function AdToBuf

adtool.c	function AddTT
alloc.c	manage UP data classes
allocn.c	manage PP and TP data classes
atoint.c	function AToInt
att.c	low-level tool table handling routines
ckatt.c	tool table dump program
cklog.c	event log display program
clrbyt.c	function ClrByte
cnvint.c	function Cnvlntg
convlc.c	function ConvLC
convuc.c	function ConvUC
copyfl.c	function CopyFil
cpbyte.c	function CopyByt
dalloc.c	function DoAlloc
ercond.c	function ErrCond
flipby.c	function FlipByt
flshfd.c	function FlushFD
fntool.c	look up and/or delete specified TT entries
gtree.c	grammar tree
htree.c	Help tree
intrch.c	function InitPB
intrch.c	function IntrpCh
ldvstr.c	function LdVStr
log.c	logging routines
makindx.c	program to generate FE text index file from FE text file
mkerds.c	function MkErDes
mkfmen.c	function MkFMEnd
mkhnam.c	function MkHstNm
mpars.c	parse NSWTP messages
mprt.c	print out NSWTP messages
mutil.c	load NSWTP messages
mwrite.c	write NSWTP messages
nswfe.c	the FE initialization process
ntoa.c	function NToA
parg.c	function ProcArg
pdc_mb.c	manage MB data class
pipe.c	pipe interface
pissue.c	issue MSG primitives
ppipe.c	UP to PP pipe interface
pprim.c	handle MSG primitives and FE-callable procedures
precv.c	handle received NSWTP messages
prtlng.c	function PrtLong
pscen.c	initiate and terminate protocol scenarios
pscentbl.c	Protocol Scenario Definition Table
ptcl.c	high-level PP routines
ptclv.c	global PP variables
putil.c	utility routines for PP
rdbyte.c	function RdBytes
readwd.c	function ReadWrd

skipln.c	function SkipLin
skipwd.c	function SkipWrd
streq.c	function StrEq
telnio.c	handle TELNET Remote Echo option
tool.c	high-level TP routines
toolv.c	global TP variables
tpipe.c	UP to TP pipe interface
ttool.c	open and close Conversational Partners; change process mode
tutil.c	utility routines for TP
uchar.c	handle FE command language special function characters; buffer other characters
ucmd.c	execute certain local user commands
udc_tb.c	manage TB data class
udc_tq.c	manage TQ data class
udc_um.c	manage user message buffer
udc_us.c	manage US data class
uhelp.c	handle Help reply mode
uinit.c	UP initialization routines
ulkp.c	parse lookup routines
umesg.c	high-level NSWTP message handler
umutil.c	message handling utilities
uparam.c	parameter parse routines
upars.c	high-level parse routines
upipe.c	PP to UP, TP to UP pipe interface
uproc.c	status utilities
uputil.c	parse utilities
urmes.c	assemble NSWTP messages
user.c	high-level UP routines
userv.c	global UP variables
usig.c	signal handlers
uterm.c	handle terminal I/O
utool.c	handle Conversational Partners
util.c	utility routines for UP
wrtbyt.c	function WrtByte

4.2 Procedures

Shell files are used to compile and load the UNIX FE. These shell files assume that the C-language source files are in a subdirectory named src, that the object files are to be placed in a subdirectory named obj of the same (immediate) parent directory, and that the executable binaries are to be placed in the user's bin subdirectory. The files are listed below.

4.2.1 Creating the Protocol Process

To create the PP:

```
sh comp ptcl
sh comp ptclv
sh comp pdc_mb
sh comp pissue
sh comp ppipe
sh comp pprim
sh comp precv
sh comp pscen
sh comp pscentbl
sh comp putil
sh comp mpars
sh comp mprt
sh comp mutil
sh comp mwrite
sh comp adtool
sh comp fntool
sh comp att
sh comp allocn
sh comp log
sh comp pipe
cd ../obj
echo "[Loading PP]"
ncc -i  ptcl.o\
       ptclv.o\
      pdc_mb.o\
      pissue.o\
       ppipe.o\
       pprim.o\
       precv.o\
    pscentbl.o\
       putil.o\
       mpars.o\
        mprt.o\
       mutil.o\
      mwrite.o\
      allocn.o\
         log.o\
      adtool.o\
      fntool.o\
         att.o\
        pipe.o\
   /usr/nswmsg/bin/pmsg.o\
       libfe.a\
  /lib/libn.a
```

```
mv ptcl .../bin/ptcl
size .../bin/ptcl
cd ../src
printf "\007"
```

4.2.2 Creating the Tool Process

To create the TP:

```
sh comp tool
sh comp toolv
sh comp telnio
sh comp tpipe
sh comp ttool
sh comp tutil
sh comp att
sh comp log
sh comp allocn
sh comp fntool
sh comp pipe
cd ../obj
echo "[Loading TP]"
ncc - i      tool.o\
            toolv.o\
           telnio.o\
            tpipe.o\
            ttool.o\
            tutil.o\
           allocn.o\
              att.o\
              log.o\
           fntool.o\
             pipe.o\
            libfe.a\
           libtn.a\
           /lib/libn.a
mv tool .../bin/tool
size .../bin/tool
cd ../src
printf "\007"
```

4.2.3 Creating the User Process

To create the UP:

```
sh comp user
sh comp userv
sh comp uchar
sh comp ucmd
sh comp udc_tb
sh comp udc_tq
sh comp udc_um
sh comp udc_us
sh comp uhelp
sh comp uinit
sh comp ulkp
sh comp umesg
sh comp umutil
sh comp uparam
sh comp upars
sh comp upipe
sh comp uproc
sh comp uputil
sh comp urmes
sh comp usig
sh comp uterm
sh comp utool
sh comp uutil
sh comp gtree
sh comp htree
sh comp mpars
sh comp mprt
sh comp mutil
sh comp att
sh comp log
sh comp adtool
sh comp fntool
sh comp alloc
sh comp pipe
cd ../obj
echo "[Loading UP]"
ncc -i     user.o\
          userv.o\
          uchar.o\
           ucmd.o\
         udc_tb.o\
         udc_tq.o\
         udc_um.o\
         udc_us.o\
          uhelp.o\
          uinit.o\
          umesg.o\
         umutil.o\
         uparam.o\
          upars.o\
                upipe.o\
                uproc.o\
               uputil.o\
                urmes.o\
                 usig.o\
                uterm.o\
                utool.o\
                uutil.o\
                gtree.o\
                htree.o\
                mpars.o\
                 mprt.o\
                mutil.o\
                alloc.o\
                  att.o\
               adtool.o\
               fntool.o\
                  log.o\
                 pipe.o\
                libfe.a\
               /lib/libn.a
mv user .../bin/user
size .../bin/user
printf "\007"
```

The shell file comp is listed below:

```
if ! -newer $1.c -than ../obj/$1.o exit
ncc -c -O $1.c
mv $1.o ../obj
echo $1.c
```

4.2.4 Modifying Front End Library

To update the Front End library:

```
cp ../obj/libfe.a libfe.temp
sh updlib ntoa
sh updlib skipln
sh updlib readwd
sh updlib copyfl
sh updlib skipwd
sh updlib ercond
sh updlib convlc
sh updlib convuc
sh updlib parg
sh updlib streq
sh updlib prtlng
sh updlib cnvint
sh updlib intrch
sh updlib atoint
sh updlib mkhnam
sh updlib adtobf
sh updlib flshfd
sh updlib rdbyte
sh updlib mkfmen
sh updlib mkerds
sh updlib cpbyte
sh updlib flipby
sy updlib clrbyt
sh updlib initpb
sh updlib ldvstr
sh updlib dalloc
sh updlib wrtbyt
my libfe.temp ../obj/libfe.a
echo "[libfe updated]"
printf "\007"
```

The shell files updlib, addlib, and rmlib are listed below:

updlib:

```
if ! -newer $1.c -than ../obj/libfe.a exit
ncc -c -O $1.c
ar u libfe.temp $1.o
rm $1.o
echo $1.c
```

addlib:

```
ncc -c -O $1.c
ar u ../obj/libfe.a $1.o
ar t ../obj/libfe.a
rm $1.o
echo ""
echo $1.c
printf "\007"
```

rmlib:

```
cd ../obj
ar d libfe.a $1.o
ar t libfe.a
cd ../src
```

4.2.5 Creating the Front End Initialization Process

To create the initialization process (ncc version)

```
sh comp nswfe
cd ../obj
echo "[Loading nswfe]"
ncc nswfe.o libfe.a
mv nswfe .../bin/nswfe
cd ../src
```

4.3 Debugging Facilities

This section describes debugging facilities available for debugging the Front End.

4.3.1 Debugging Facilities Available from UNIX

UNIX provides an interactive debugger called adb which allows interactive debugging of top-level processes and of core dumps of extinct processes. It operation is described in [6] under ´adb(I)´. Adb, however, cannot be used to debug interactively inferior processes in a process group, nor can it be used to debug any UNIX processes executing with separate Instruction and Data space. Since all Front End processes run in separate Instruction and Data space, abd cannot be used interactively to debug them.

Under certain conditions, a process may terminate abnormally and leave a core dump of itself. Core dumps are taken when certain UNIX signals are not caught or when the process executes an abort system call. In all cases, the core dump is placed in a file called core, and if more than one process so terminates at the same time, only one core dump remains. This places a constraint on the use of core dumps to obtain debugging information. It is partly in view of these constraints that FE-specific debugging facilities have been implemented.

4.3.2 Debugging Facilities Specific to the Front End

Debugging facilities specific to the Front End are (1) a facility to obtain diagnostic typeouts, and (2) error logging.

4.3.2.1 Diagnostic Typeouts

In the Front End initialization file, the bit string governing diagnostic typeouts may be used to cause several classes of typeouts to appear on the user's terminal. These classes and other details concerning setting up the initialization file are described in [3].

4.3.2.2 Error Logging

It is possible to force all such error entries to appear on the user's terminal on the user's terminal as well as being written to the error log file by setting LOG_TTY (bit 15, numbered from right to left) in the bit string governing event-logging in the Front End initialization file. For details on setting this bit string in the initialization file, refer to [3].

The Front End broadly distinguishes two kinds of errors: (1) those generated in the UNIX operating system (e.g., I/O errors, file reference errors, etc.), and (2) those generated in the Front End (e.g., running out of a resource, or an internal inconsistency). Both classes of errors are always logged in the error log file. However, generally, if an error is not fatal, it will not be indicated on the user's terminal, and there may or may not be a users message. If an error is fatal, the error log entry will also appear on the user's terminal, and the Front End will halt.

Error log file names begin with 'PP', 'TP', or 'UP' and end with 'err+.' If an error log file to be written into already exists, the Front End appends new entries to it; otherwise, the file is first created and then written into.

4.4 Logging Facilities

This section describes the logging facilities available for the Front End. Conceptually, three kinds of logging may be distinguished:

- Error Logging: errors detected in execution are logged; see Section 4.3.2.2 above.
- Event Logging: certain events in the execution of the front End are logged.

o Performance Logging: information relating to the performance of the Front End is logged.

4.4.1 Logging Facilities Available from UNIX

C programs may be compiled with the -p option, which causes special profiling code to be included in the object output. Using the monitor subroutine (see ´monitor (II)´ in [6]), the running process will produce a profile output file. This file is read by the program prof (see ´prof(I)´ in [6]) to produce output giving the number of times each function is called and the percentage of time spent in each function; this information is based upon a sample of the program counter each 1/60-th second.

No Front End process is normally compiled with UNIX profiling enabled.

4.4.2 Logging Facilities Specific to the Front End

In the Front End initializtion file, the bit string governing event logging may be used to cause several classes of events to be logged in an events to be logged in an event log file. These classes and other details concerning setting up the initialization file are described in [3].

Event log file names begin with process id, followed by ´PP´, "TP, or ´UP´, then followed by ´log+´ (for example, ´35PPlog+´). Event log files are non-Ascii and are read by program cklog (see Section 4.6.3 below) to write an Ascii interpretation to the standard output.

For event logged, there are three pieces of timing information which may be used for Front End performance analyses: (1) the current time, as returned by the time system call, (2) the (cumulative) number of clock ticks the process has spent in UNIX system code, and (3) the (cumulative) number of clock ticks the process has spent in the user ciode. The latter two values are obtained from the times system call.

These three timing parameters, in the order given, are the first three parameters in each event log record; refer also to Section 2.2.9.1 concerning module log.h.

4.5 Verification

At present there is no formal verification procedure for the UNIX Front End.

4.6 Special Maintenance Programs

This section describes special maintenance programs used to maintain the Front End.

4.6.1 MAKF

The function headers described in Section 1.5.3 above are generated by using the shell file makf; makf uses the UNIX M6 macro processor and the file func.m6 to create a template for each function header. makf is listed below:

```
echo @func,$1"<$3>":lm6 func.m6>y
cat $2 y>z
mv z $2
rm y
```

A sample invocation of makf would be:

```
sh makf ChkVal file.c "argl,arg2"
```

where ChkVal is the name of the function to be created; file.c is the name of the file to which to append the new function; and argl and arg2 are the formal parameters of the function.

4.6.2 MAKINDX

Program makindx accepts as input the Front End text file and produces as output the Front End text index file. The name of the input file must be ´fe-text´, and the name of the output file will always be ´fe-txtindex´. The program takes no argument. Makindx must be executed on every occasion that the Front End text file is altered.

4.6.3 CKLOG

Program cklog reads an event log file and writes an Ascii interpretation of it to the standard output. It requires an event log file name as an argument. It uses the standard I/O library <u>stdio</u> and is compiled with <u>pcc</u>.

4.6.4 CKATT

Program ckatt reads an active tool table and writes an Ascii dump of it to the standard output. It requires a tool table file name as its argument. It uses the standard I/O library stdio and is compiled with pcc.

4.7 Other Special Maintenance Procedures

None.

4.8 Error Conditions

Refer to Section 4.3.2.2 above.

4.9 Listings

Listings for the Front End source programs are supplied as an appendix to this document under separate cover.

PRECEDING PAGE BLANK-NOT FILMED

APPENDIX AA
HOW TO ADD A NEW COMMAND TO THE FRONT END

A.1 Adding a Local Command

The following is a list of steps that must be done in order to add a local command to the Front End.

- o add the command code to gtree.h; LOC_CMD must be on
- o add the command syntax to gtree.c
- o add a ´case´ to LoclCmd in umesg.c of the form:

```
case <command-code>:
    { the code to execute the command }
    break;
```

It may be that if other modifications, e.g., pipe message handling, are part of the new command, new pipe instructions codes and handling routines must also be added. The preceding list is minimal.

A.2 Adding a Remote Command

To add an Front End command that generates remote activity, the preceding minimal steps for adding a local command must be followed. In addition, the following minimal steps must be done:

- o If a new protocol scenario is being defined, define a name for the new scenario in fe.h and add it to the command code word of the command in gtree.c, and add a new protocol scenario (including entries to <u>pstmap</u> and <u>prname</u>) to pscentbl.c.

- o In statements of the form: ´switch (asptr->as_scen)´ or ´switch (ipptr->ip_scen)´ see if additional cases must be added.

- o Write C functions for the PP to process all messages received as part of the new protocol scenario (if one is defined), these functions probably called from RcvSpDn.

The preceding steps are minimal, but may, in some cases, be sufficient.

References:

[1] Ritchie, D. M., Johnson, S. C., Lesk, M. E., and Kernighan, B. W.
The C Programming Language.
The Bell System Technical Journal 57(6, part 2):1991-2020, July-August, 1978.

[2] Thomas, Robert H.
UNIX NSW Front End Final Report.
Technical Report 4242, BBN, November, 1979.

[3] Lind, Henrik O.
NSW UNIX Front End User's Manual.
Technical Report 4476, BBN, November, 1980.

[4] Toner, Stephen G.
MSG: The Interprocess Communication Facility For The National Software Works, Program Maintenance Manual: UNIX Implementation.
Technical Report 4579, BBN, January, 1981.

[5] Johnson, Paul R. and Toner, Stephen G.
MSG: The Interprocess Communication Facility For The National Software Works, User Manual: UNIX Implementation.
Technical Report 4580, BBN, January, 1981.

[6] Bolt Beranek and Newman, Inc.
Programmers Manual for the UNIX Operating System, Sixth Edition
1979.
Revised by Bolt Beranek and Newman, Inc.; based on the work of K. Thompson and D. M. Ritchie.

Appendix B

UNIX MSG Maintenance Manual

MSG: THE INTERPROCESS COMMUNICATION FACILITY FOR THE NATIONAL SOFTWARE WORKS

PROGRAM MAINTENANCE MANUAL:
UNIX IMPLEMENTATION

Stephen G. Toner

TABLE OF CONTENTS

LIST OF FIGURES

1. GENERAL DESCRIPTION

1.1 Purpose of MSG Program Maintenance Manual

The purpose of this Program Maintenance Manual (PMM) for the UNIX[1] implementation of MSG is to provide maintenance programmer personnel with sufficient information to maintain that implementation. The reader is assumed to be familiar with the MSG System/Subsystem Specification and the UNIX MSG User Manual.

1.2 MSG Application

MSG was designed to be the interprocess communication facility for the National Software Works (NSW) system. The NSW system is an operating system for a collection of heterogeneous computers (called hosts) connected to a computer network. NSW itself is implemented by a collection of modules which execute as processes on the various host computers. The ARPA computer network (ARPANET) supports inter-host communication for the current NSW implementation.

MSG supports the inter-host and intra-host communication requirements of the various modules which implement the NSW system. Because the interprocess communication requirements of NSW are fairly general, MSG is a generally useful inter-host interprocess communication facility which is applicable outside of the NSW system. The remainder of this document, however, focuses on MSG maintenance as it relates to the NSW system.

MSG supports NSW patterns of communication by providing two different modes of process addressing:

- o generic addressing;
- o specific addressing;

[1] UNIX is a trademark of Bell Laboratories

and three different modes of communication:

- o messages;
- o direct communication paths (connections);
- o alarms.

Each mode of process addressing and communication is intended to satisfy certain NSW requirements and to be used in certain kinds of situations. However, MSG itself does not impose any limitations on how processes use the various communication modes. MSG does not interpret messages or alarms, nor does it intervene in communication on direct connections. The interpretation of messages, alarms, or direct connections is entirely a matter for the processes using MSG to communicate.

Message exchange is provided by MSG to support the requirements of NSW transaction protocols. It is expected to be the most common mode of communication among NSW processes. To send a message, a process addresses it by specifying the address of the process to receive the message and then executes an MSG "send" primitive which requests MSG to deliver the message. When MSG delivers a message to a process it also delivers the name (i.e. specific address) of the process that sent the message.

Generic addressing is used by processes which either have not communicated before or for which the details of any past communication is irrelevant. It is restricted to the message mode of communication. A valid generic address specifies a functional process class. When MSG accepts a generically addressed message it selects as destination some process which is not only in the generic class addressed but has also declared its willingness to receive a generically addressed message. If there is no such process, MSG may create one. Transactions between previously unrelated processes are always initiated by the transmission of a generically addressed message between some pair of processes. A valid specific address refers to exactly one process and this address remains valid for the life of that process. Specific addressing may be used with all three communication modes. Specific addressing is used between processes which are familiar with each other. The familiarity is generally because the processes have communicated with each other before, either directly or through intermediary processes.

The second mode of MSG communication is direct access communication. A pair of processes can request that MSG establish a direct communication path between them. Direct

Communication paths are provided to support the requirements of NSW transaction that are very long (in terms of the amount of data exchanged and possibly the duration), such as terminal-like communication between a Front End and tool/Foreman. (The ARPANET realization for a direct communication path is a host/host connection or connection pair.)

The alarm mode of communication is supported by MSG to satisfy a communication requirement typically satisfied by interrupts in other interprocess communication systems. Alarms provide a means for one process to alert another process to the occurrence of an exceptional or unusual event. Processes may send and receive alarms much as they send and receive messages. However, there are significant differences between alarms and messages. The rules that govern the flow and delivery of alarms are different from those that govern the flow and delivery of messages. In particular, the delivery of an alarm to a process is independent of any message flow to the process. That is, the delivery of an alarm to a process cannot be blocked by any messages queued for delivery to the process. Unlike a message which can carry a substantial amount of information, the information conveyed by an alarm is limited to a very short alarm code. This limitation implies that the delivery of alarms can be accomplished in a way that requires little in the way of communication or storage resources. This makes it possible for MSG to insure certain "priority" treatment for alarms which makes them suitable for alerting processes to exceptional events. While similar to traditional interrupts, alarms are different in one important respect: the delivery of an alarm does not necessarily imply that the process is subjected to a forced transfer of control by MSG. For this reason, we have chosen to use the term alarm rather than interrupt.

1.3 Equipment Environment for MSG

The UNIX implementation of MSG requires a hardware base capable of supporting the BBN-UNIX operating system. This hardware is a DEC PDP-11 model 45 or higher processor with at least 128 KBytes of main memory and at least 20 MBytes secondary disk storage. In addition, a host interface to an ARPANET IMP (Interface Message Processor) is required.

1.4 Program Environment for MSG

The UNIX implementation of MSG runs under the BBN-UNIX operating system. This version of the Bell Labs UNIX system contains the following enhancements:

- o ARPANET NCP and network-related system calls;
- o Ports;
- o the awtenb, awtdis, await and capac system calls;
- o the itime system call.

1.5 Conventions

This section documents the principal programming conventions used in the UNIX MSG implementation.

1.5.1 C Language Implementation

UNIX MSG is implemented in the programming language C (D.M. Ritchie, S.C. Johnson, M.E. Lesk, and B.W. Kernighan, "The C Programming Language." Bell Systems Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978, pp 1991-2019.), and MSG primitive calls are implemented as C language function calls.

1.5.2 Machine-Dependent Code

When it is necessary to perform an operation (e.g. swap bytes of a word) that might not be necessary if MSG were to run on a (16-bit) machine other than the PDP-11, the operation will be surrounded by the compiler control lines:

```
#ifdef pdp11
```

and

```
#endif
```

To enable this code, the identifier pdp11 will be defined (with a #define compiler-control line), and to disable it, pdp11 will be left undefined.

1.5.3 Component-Dependent Code

Certain modules contain code which is used by more than one of the Central MSG, Local MSG and MSG Primitive routines. Often this code is similar but not identical. When a case arises where the code must differ between different components, the code will be surrounded by compiler-control lines of the form

```
    #ifdef cmsg (or lmsg or pmsg)
```

and

```
    #endif
```

One of the identifiers cmsg, lmsg or pmsg will be defined at compile time, using the -D flag of the C compiler (see Section CC(I) of the UNIX Programmers Manual).

1.5.4 Subroutine Commenting Conventions

The top of each page on which a procedure is defined contains a comment which lists the names and arguments of all procedures defined on that page. Immediately before each procedure declaration, there is a brief comment describing that procedure, and the body of the procedure contains "sufficient" comments to describe the action of the procedure.

1.5.5 Global Data Declarations

Global data for a module is defined on the second page of the module (following the #include lines - see below). External symbol declarations give the name of the module in which the symbol is defined.

1.5.6 Declaration Files

MSG uses two types of declaration files: those with extension .names and those with extension .h. The .names files are used to define the long identifier names used in MSG down to 7 or 8 character unique names which are acceptable to the C compiler and the loader (see Section 4.1). The .h files are used to define structures and compile-time constants. All program modules include those files necessary for proper compilation, using the C compiler-control line

```
#include "filename"
```

The first page of each program module contains all the #include lines for that module.

1.5.7 Internal Consistency Checks

MSG contains a number of internal consistency checks. One of the subroutines ErrorHalt or ErrorCheck is used to report the failure of a consistency check. The ErrorCheck subroutine is for non-fatal inconsistencies; it merely logs the consistency check failure and allows execution to continue, presumably first to correct the inconsistency and then to resume normal execution. The ErrorHalt subroutine is for fatal inconsistencies; it logs the inconsistency and terminates MSG, producing a core dump of the MSG component which ErrorHalted in the file core.

1.6 Status of Implementation

The current version of UNIX MSG provides the functions specified in the MSG System/Subsystem Specification with the following exceptions:

- o Sequenced and stream-marked messages are not implemented (see Sections 2.4.1 and 4.2.1.9 of the MSG System/Subsystem Specification).
- o Inter-MSG authentication is not implemented (see Section

4.2.3.4 of the MSG System/Subsystem Specification).

- o The MESS-HOLD protocol is not supported (see Section 4.2.2.3.1 of the MSG System/Subsystem Specification).
- o The SEND-STATUS protocol is not supported (see Section 4.2.2.3.4 of the MSG System/Subsystem Specification).
- o MSG does not guarantee that one each of the primitives listed in Section 4.2.1.8 of the System/Subsystem Specification (under "Access to communication") may be in each process' Pending Event set.

2. SYSTEM DESCRIPTION

This section documents the principle program modules of the UNIX MSG implementation. It describes the structure, operation and composition of the MSG implementation.

2.1 General Description

2.1.1 MSG Processes

An MSG configuration on UNIX is implemented as a collection of UNIX processes. There are two different types of processes used in a configuration: Central MSG processes and Local MSG processes. In addition, there are MSG primitive routines which are loaded with user processes and handle communication between the user processes and the Local MSGs.

The Central MSG is responsible for system initialization and communication with MSG implementations on other hosts. In addition, it handles the creation of processes to handle generic messages for which there is no outstanding ReceiveGenericMessage operation.

Local MSGs buffer messages being sent to and from user processes, and communicate with other Local MSGs and the Central MSG as necessary to support inter-process communication.

An MSG configuration can be started in a variety of ways depending on the configuration operator's objectives (See UNIX MSG User Manual for details). The Central MSG may be started either manually using standard UNIX shell commands, automatically when the first user process which uses MSG is started, or automatically at system startup. User processes can be started automatically by the Central MSG as part of the standard MSG configuration initialization, or they can be started manually from the shell.

2.1.2 Process Structure of the MSG Configuration

The Central MSG consists of several processes (See Figure 1). These processes are described below:

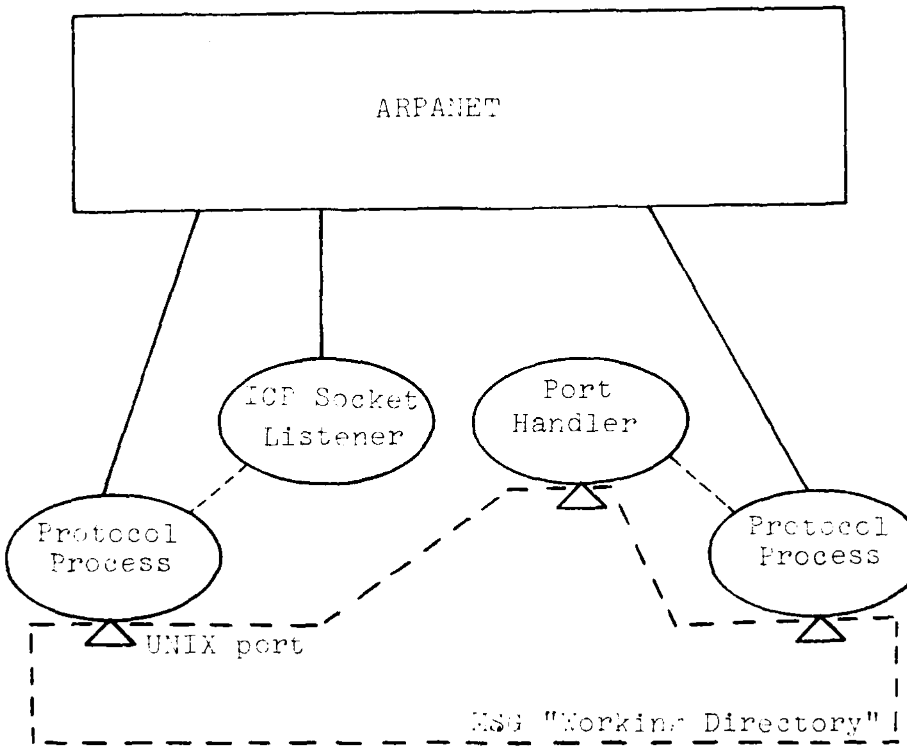


FIG. 1. CENTRAL MSG PROCESS STRUCTURE

- The ICP socket listener

  This process maintains the MSG contact socket. When an MSG on a remote host contacts this socket the UNIX NCP engages in an Initial Connection Protocol (ICP) exchange with it to establish an MSG-to-MSG connection between the remote MSG and the ICP listener process. After the connection has been established, the ICP listener forks to create a Protocol process and goes back to listening on the contact socket.

- The Port handler

  This process maintains the Central MSG port. When a Local MSG desires to send a message to a remote host, and there is no Protocol process for that host, the Local MSG sends the message to the Port handler, which forks to create a Protocol process and hands the message to it. All generically addressed messages destined for the local host also pass through the Port handler, which delivers them to a process which has issued a matching ReceiveGenericMessage primitive, queues them, or creates new processes to handle them, as appropriate.

- Protocol Processes

  There may be zero, one, or more Protocol processes. The principal responsibility of these processes is to implement the MSG-to-MSG protocol with MSGs on remote hosts. There is one protocol process for each MSG-to-MSG connection which is created when the connection is established (see above discussion of ICP contact socket listener and port listener processes). Each Protocol process (except for mutants - see below) maintains a port with a well-known name (actually the host name of the remote host to which it has a connection). The basic action of a Protocol process is to wait for a message on the port or its MSG-to-MSG connection and when it receives one, to translate the message to the proper format and pass it on to the remote MSG or the appropriate Local MSG. Since there may be several MSG-to-MSG connections to a single remote MSG, and there can only be one port with a given name, there may be Protocol processes with no port, which handle incoming traffic only. These are called "mutant" Protocol processes.

Programs which communicate via MSG may consist of any number of processes, but only one of these processes may call MSG primitives. Each process which may issue MSG primitive calls has a child Local MSG process with which it communicates over a pair of pipes (See Figure 2.) At any given time, zero, one or more User Program/Local MSGs may exist on the system.

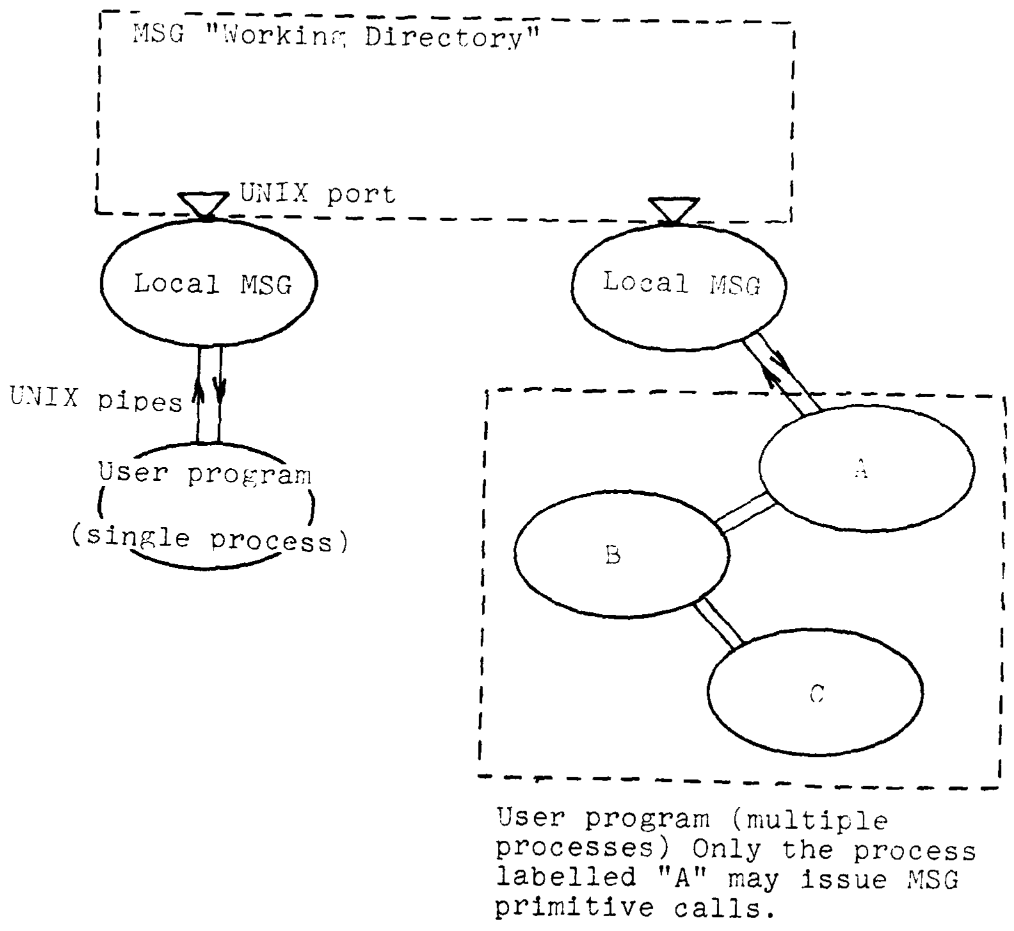

MSG "Working Directory"
UNIX port
Local MSG
Local MSG
UNIX pipes
User program (single process)
A
B
C
User program (multiple processes) Only the process labelled "A" may issue MSG primitive calls.

FIG. 2. USER PROGRAM/LOCAL MSG PROCESS STRUCTURE

2.1.3 Communication among MSG modules

The processes which implement an MSG configuration must communicate with each other in order to perform their functions. This inter-process communication is achieved through the use of UNIX pipes and ports.

Pipes are used between the user process and its associated Local MSG process.

Ports are used for communication between Local MSGs and between Local MSGs and the Central MSG. The Local MSG port names are formed from the generic class name and instance number of the process and are found in the MSG "working directory" as defined in the MSG Configuration File (see Section 3.1.1). Protocol process ports are also found in the MSG working directory, and are formed from the name of the host with which the Protocol process is communicating. The pathname of the Central MSG port is also specified in the Configuration File.

Communication with MSG implementations on remote hosts is achieved by using the standard ARPANET communication functions provided by the UNIX Network Control Program (NCP). The MSG-to-MSG protocol is implemented by the Protocol processes of the Central MSG, as mentioned in Section 2.1.2

2.1.4 Invocation of MSG operations

Executing processes make requests for MSG communication services by invoking MSG operations. These operations are invoked by making C-language function calls (see the UNIX MSG User Manual).

When a process calls an MSG primitive function, the arguments of the call are checked for validity. If they are valid, a Pending Event is created (for those operations that create pending events), and the relevant parameters are then passed on to the Local MSG process. After the MSG primitive routines have completed the requested operation (e.g. for a message sending primitive, they create a pending event and send the message text to the Local MSG, which will attempt to send it on to the destination process), they will return control to the user process (unless the Unblock signal was specified, in which case control is not returned until the operation actually completes) along with an event handle which may be used (as an argument to the ReqSig primitive) to determine when the operation has completed. There is no forced transfer of control when an MSG operation completes.

2.1.5 General MSG Flow of Control

The basic flow of control in terms of MSG modules for MSG primitives which cause messages to be sent to another process is (refer to Figure 3):

- Outgoing messages:

  The user process calls an MSG primitive routine in pmusr.c. pmusr calls pmlmint.c routines to send a message over the pipe to the Local MSG. lmsgo.c reads the message from the pipe and sends it to the Central MSG or the destination Local MSG port. cmsgo.c reads the port and sends the message to the remote host over the network.

- Incoming messages:

  cmsgi.c reads the message from the network connection and writes it to the destination Local MSG´s port. lmsgi.c reads the message from the port and sends it over the pipe to the user process, where it is read by the routines in pmlmint.c and delivered to the user.

The following paragraphs describe in general terms how MSG supports message, alarm, and direct connection communication among processes.

2.1.5.1 Message Communication

When a process executes a receive operation, the MSG Primitive Routines create a pending event (PE) for the receive and pass the receive on to the process´s Local MSG. The Local MSG also creates a PE for the receive, then checks to see if it has received any message that matches the PE. Received messages are kept on separate queues depending on addressing mode (there is a Received Specific Messages queue and a Received Generic Message queue). If a matching message is found, the Local MSG sends the message to the user process, where it is read and delivered by the Local MSG interface routines. The Local MSG interface routines then send an acknowledgement back to the Local MSG, which allows it to release the PE for the receive operation, and the Message Control Block (MCB) which held the message. The Local MSG interface routines may also send back a negative

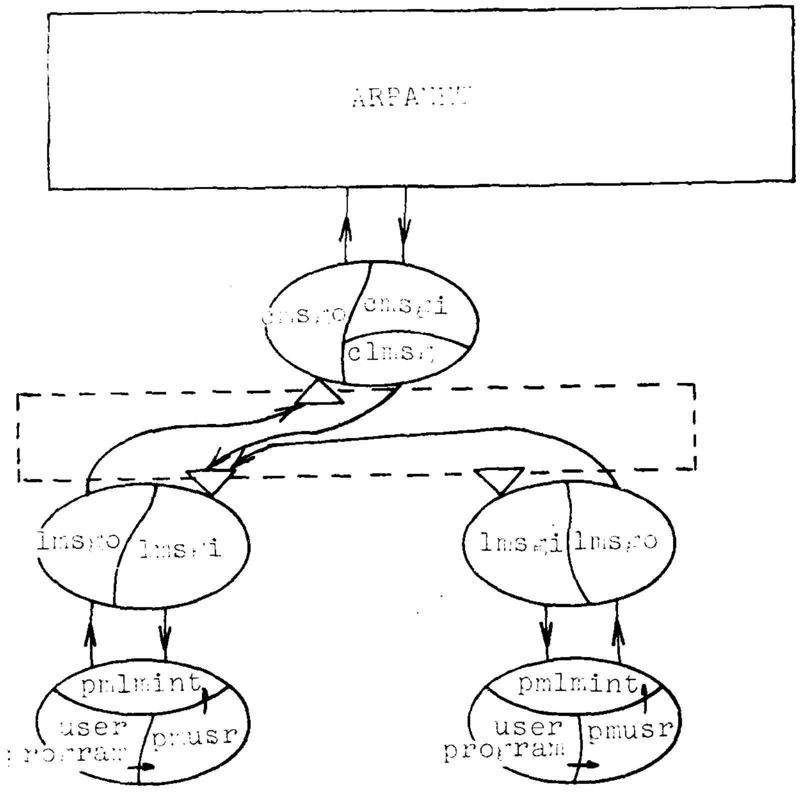


FIG. 3. MSG FLOW OF CONTROL

acknowledgement, in which case the received message is not released, but put back on the queue. After the Local MSG interface routines have acknowledged receipt of the message, they "complete" the PE and deliver the disposition value. If the Request signal was specified, the PE is put on a completed Pending Event queue, where it will be found by the ReqSig primitive. If no matching received message is found by the Local MSG, it will queue the receive Pending Event on a ReceiveSpecificMessage or ReceiveGenericMessage queue. All PEs and MCBs are kept in core.

When a process executes a send operation, the MSG primitive routines create a send PE. They then send the message text on to the Local MSG, which creates a Message Control Block (MCB) for the message and a send PE for the event.

If the destination process is on the local host, the Local MSG will open the destination process's receive port and write the message into it. If the destination process is the same as the source process, the Local MSG will not attempt to write the message into its own port, but will handle it as if it had just been read from the port.

When a Local MSG has received a message from another LMSG (or itself), it sends back an acknowledgement and then checks to see if there is a matching Receive PE. If so, the Local MSG sends the message to the user process, which will cause the receive to complete and the message and disposition to be delivered. When a Local MSG receives a message acknowledgement from another LMSG, it completes the send operation, releasing the send PE and MCB.

If the destination process is on a remote host, the Local MSG will write the message to the Central MSG receive port. The Central MSG will read the message, create an MCB and a PE for it, convert it to the proper format, and queue it to be sent to the remote host. When the Central MSG receives an acknowledgement for the message, it releases the PE and MCB for the send, converts the acknowledgement to internal format and send the acknowledgement on to the source process Local MSG.

The above discussion assumes specific addressing. Generic addressing is rather different, in that all generic messages destined for the local host must pass through the Central MSG port handler process.

When a process executes a receive generic message operation, the MSG primitive routines create a pending event and pass the receive on to the Local MSG. The Local MSG creates a pending event for the receive and passes it on to the Central MSG Port handler process. The Port handler takes the message it receives and attempts to match it to a received generic message for the specified class. If a matching received message is found, it is delivered to the process that executed the receive generic. If there is no outstanding received message, it queues the receive message on a queue of receive generic primitives for the specified class.

When a process executes a send generic operation to a process on the local host, the process's Local MSG sends the message to the Central MSG Port handler process, rather than the destination process's receive port. The Port handler attempts to match the received message with a previously queued receive generic. If it cannot, it may create a process to handle the message, queue the message, or reject it, whichever is appropriate. If a matching receive generic is found, the received message is delivered to the process which issued the receive generic.

When a Protocol process receives a generically addressed message, it also sends the message to the Port handler, which handles it in the same way as a locally generated generic message.

2.1.5.2 Alarm Communication

When a process executes the AcceptAlarm operation, the MSG primitive routines set a flag indicating willingness of the process to accept alarms.

When a process executes an EnableAlarm operation, the MSG primitive routines create an alarm receive PE. They then check to see whether an alarm has already been received by the process. If so, they complete the alarm receive PE by delivering the alarm code, signalling the process and returning control to it. If no alarm is queued, the primitive routines remember the alarm receive PE and return control to the process.

When a process executes a SendAlarm operation the MSg primitive routines create a send alarm pending event and send the alarm code on to the Local MSG, which will send it on to the destination process's Local MSG, or the Central MSG if the destination process is on a remote host. When a Local MSG receives an alarm from another Local MSG (or the Central MSG), it creates a PE and passes the alarm code to the user process. The

Local MSG interface routines then check to see if the process is accepting alarms. If it is not, the alarm is rejected. If it is, the Local MSG interface routines check to see if there is a matching alarm receive PE outstanding. If so, the alarm is accepted and delivered and the alarm receive PE is completed. If there is no matching alarm receive PE, the alarm is accepted and the alarm code remembered.

2.1.5.3 Direct Connection Communication

When a process executes an OpenConn operation, the MSG Primitive routines check to see if a Connection Control Block (CCB) with the same destination process name exists. If so, an error is returned, since the MSG primitive routines keep only active CCBs. If no matching CCB is found, the OpenConn is passed to the Local MSG, which creates a PE and then checks to see if a matching CCB has been queued. If the remote target process has already initiated the connection and the remote MSG has started the protocol exchange necessary to open the connection, a CCB will be found. In this case, the Local MSG "matches" the CCB with the open PE. Next it queues a matching CONNECTION_OPEN message to be sent to the remote process, and returns the remote socket number to the MSG primitive routines.

If there is no matching CCB for the connection, the Local MSG creates one and links it to the open PE. Then it sends a CONNECTION_OPEN message to the remote process.

When a CONNECTION_OPEN message is received by a Local MSG, it looks for a matching CCB. If it finds one, it completes the open PE by sending the remote socket number to the MSG primitive routines. If it does not find a matching CCB, it creates a new CCB which will be matched when the user process issues a proper OpenConn call.

2.2 Detailed Description

This section describes in detail the principal program modules and routines of the UNIX MSG implementation.

2.2.1 Central MSG

The Central MSG consists of the following modules: cmsg.c, cmsgi.c, cmsgo.c, clmsg.c, cmsgnt.c, hosts.c, the utility routines util.c, lmutil.c, generic.c, incnum.c, getconfig.c and the data abstractions llist.c, queue.c, streamio.c and string.c. A module dependency diagram for the Central MSG is shown in Figure 4. The principal modules of the Central MSG are described below.

2.2.1.1 CMSG

The module cmsg.c contains the initialization and main loop procedures for the three different kinds of Central MSG processes.

main is run at program startup. It does all necessary initialization of dynamic data structures, then forks. The parent process then becomes the ICP contact socket listener by calling the procedure HandleContact, while the child process becomes the Port handler, calling the procedure HandlePort.

HandleContact attempts to open a duplex, general, icp server connection (see Section NCP(IV) of the UNIX Programmer's Manual), and if successful, forks to create a protocol process to handle traffic to the remote host which initiated the ICP. After creating this protocol process, the ICP listener process goes back to listening on the contact socket. The newly created protocol process attempts to open its receive port (which may fail), then calls CentralMSG.

CentralMSG is the main loop of the protocol process. It listens for messages coming in over the network, or messages written to its port by Local MSGs (if it is not a mutant process), and passes them on to the appropriate Local MSG or the remote MSG, respectively, after converting them to the proper format. Protocol processes keep an idle time clock, and if a protocol process has not received a message from either the remote MSG or a Local MSG in a certain amount of time (specified in the configuration file - see UNIX MSG User Manual), it will initiate the MSG-to-MSG CLOSE protocol with the remote MSG, and terminate by calling the UNIX system call exit.

HandlePort is the main loop of the MSG port handler. This process may receive two kinds of messages from Local MSGs or other Central MSG processes:

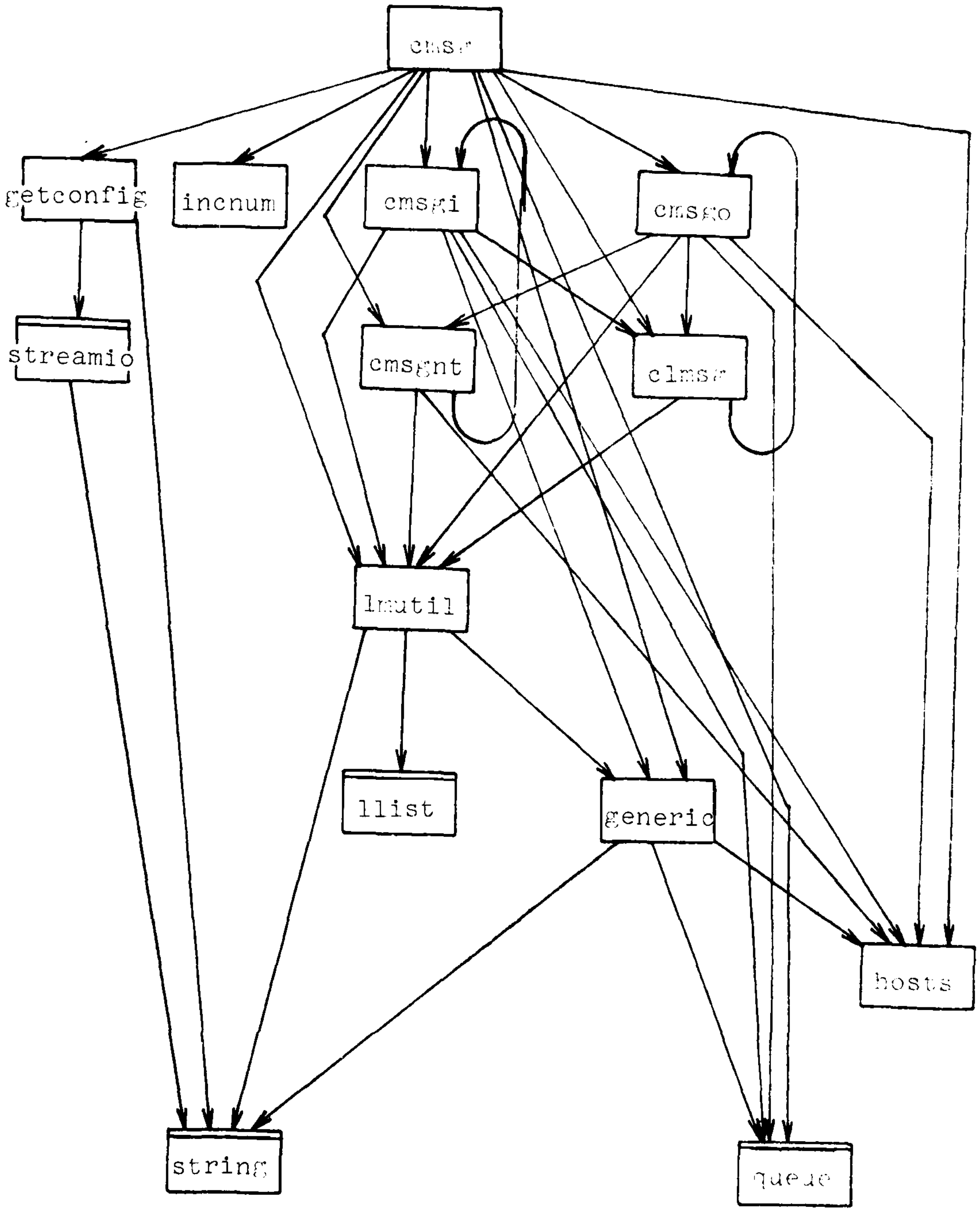


FIG. 4. MODULE DEPENDENCY DIAGRAM FOR CENTRAL MSG

1. Messages destined for a remote host for which there is currently no protocol process. In this case, the port handler will fork to create a protocol process to handle communication with the remote host. The protocol process opens its port and attempts to establish contact with the remote MSG, then calls CentralMSG.

2. Generic messages destined for the local host/ ReceiveGeneric primitives issued by processes on the local host. In this case, the port handler attempts to match the generic message with an outstanding ReceiveGeneric on one of the queues RcvGenrs[i], or to match the ReceiveGeneric with a previously received generic message which was queued on one of the RcvdGenrs[i] queues. If a generic message is received for which there is no outstanding ReceiveGeneric, the port handler may (depending on the CreateSpec for the destination process class) fork to create a process to handle the message. The new process will attempt to execute (using the UNIX <u>execl</u> call) the program specified in the generic_names file for the destination process class.

2.2.1.2 CMSGI

The module cmsgi.c contains routines that handle incoming MSG-to-MSG traffic from the network. Routines in this module are only called by Protocol processes.

The main routine in cmsgi is HandleAnyNetMSGMessage. It reads the length and op code fields of an MSG-to-MSG message and then dispatches on the op code to one of the routines MM_NOOP, MM_ECHO, MM_HCLOS, MM_MESS, MM_Repl, MM_ALRM, MM_PERR, MM_OPEN, MM_CLOSE, or MM_REJECT. These routines read the rest of the message and convert it to the proper format so that it may be sent to a Local MSG. The routines MM_NOOP, MM_ECHO, and MM_HCLOS do not convert the received message to Local MSG format, but act on the message immediately, ignoring it (MM_NOOP, which handles MSG-to-MSG NOOP messages), queueing an ECHO-REPLY message to be sent back (MM_ECHO, which handles ECHO messages), closing or setting up to close the MSG-to-MSG connection (MM_HCLOS, which handles CLOSEs).

The routine ChkExNm checks the process names in the MSG-to-MSG message for validity, and CnvExNm converts them from MSG-to-MSG format to internal format.

2.2.1.3 CMSGO

The module cmsgo.c handles outgoing messages from the local host to remote MSGs. It has three main routines:

- o HandleLMSGTraffic reads messages from the port, and returns the MCB for a message when it has read a complete one.
- o HandleRcvdInterMSGMessage dispatches on the op code of the message to the appropriate routine which queues the message for delivery to the remote host.
- o TryToSendMSGToMSGMessage takes messages off the alarm and message queues and attempts to send them to the remote MSG. If there is no connection to the remote MSG, or if TryToSendMSGToMSGMessage gets an error while writing a message, it will attempt to open a new connection to the remote host. If this attempt fails, all outstanding queued alarms and messages will be aborted (completed with a disposition of "Remote host unreachable."). TryToSendMSGToMSGMessage converts the message to MSG-to-MSG format before sending it by calling the routine ConvertToMSGToMSGMsg.

2.2.1.4 CLMSG

The module clmsg.c contains routines which send messages from the Central MSG to Local MSGs.

SendCLPEAborted sends a message telling the Local MSG to abort the specified Pending Event. It is called by TryToSendMSGToMSGMessage when it is unable to open a connection to the remote host, and also in response to an AbortPE message from the Local MSG (which will be sent if a PE times out or is rescinded).

SendCLMessage opens the destination Local MSG's receive port and writes the message (passed as an argument to SendCLMessage) to it. If it is unable to open the receive port, it will reject the message with a reason of "destination process unknown." It could be unable to open the port for at least two reasons: the remote MSG has specified an invalid process name, or the destination process has died.

2.2.1.5 CMSGNT

The module cmsgnt.c handles initial contact with a remote host.

The routine MakeContact attempts to open an MSG-to-MSG connection to the specified remote host, and if successful, goes through the SYNCH dialogue with that host by calling DoSynch.

DoSynch is called both by MakeContact, and by the ICP socket listener when it has received an ICP from a remote host. When called by MakeContact, it sends a SYNCH message and awaits a reply. When called by the ICP socket listener, it waits for the remote host to initiate the SYNCH dialogue, then sends a matching SYNCH message in reply.

2.2.2 Local MSG

The Local MSG consists of the following modules: lmsg.c, lmsgi.c, lmsgo.c, the utility routines lmutil.c, util.c, incnum.c, generic.c, getconfig.c, and the data abstractions ccb.c, queue.c, llist.c, streamio.c, and string.c A module dependency diagram for the Local MSG is shown in Figure 5. The principal modules of the Local MSG are described below.

2.2.2.1 LMSG

The module lmsg.c contains the initialization and main loop procedures for the Local MSG process.

The program is started at the procedure main, which reads the configuration file, starts up a Central MSG if there is not already one running, initializes dynamic data structures and sends the process name of this process (this is the name which will be returned by the MSG primitive WhoAmI) to the MSG primitive routines. It then opens its receive port and calls the routine LocalMSG to handle pipe and port traffic.

LocalMSG awaits activity on its receive port or the pipe from the MSG primitive routines. When activity is detected, it calls HandleAnyMessageFromPMSG (in lmsgo) and HandleAnyInterMSGMessage (in lmsgi) to handle any message that might have arrived. Both of these routines are called on any pipe or port activity, so the routines must check for activity which is of interest to them.

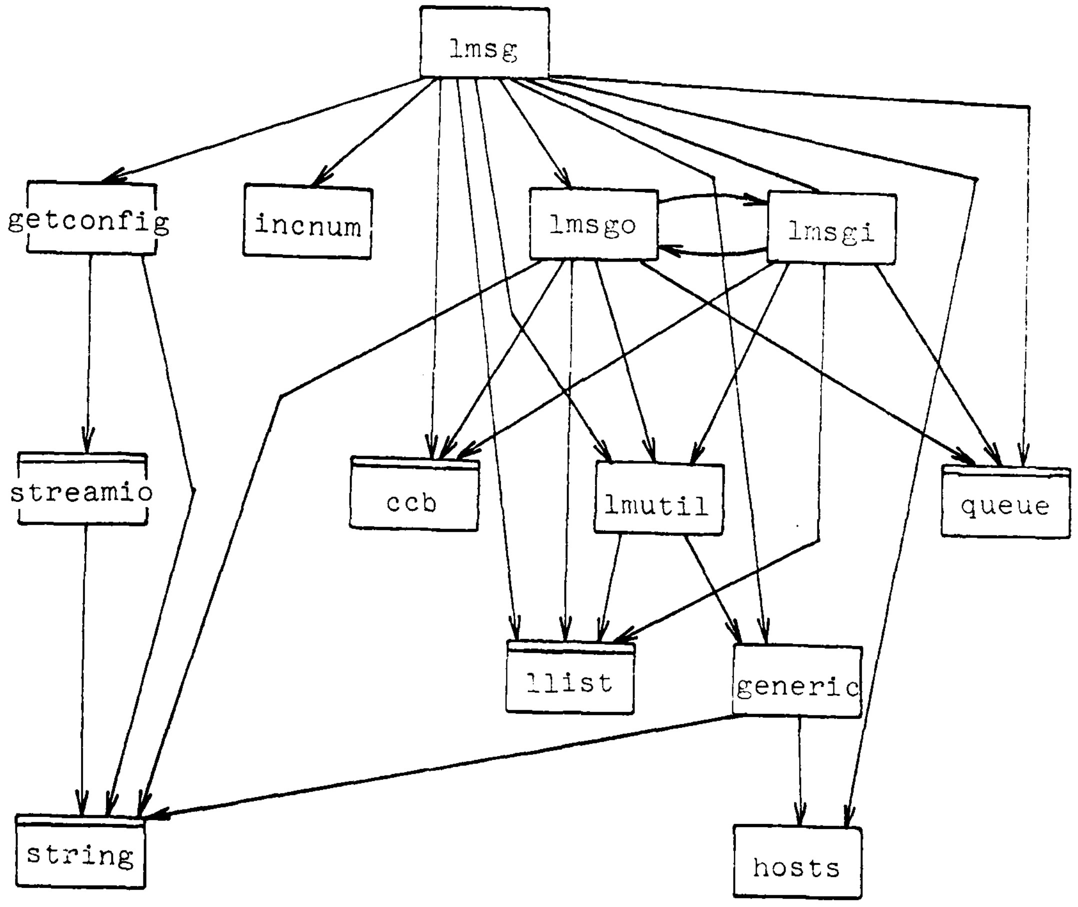


FIG. 5. MODULE DEPENDENCY DIAGRAM FOR LOCAL MSG

2.2.2.2 LMSGI

The module lmsgi.c contains routines that handle incoming message traffic from the Central MSG and other Local MSGs.

The main routine in lmsgi is HandleAnyInterMSGMessage, which reads messages from the receive port, and when it has read an entire message, calls HandleRcvdInterMSGMessage which dispatches on the op code to one of the routines L_SpecificMessage, L_GenericMessage, L_Alarm, L_ConnOpen, L_ConnClose, L_ConnReject, L_AbortPE, or CompleteThePE. These routines convert the received message to the proper format and pass it on to the user process. The L_Specific message routine may queue the message rather than sending it to the user process, if there is no outstanding ReceiveSpecific. Also, the routines L_ConnOpen, L_ConnClose, and L_ConnReject maintain theCCB for the user connection, and only send a message to the user process if a major change of state (such as the connection becoming fully open) occurs. See Figure 6 for a state transition diagram for CCBs.

The routine SendLPMessage is used to send a message over the pipe to the user process.

2.2.2.3 LMSGO

The module lmsgo.c handles outgoing traffic from the user process to other Local MSGs and the Central MSG. Its main routine is HandleAnyMessageFromPMSG, which reads a message from the pipe, then calls HandleRcvdMessageFromPMSG.

HandleRcvdMessageFromPMSG dispatches on the op code of the received message to one of P_SendSpecific, P_ReceiveSpecific, P_SendGeneric, P_ReceiveGeneric, P_SendAlarm, P_MessageAccepted, P_MessageRejected, P_AlarmAccepted, P_AlarmRejected, P_ConnOpen, P_ConnClose, P_Error, P_AbortPE, P_AlarmAborted, or P_StopMe. These routines convert the message to the proper format, create a pending event if necessary, and send the message on to the destination Local MSG, or the Central MSG (in the case of ReceiveGeneric and messages destined for remote hosts). In addition, the connection handling routines (P_ConnOpen, P_ConnClose, P_ConnReject) maintain the CCB for the user connection (see Figure 6), and pass information on to the user process if the received message causes a change of state, using the routines SocketToPMSG, SendCloseToPMSG, and SendRejToPMSG. (These routines are found in lmsgi.c).

SendLLMessage is used to send a message to another Local MSG

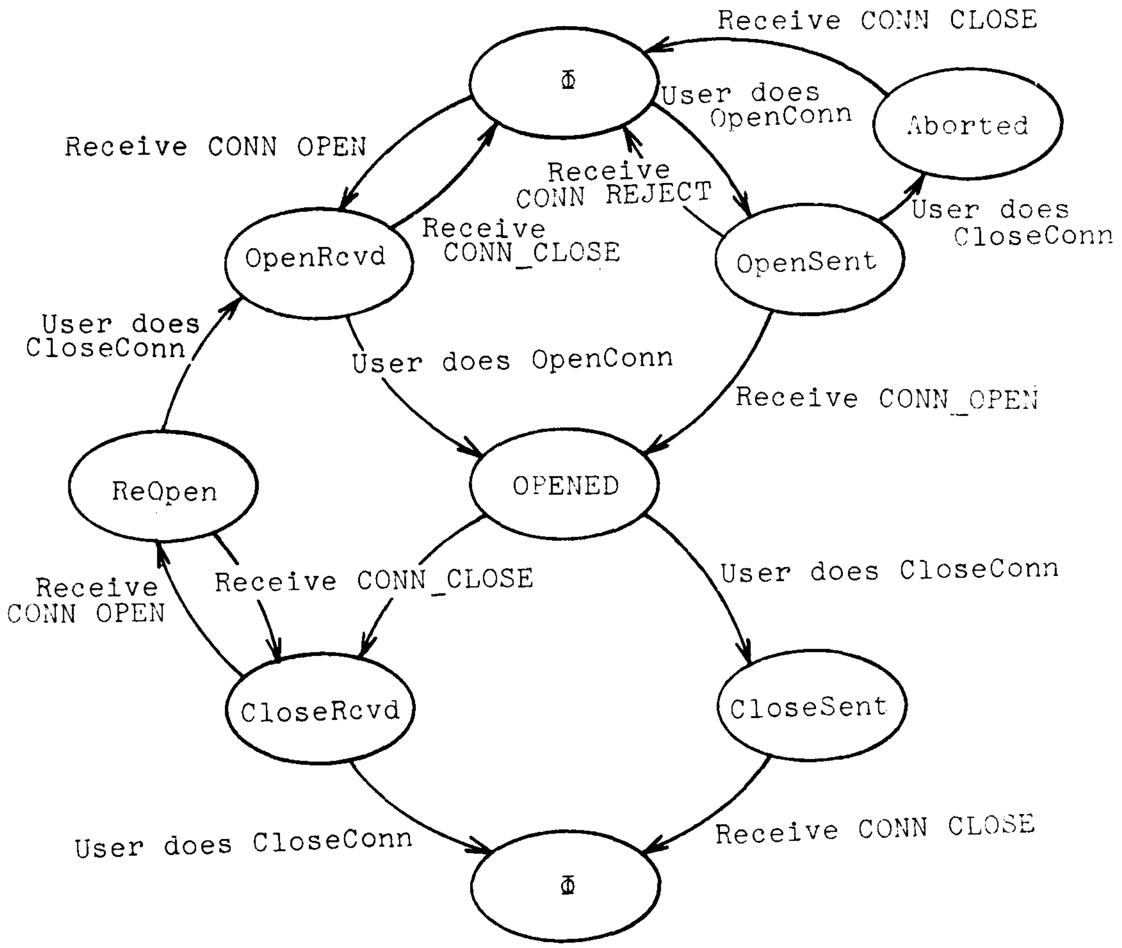


FIG. 6. CCB STATE DIAGRAM

or the Central MSG. If the destination process is the same as the sending (current) process, the message is not written to the port (which would cause the process to hang if the message to be sent were longer than the 1000 byte UNIX port buffer size), but is handled as if it had just been read from the port by calling the routine HandleRcvdInterMSGMessage. If the destination process is not the sending process, SendLLMessage opens the receive port by calling OpnSnPt, and writes the message to it.

The routine OpnSnPt checks whether the destination process is on the local host or not. If it is, and if the message is generically addressed, it opens the Central MSG receive port and returns the resulting file descriptor. If the message is specifically addressed, OpnSnPt opens the receive port of the destination Local MSG. Local MSG port names are formed from the generic class name of the process and its instance number (which is a word containing its UNIX process ID in the left byte and the 8 low order bits of the time when the process was started in the right byte). The actual form of the port name is "CLASS_NNNNNN", where CLASS is the first seven characters of the generic class name (this because UNIX filenames are limited to 14 characters in length), and NNNNNN is the octal representation of the instance number (without a leading zero). If the message is destined for a remote host, OpnSnPt attempts to open that host's Protocol process port (this port name is just the lower-case, standard ARPANET name of the remote host). If this attempt fails, it opens the Central MSG receive port.

If OpnSnPt is unable to open a port, it returns an error code. Otherwise, it returns the file descriptor for the port.

2.2.3 MSG Primitive Routines

These routines are loaded with the user process, and provide an interface between the user process and its associated Local MSG. They handle requests from the user in the form of C-language function calls, pipe traffic between the user process and the Local MSG, and timing-out of pending events. The MSG primitive routines consist of the modules inipmsg.c, pmusr.c, pmlmint.c, ptmo.c, the utility routines util.c, incnum.c, generic.c, getconfig.c, and the data abstractions pmpe.c, ccb.c, queue.c, streamio.c and string.c.

2.2.3.1 INIPMSG

This module contains initialization routines for MSG. The function InitMSG starts up a Local MSG (by calling the function MakProc) and initializes all internal data structures. It returns the file designator of the receive pipe from the LMSG. This file descriptor is used by some programs (e.g. the Front End) to determine when MSG needs to gain control without having to relinquish control as would happen if they called ReqSig.

The MakProc routine forks to create a Local MSG process, giving that Local MSG the Generic Class name that was specified in the call to InitMSG.

2.2.3.2 PMUSR

This module contains the MSG primitive routines that are callable by the user process, and routines which check the validity of the arguments given by the user.

Those MSG primitive routines which create Pending Events (see Section 3.2.1 of the UNIX MSG User Manual) first check their arguments for validity, using the routines ChkTimo, CheckMessage, CheckRetWord, CheckSpclHandling, CheckBuffer, CheckFlag, ChkSig, CheckAndConvertSpecificName, CheckAndConvertGenericName, CheckConnType, and ValidHost. If an illegal argument value is found, the routines call DoPMSG (see below), and return an error code. If no error is found, a new Pending Event is created and placed on the timeout queue. Then a message is sent to the Local MSG specifying the operation it is to perform. Depending on the signal value which was specified, the routines then either wait for the Pending Event to complete (Unblock), or call DoPMSG and then return an event handle to the caller.

The routine DoPMSG checks to see if any PEs have completed or timed out, and if so, delivers the disposition and does whatever else is appropriate based on the primitive type and the signal specified.

The AcceptAlarms primitive checks the validity of its argument, and if it is valid, AcceptAlarms set the global variable IsAcceptingAlarms to the value of the argument. It then calls DoPMSG and returns.

WhoAmI simply copies the process name which was sent to it by the Local MSG when InitMSG was called into the space specified by the argument.

Rescind looks for a PE with the specified event handle. If no such PE is found, an error is returned. Otherwise, it checks to see whether it can still rescind the event, and returns an error if not. If the event can be rescinded, Rescind sets its disposition to the value vdispAborted, changes its signal to null so that it will not be queued when it is signalled, and sends a message to the Local MSG to rescind the PE. It then calls DoPMSG and returns. The Local MSG will send back a message which completes the PE and causes it to be released.

ReqSig first checks whether any PEs exist at all. If not, it returns an error. Otherwise, it looks for a completed EnableAlarm PE (there can be at most one of these). If one of these is found, it is released and its event handle is returned. If there is no completed EnableAlarm PE, ReqSig looks for other completed PEs, and returns the event handle of the first one it finds if it finds any.

If there are no completed PEs when ReqSig is called, it creates a new pending event with a timeout period as specified in the argument to ReqSig. It then waits for a pending event to complete or time out. When an event completes, ReqSig checks to see if the completed event is the new PE it created. If so, it returns a response code indicating that no PEs completed. Otherwise, it removes the new PE from the timeout queue and returns the event handle of the PE that completed.

StopMe sends a message to the Local MSG, then waits for a response which tells it that it may call exit.

2.2.3.3 PMLMINT

The module pmlmint.c contains routines that interface between the user process and the Local MSG.

The routines StartMessageToLocalMSG, SendWordToLocalMSG, SendNameToLocalMSG, SendMessageToLocalMSG, and EndMessageToLocalMSG are called by the primitive operations in pmusr.c, and buffer a message being sent to the Local MSG to reduce the number of write system calls.

HandleAnyMessageFromLMSG reads a message header from the pipe and dispatches on its op code to one of the routines MessageRcvd, AlarmRcvd, OpenRcvd, CloseRcvd, RejectRcvd, PECompleted, HandleLPError, AbortAlarm, or YourName. These routines read the rest of the message, deliver any received message, remote socket number or alarm code where appropriate, and signal the Pending Event that has now completed (except for Error, Abort Alarm, YourName and StopMe messages, a message from the Local MSG always causes a PE to complete.)

A PECompleted message will be received for a SendGeneric, SendSpecific, or SendAlarm which has been acknowledged by the destination process. It will also be received if a pending event is rescinded or times out, or if some error keeps an event from completing properly (e.g. foreign host is dead, or a MESS_REJECT is received for a message).

2.2.3.4 PTMO

The module ptmo.c contains routines for timing out of pending events. It maintains a queue of pending events in the order in which they are scheduled to time out.

StartToTimeoutPE adds a PE to the timeout queue, inserting it after any other pending events which are scheduled to time out at the same time or earlier than the new PE is scheduled to time out.

SecondsToNextTimeout returns the number of seconds until the next pending event on the queue is scheduled to timeout. If the queue is empty when SecondsToNextTimeout is called, the value vMaxTimeout is returned.

EndTimeoutOfPE removes a pending event from the timeout queue.

HandleAnyTimeout checks to see if any pending events have timed out, and if so, removes them from the timeout queue and either signals them (if no message was sent to the Local MSG when this PE was created - only true for EnbAlrm and ReqSig type PEs), or sends a message to the Local MSG to time out the PE. If the Local MSG is able to abort the PE, it will send back a PECompleted message with a disposition value of "timed out," and the PE will be signalled then. Each time HandleAnyTimeout is called, it will handle all PEs that timed out since it was last called.

2.2.4 Utility Routines

2.2.4.1 UTIL

The module util.c contains utility routines used by all MSG components. Calls are:

CharsToBeRead(FileDesc: int) returns int
Returns the number of characters which may be read from FileDesc without blocking.

SendCapacity(FileDesc: int) returns int
Returns the number of characters which may be written to FileDesc without blocking.

minimum(Int1: int, Int2: int) returns int
Returns the (signed) minimum of Int1 and Int2.

FlushInputChars(FileDesc: int, CharCount: int)
Reads CharCount bytes from the file FileDesc and ignores them.

oitoa(Int: int, Ascii: char *) returns int
Converts the integer Int to its unsigned octal ascii representation (oitoa does not include a leading zero), leaving the result in the area pointed to by Ascii. Returns the number of characters in the result string.

ditoa(Int: int, Ascii: char *) returns int
Converts the integer Int to its signed decimal ascii representation, leaving the result in the area pointed to by Ascii. Returns the number of characters in the result string.

ErrorCheck(ArgList)
The MSG ErrorCheck routine. Prints message and returns to caller. ArgList is an argument list which is passed to printf.

ErrorHalt(ArgList)
The MSG ErrorHalt routine. Prints message and aborts, dumping core contents. ArgList is passed to printf.

MoveBytes(From: char *, To: char *, Count: int)
Moves the number of bytes specified by the Count argument from the area pointed to by From to the area pointed to by To. Does not check for conflicting overlap.

ClrBlk(Start: char *, Count: int)
Clears (sets to zero) Count bytes, starting at the location Start.

ReadAll(FilDes: int, Buf: char *, NumToRead: int) returns int
Reads from FilDes into Buf until NumToRead characters are read or EOF is encountered. Returns the number of bytes actually read. This call is useful since the normal UNIX read call may not actually read the number of bytes specified. Unless an EOF is encountered, this call will not return until it has read the specified number of bytes.

InputNum(InputBuf: buf *) returns int
Reads from the buffered-input buffer InputBuf a number string, and converts it to an int. A number string is any

sequence of digits, ignoring leading spaces, tabs, and newline characters, and terminates on the first character which is not a digit. If the first (non-blank) character is a zero ('0' - not null), the number is assumed to be octal, else decimal. The resulting integer is returned.

InputString(InputBuf: buf *, StringPointer: char *) returns int
Reads a string from the buffered input file InputBuf, putting the result in StringPointer. A string is any sequence of characters (ignoring leading spaces, tabs, and newlines) which terminates with a string terminator character (i.e. a character for which the IsStringTerminator function returns true). Returns the number of characters read.

InputNumString(InputBuf: buf *, StringPointer: char *) returns int
Reads a number string (sequence of digits) from InputBuf until a non-digit is read, and puts the result in StringPointer. Returns the number of characters read.

StringToDoubleInt(NumString: char *, NumChars: int, DoubleIntP: long *) returns int
Converts the string specified by NumString, which is assumed to have length NumChars, to a 32-bit integer. Used to read contact socket numbers for hosts specified in the host configuration file.

IsAlphaNumeric(c: char) returns boolean
Returns true if c is an upper or lower case alphabetic character, or a digit between 0 and 9 (inclusive). Else returns false.

IsStringTerminator(c: char) returns boolean
Returns true if c is a "string terminator" - a space, comma, carriage return, newline, or tab. Else returns false.

SkipToEndOfLine(InputBuf: buf *)
Reads characters from InputBuf until a newline character has been read, or end-of-file has been reached.

SwapByt(addr: char *) returns int
Swap bytes of a word. addr points to a word (which need not begin on a word boundary) in memory. SwapByt returns the word pointed to by addr, with its bytes swapped, leaving the original word unchanged.

2.2.4.2 LMUTIL

The module lmutil.c contains utility routines which are used by the Central and Local MSGs, but not the MSG Primitive routines.

RdBytes(FilDes: int, Buf: char *, NumToRead: int)
Similar to ReadAll (Section 2.2.4.1), but ErrorHalts if it is unable to read NumToRead bytes from the file FilDes.

ConvertPToLMessage(Pmsg: PLMsg *, SourceName: InternalMSGName *) returns LLMsg *
Converts Pmsg, which is a message in PLMsg format (see Section 3.1.1.2) to LLMsg format, by moving the plLength through plArg fields of Pmsg back by vInternalMSGNameSize bytes, and copying SourceName into the resulting hole. The IsFromPMSG flag must have been true when the MCB which holds Pmsg was created. Returns a pointer to the (new) start (llLength field) of the message.

ConvertLToPMessage(Lmsg: LLMsg *) returns LPMsg *
Converts the LLMsg Lmsg to LPMsg format by moving the llLength through llArg fields up by vInternalMSGNameSize bytes, thus overwriting the llSourceName field. A pointer to the new start of the message (the address of the lpLength field) is returned.

NameIsSelf(DestName: InternalMSGName *, LocalPCB: PCB *) returns boolean
Compares the InternalMSGName specified by DestName with the pcbProcessName component of LocalPCB and returns true if they are equal, else false.

ValidDestName(DestName: InternalMSGName *, LocalPCB: PCB *) returns boolean
Used by Local MSGs to check if a message has been sent to the wrong Local MSG. Compares the generic code in the pcbProcessName component of LocalPCB with that specified in DestName, and checks that the process number in Destname is either 0 (as it would be for a generically addressed message) or the same as the process number in the pcbProcessName of LocalPCB. If one of these tests fails, false is returned, else true.

ValidMSGSource(MsgHeader: PortHeader *) returns boolean
Checks that the received message specified by MsgHeader came from a valid process. Currently always returns true.

MkPtNam(<u>WorkDir</u>: char *, <u>ProcName</u>: InternalMSGName *, <u>PortName</u>: char *) returns char *

Builds in the <u>PortName</u> argument the UNIX pathname of the port for the process specified by the <u>ProcName</u> argument. <u>WorkDir</u> specifies the MSG working directory and is prepended to the rest of the port name, which consists of the first seven characters of the generic class name of that process, followed by an underscore ("_"), followed by the octal representation (without a leading zero) of the instance number of the process. Returns a pointer to the result (i.e. returns the <u>PortName</u> argument).

NamesMatch(<u>Name1</u>: InternalMSGName *, <u>Name2</u>: InternalMSGName *) returns boolean

Compares the two InternalMSGNames <u>Name1</u> and <u>Name2</u> and returns <u>true</u> if they name the same process, and <u>false</u> otherwise.

2.2.4.3 INCNUM

The module incnum.c contains routines for handling the incarnation number file. The incarnation number file is a text file containing a single line of text, which contains the decimal ascii representation of the current incarnation number followed by a carriage-return and line-feed. Local MSGs and the MSG Primitive routines use the function GetIncarnationNumber (described below) to read the incarnation number file. When it starts up, the Central MSG calls the routine InitIncarnationNumber, which increments the incarnation number in the incarnation number file.

InitIncarnationNumber(<u>IncFile</u>: char *)

Reads the incarnation number from the file specified by <u>IncFile</u>, increments the number found there and writes it back. Saves the new incarnation number in the global <u>LocalHostIncarnation</u>. ErrorHalts if it cannot open the file specified by <u>IncFile</u>.

GetIncarnationNumber(<u>IncFile</u>: char *)

Reads the incarnation number from the file specified by <u>IncFile</u> and stores it in the global <u>LocalHostIncarnation</u>. ErrorHalts if it is unable to open the file specified by <u>IncFile</u>.

2.2.4.4 GETCONFIG

The module getconfig.c contains a single entry, GetConfig, which reads the MSG configuration file. This file contains configuration-dependent information, such as the name of the MSG working directory and the number of the MSG contact socket. The format of the configuration file is described in the UNIX MSG User Manual.

GetConfig is passed two array pointers as arguments. The first of these contains strings which specify the keyword to match. These strings need not be in any particular order, as the entire array is checked for each line that is read from the file. The second array contains pointers to areas where the rest of the line which contains the keyword (excluding any spaces or tabs after the colon which delimits the keyword) is to be placed.

The configuration file is expected to be found in the directory where the MSG component was started, and is named config.file. If no such file is found in the connected directory, GetConfig looks for a file named config.file in the directory /usr/nswmsg - that is, it looks for the file named /usr/nswmsg/config.file.

GetConfig returns <u>true</u> (-1) if it was able to open the configuration file, and <u>false</u> (0) if not. Note that a value of <u>true</u> returned by GetConfig does not necessarily mean that all (or even any) keywords were found in the file - only that it was able to open the file.

PRECEDING PAGE BLANK-NOT FILMED

3. INPUT/OUTPUT DESCRIPTIONS

This section provides detailed information on the structure and composition of data used by the UNIX MSG implementation.

3.1 General Description

3.1.1 Input/Output

MSG is a "system program" which functions more like an operating system than like a language processor, such as a compiler, or a scientific program, such as a statistics package. Therefore, the MSG implementation does not deal with input/output in the "traditional" sense of taking a set of inputs and processing it to produce a set of outputs.

MSG does, however, accept input from several configuration files to control certain aspects of its operation. These files are described below.

3.1.1.1 Configuration Files

The file config.file contains information such as the pathname of the Central MSG executable file, the pathnames of the Generic names and Network Configuration files, and the MSG contact socket (see the UNIX MSG User Manual for a detailed description of the configuration files). This file is read by the routine GetConfig (see Section 2.2.4.4.

The Generic names file declares the process classes to be known to MSG and details their allocation procedures. This file is read by the routine InitGenericNames in generic.c.

The network configuration file specifies the MSG contact sockets for the ARPANET hosts which are to be part of the configuration. This file is read by the routines InitHosts in hosts.c.

3.1.1.2 Inter-component messages

MSG processes communicate with each other over UNIX pipes and ports. All these messages are in a standard format, described below:

The structure of user process-to-Local MSG messages is:

```
struct PLMsg
{
  int plLength;      /* Total length in chars */
  int plOpCode;      /* Message type */
  int plID;          /* Transaction ID of message*/
  int plLID;         /* Not used here */
  int plArg;         /* Argument word */
  struct InternalMSGName plDestName;
                     /* Destination process */
  int ConnType;      /* Conn type for OpenConn */
                     /*-Message starts here for */
                     /*-SendGeneric/SendSpecific */
  long plSocket;     /* Local socket # for OpenConn */
};
```

The argument word contains the Conn ID (for OpenConn/CloseConn), the special handling (for SendGeneric/SendSpecific), the alarm code (for SendAlarm), or the reason (for error messages).

LMSG-to-user process messages have the following format:

```
struct LPMsg
{
  int lpLength;      /* Total length in chars */
  int lpOpCode;      /* Message type */
  int lpID;          /* Transaction ID of mess */
  int lpLID;         /* Local MSG PE ID for mess */
                     /*-acknowledge from PMSG */
  int lpArg;         /* Argument */
  struct InternalMSGName lpSourceName;
                     /* Source process name */
  long lpSocket;     /* Remote socket # for OpenConn */
                     /*-Also (int) Reject reason for */
                     /*-ConnReject, Close disp. for */
                     /*-ConnClose */
                     /*-Message starts here if */
                     /* generic or specific mess */
};
```

The argument word contains the Connection ID (ConnOpen/ConnClose/ConnReject), special handling specific message), the qwait flag (generic message), the disposition (PE completed message), or the alarm code (alarm message).

Local MSG-to-Local MSG, Local MSG-to-Central MSG and Central MSG-to-Local MSG messages all have the same format. These messages are similar in format to the above messages, but they have two process names instead of one. The format of an LMSG-to-LMSG message is:

```
struct LLMsg
{
  int llLength;
  int llOpCode;
  int llID;
  int llLID;
  int llArg;
  struct InternalMSGName llSourceName;
  struct InternalMSGName llDestName;
  int llConnType;     /* Also Close reason, */
                      /*-reject reason, and */
                      /*-start of message... */
  long llSocket;
};
```

The argument word may contain the same things as the LPMsg argument word.

3.1.2 Internal MSG Data

To support its operation MSG maintains two kinds of internal data:

- o Static data.

  Internal static data items are those that do not change as a result of MSG execution after a process has been initialized. Included are such items as the host tables (Sections 3.2.2.1, 3.2.3.1 and 3.2.4.1), the generic names tables (Sections 3.2.2.2, 3.2.3.2 and 3.2.4.2), the Host Control Block for a Protocol process (Section 3.2.2.4), and the host incarnation number, which is kept in a global symbol named <u>LocalHostIncarnation</u>.

- o Dynamic data.

  Dynamic data includes items that change as a result of MSG execution. These include Pending Events (Sections 3.2.2.3, 3.2.3.3 and 3.2.4.3), Message Control Blocks (Sections 3.2.2.5 and 3.2.3.4), Connection Control Blocks (Sections 3.2.3.5 and 3.2.4.4), network transmission queues (the heads of these queues are contained in the Host Control Block), generic message queues (Section 2.2.1.1), and so forth.

3.2 Data Structures

This section describes the structures and internal data used in the MSG implementation, and the operations on these data items.

3.2.1 General

3.2.1.1 Linked Lists

Linked lists are implemented by the routines in llist.c.Each list element contains a right (forward) and left (backward) pointer. The last entry on the list has a right pointer which points back to the first element, and the first list element's left pointer points to the last element on the list. (See Figure 7). The structure of a list element is:

```
struct LE
{
  struct LE *leRLink;   /* Forward pointer */
  struct LE *leLLink;   /* Backward pointer */
};
```

List operations are:

```
InitLst(List: LE **)
     assigns: *List
```

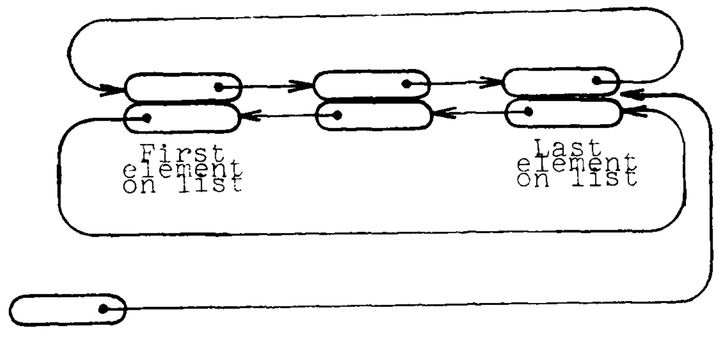


FIG. 7. A LINKED LIST

Makes List an empty list. An empty list is represented by a list pointer (the List argument contains the address of the list pointer) containing the value vNoLEntry.

AddToList(List: LE **, NewEntry: LE *)
assigns: *List
modifies: List
Adds NewEntry to the end of list List.

RmvFrmList(List: LE **, OldEntry: LE *)
requires: OldEntry on list
modifies: List
Removes OldEntry from the list List. ErrorHalts if list pointers are inconsistent.

FrstOnList(List: LE **) returns LE *
modifies: List
Removes and returns the first entry on the list List, or the value vNoLEntry if the list is empty.

IsOnList(List: LE **, TestLE: LE *) returns boolean
Returns true if TestLE is on List, else false. If List is empty, returns false.

NextOnList(List: LE **, CurEntry: LE *) returns LE *
Returns (but does not remove from the list) the next entry after CurEntry on the list List. If CurEntry is vNoLEntry, the first element on the list is returned.

3.2.1.2 Queues

Queues are implemented by the routines in queue.c. A queue is a singly-linked list. The last entry on the list points back to the first entry. (See Figure 8). The Queue Head pointer points to the last element on the queue. This makes inserting at the end of the queue easier. The structure of a queue element is:

```
struct QE
{
  struct QE *QELink;  /* Link to next in queue */
};
```

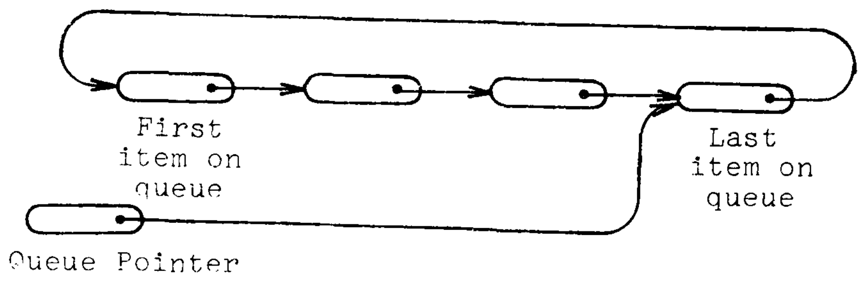


FIG. 8. A QUEUE

Queue operations are:

InitQ(QHead: QE **)
assigns: *QHead
Makes QHead an empty queue. An empty queue is represented by a queue pointer (the QHead argument contains the address of the queue pointer) containing the value vNoQEntry.

AddToQ(QHead: QE **, QEntry: QE *)
assigns: *QHead
modifies: QHead

Adds QEntry to the end of the queue pointed to by QHead.

FrstOnQueue(QHead: QE **) returns QE *
    If QHead is an empty queue, returns the value vNoQEntry.
    Else returns the first element on the queue pointed to by
    QHead, and removes it from the queue.

RmvFrmQueue(QHead: QE **, QEntry: QE *)
    assigns: *QHead
    modifies: QHead
    If QEntry is on the queue pointed to by QHead, it is removed
    from the queue. If QEntry is not on the queue (which
    includes the case where the queue is empty), the queue is
    left unchanged.

3.2.2 Central MSG

3.2.2.1 Host Table

The Central MSG's host table consists of two arrays: HostNum[], which contains the host numbers of those hosts known to MSG, and Socket[], which contains the MSG contact socket for the host with the same index in the HostNum array.

These arrays are initialized at startup by the routine InitHosts(HostFile), where HostFile is a string which specifies the name of the Network Configuration File (see Section 3.1.5 of the UNIX MSG User Manual). InitHosts also initializes the global variable LocalHostNumber to contain the address of the local host. InitHosts is found in hosts.c

3.2.2.2 Generic Names Table

The generic names table consists of the following arrays:

GnCdVec[]	Contains the generic code for the generic class.
GnNmVec[]	Contains the generic class name corresponding to the generic code with the same index in GnCdVec[].
GnNmCVec[]	Contains the length of the generic class name with the same index in the GnNmVec array.

GnCrSp[]	Contains the Create-Spec (see Section 3.1.5 of the UNIX MSG User Manual) for this generic class.
GnCrMd[]	Contains the Create-Mode (see Section 3.1.5 of the UNIX MSG User Manual) for this generic class.
GnRnFlVec[]	Contains the pathname of the file which implements this generic class, if the corresponding element of the GnCrSp array does not identify this as a "RemoteProc."
GnTrmSp[]	Contains the Terminate-Spec (see Section 3.1.5 of the UNIX MSG User Manual) for this generic class, unless this class is identified as a "RemoteProc."
GnRmHsts[][]	Contains the host numbers of the remote hosts this class is defined to run on, if this class has a GnCrSp entry of "RemoteProc." (Note that this is a two-dimensional array. The first dimension is the index which corresponds to the GnCdVec array, and the second index holds a list of host numbers.)

All of these arrays use the same index for a given class, so if, for example, GnCdVec[1] = 3, then GnNmVec[1] will contain the string "FOREMAN".

Two other arrays, RcvGenrs[] and RcvdGenrs[] also use the same index, and contain, respectively, a queue of outstanding RcvGenr primitives for the class specified by the corresponding GnCdVec[] entry, and a queue of received generic messages for the class (see Section 2.2.1.1). If one of these queues is non-empty, the other must be empty.

The generic names table is initialized by the routine in generic.c called InitGenericNames(GenFile), where GenFile is a string containing the pathname of the Generic Name File (see Section 3.1.5 of the UNIX MSG User Manual).

The global variable NmGnNames is set to the number of defined generic classes when InitGenericNames is called.

3.2.2.3 Pending Events

In the Central MSG, Pending Events (PEs) are used to keep track of messages received from and being sent to remote hosts. The structure of Central MSG Pending Events is:

```
    struct PE
    {
      struct LE peListEntry;
                          /* Doubly linked list pointers */
                          /*-Also single linked queue pntr */
      int peID;           /* PE ID (offset in the in-use */
                          /*-PE table) */
      int peSourceID;     /* Source ID of message */
      struct MCB *peMCB;  /* Associated MCB (if any) */
      char peState;       /* PE state code */
      char peType;        /* PE Type code */
    };
```

PEs are implemented in the module lmutil.c. PE operations are:

GetNewPE(<u>TypeCode</u>: int, <u>AssociatedMCB</u>: MCB *) returns PE *
    Allocates a new Pending Event from the free storage pool, ErrorHalting if there is not enough free memory available. Adds the new PE to the active PE list (see below), reserves the MCB <u>AssociatedMCB</u>, and initializes the PE entries. Returns a pointer to the new PE.

ReleasePE(<u>OldPE</u>: PE *)
    If <u>OldPE</u> is <u>vNoPE</u>, does nothing. Else removes <u>OldPE</u> from the list of known PEs, releases the associated MCB entry, and returns the storage used for the PE to the free storage pool.

InitPEList()
    Initializes the active PE list. PEs are put on this list when created, and removed when the PE is released.

AddToPEList(<u>NewPE</u>: PE *)
    Adds the PE <u>NewPE</u> to the active PE list. ErrorHalts if there is no room on the PE list for more PEs.

RemoveFromPEList(<u>OldPE</u>: PE *)
    Removes the PE <u>OldPE</u> from the active PE list.

GetOldPE(<u>List</u>: LE **, <u>ID</u>: int, <u>argSrc</u>: int) returns PE *
    Searches the active PE list for a PE with the specified

Source ID (if argSrc is true) or PEID (if argSrc is false), and if found removes the PE from the list List and returns a pointer to it. If the PE is not found, MSG ErrorChecks and returns vNoPE.

SearchForPE(List: LE **, SrcID: int) returns PE *
Searches the list List for the PE with ID SrcID, and if found, removes it from the list and returns a pointer to it. If the specified PE is not on the list, vNoPE is returned.

FindProcessPE(ProcessID: int) returns PE *
Searches the active PE list for a PE with a Source ID of ProcessID and returns a pointer to it if found, else returns vNoPE.

FindLocalPE(SourceID: int, SourceName: InternalMSGName *) returns PE *
Searches the active PE list for a PE with a Source ID of SourceID and whose peMCB entry has a source name of SourceName. If found, a pointer to the PE is returned, else vNoPE.

FindPEFromID(SourceID: int, SourceName: InternalMSGName *) returns PE *
Internal routine which does the work for FindProcessPE and FindLocalPE. If SourceName is zero, as it is when this routine is called by FindProcessPE, the SourceName field of the peMCB is not checked. Otherwise it is compared to SourceName. As for FindProcessPE and FindLocalPE, if the PE is found, a pointer to it is returned, else vNoPE is returned.

3.2.2.4 Host Control Blocks

Host Control Blocks (HCBs) keep track of information about remote hosts known to MSG. This information includes the remote host's host number, its incarnation number, the file descriptors for the connection (if any) to the host, and a list of PEs which have been queued to be sent to the host. The structure of a Host Control Block is:

```
struct HCB
{
  long hcbHostNumber;   /* Host address number */
  int hcbAlias;         /* 16-bit alias for this host */
  int hcbState;         /* HCB state */
  int hcbIncarnation;   /* Remote host incarnation num */
  int hcbRecvFileDes;   /* Receive file descriptor */
  int hcbSendFileDes;   /* Send file descriptor */
  long hcbSocket;       /* Contact Socket */
  struct QE *hcbToBeSentPEQ;
                        /* Queue of PEs to be sent */
  struct QE *hcbAlarmSendPEQ;
                        /* Queue of alarm PEs to send */
  struct LE *hcbPendingRemotePEList;
                        /* List of remote PEs pending */
                        /*-completion. */
};
```

There is one Host Control block in each CMSG Protocol process. This HCB is called GlHCB, and the global variable HostCB contains a pointer to it.

In order to give priority treatment to alarms, the hcbAlarmSendPEQ is always scanned before the hcbToBeSentPEQ when looking for a message to send to the remote host.

There are no operations on HCBs.

3.2.2.5 Message Control Blocks

Message Control Blocks (MCBs) are used to hold the text of MSG messages in transit. The structure of an MCB is:

```
struct MCB
{
  int mcbCharsToBeRead;
                       /* Number of characters yet to be */
                       /*-read (sent). */
  int mcbToBeSent;     /* Initial value of */
                       /*-mcbCharsToBeSent in case */
                       /*-have to retransmit mess */
  char *mcbBufPointer;
                       /* Buffer pointer. Points to end */
                       /*-of buffer when full. */
  int mcbProcessID;    /* Source Process ID */
  int mcbUseCount;     /* Use Count */
  int mcbInNetFormat;
                       /* true => in MSG-to-MSG format */
                       /*-false => in internal format. */
  char *mcbExtMessage;
                       /* Extension message pointer */
  int mcbExtMessageSize;
                       /* Size of extension message */
  char mcbLLContents[0];
                       /* Start of LMSG-LMSG message */
                       /*-(or MSG-to-MSG message) */
};
```

Operations on MCBs are:

InitPendingMCBRoutines()
Initializes the Pending MCB table. This table is used to keep track of MCBs which hold messages in the process of being read from the port. Since a port message may come in several pieces, and more than one process may write into a port, it is necessary to keep track of a partially read message from one process while handling messages from others.

RememberPendingMCB(<u>PndngMCB</u>: MCB *)
Puts the MCB <u>PndngMCB</u> in the Pending MCB table, where it may be found with the FindPendingMCB function. ErrorHalts if there is no free slot in the Pending MCB table.

FindPendingMCB(<u>Header</u>: PortHeader *) returns MCB *
Searches the Pending MCB table for an MCB with the process ID specified in the <u>Header</u> argument. If found, the MCB is removed from the table and returned, else the value <u>vNoMCB</u> is returned.

GetMCB(MessageLength: int, SourceProcessID: int, IsFromPMSG: int) returns MCB *

Gets a new MCB (by calling GetNewMCB), and initializes the mcbProcessID, mcbBufPointer, and mcbCharsToBeRead fields. Also puts the MessageLength in the buffer as the first two bytes of the message. A pointer to the new MCB is returned.

GetNewMCB(MessageLength: int) returns MCB *

Allocates space for a new MCB from the free storage pool, ErrorHalting if there is no space available. The size of the MCB allocated is MessageLength plus the MCB header size (The MCB structure fields before mcbLLContents make up the MCB header.) Sets the mcbUseCount field of the new MCB to 1, and clears the rest of the MCB header. Returns a pointer to the new MCB.

ReserveMCB(TheMCB: MCB *)

Increments the use count of TheMCB. Used by routines which are passed an MCB and require later use of it, even if after they return, the caller releases the MCB.

ReleaseMCB(OldMCB: MCB *)

Decrements OldMCB's use count, and if the resulting value is zero, returns the storage used by OldMCB to the free storage pool. If OldMCB is vNoMCB, no action is taken.

3.2.3 Local MSG

3.2.3.1 Host Table

The Local MSG's host table consists of the array HostNum[], which contains the host numbers of those hosts known to MSG.

This array is initialized at startup by the routine InitHosts(HostFile), where HostFile is a string which specifies the pathname of the Network Configuration File (see Section 3.1.5 of the UNIX MSG User Manual). This routine is found in the module hosts.c.

3.2.3.2 Generic Names Table

The Generic Names table in the Local MSG is identical to that in the Central MSG as described in Section 3.2.2.2.

3.2.3.3 Pending Events

In the Local MSG, Pending Events keep track of both messages received from other Local MSGs (or MSGs on remote hosts, via the Central MSG), and pending MSG primitives that were issued by the user. They are implemented in the module lmutil.c. The format of a PE is:

```
struct PE
{
  struct LE peListEntry;
                        /* Doubly-linked list pointers */
                        /*-Also single linked queue ptr */
  int peID;             /* PE ID */
  int peSourceID;       /* Source ID of message */
  struct MCB *peMCB;    /* Associated MCB (if any) */
  struct CCB *peCCB;    /* Associated CCB (if any) */
  struct PCB *pePCB;    /* Associated PCB */
  char peState;         /* PE state code. */
  char peType;          /* PE type code */
};
```

Operations are the same as for Central MSG Pending Events (See Section 3.2.2.3).

3.2.3.4 Message Control Blocks

MCBs in the Local MSG are identical to, and have the same operations as, MCBs in the Central MSG (described in Section 3.2.2.5), except that in the Local MSG the mcbInNetFormat, mcbToBeSent, mcbExtMessage and mcbExtMessageSize fields are not defined. The structure is:

```
struct MCB
{
  int mcbCharsToBeRead;
  char *mcbBufPointer;
  int mcbProcessID;
  int mcbUseCount;
  char mcbLLContents[vDifSizeLL_LP];
                       /* Start of LMSG-LMSG message */
  char mcbLPContents[0];
                       /* Start of LMSG-PMSG message */
};
```

3.2.3.5 Connection Control Blocks

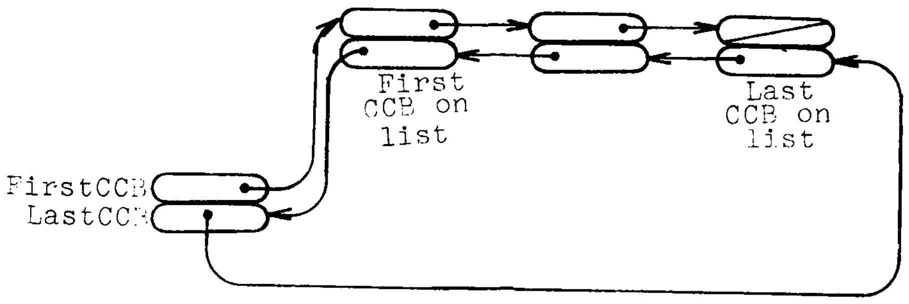


FIG. 9. CCB LIST

Connection Control Blocks (CCBs) are used to keep track of information necessary to maintain user direct connections. CCBs are kept on a doubly-linked list (not the same as a standard linked list - See Figure 9), where they may be found by the GetOldCCB and GetSpecCCB routines. CCBs are put on this list when created and not removed until released. CCBs are implemented in ccb.c. The structure of a CCB is:

```
struct CCB
{
  struct CCB *ccbNextCCB;  /* Next CCB in chain */
  struct CCB *ccbPrevCCB;  /* Previous CCB */
  int ccbConnID;           /* User connection ID */
  int ccbConnType;         /* Type/size of connection */
  int ccbState;            /* Current CCB state */
  int ccbLocalID;          /* Local ID */
  int ccbRemoteID;         /* Remote ID */
  long ccbLSocket;         /* Local socket # */
  long ccbRSocket;         /* Remote socket # */
  struct PE *ccbOpenPE;    /* PE for OpenConn */
  struct PE *ccbClosePE;   /* PE for CloseConn */
  struct CCB *ccbSecCCB;   /* Secondary CCB (if any) */
  struct InternalMSGName ccbFrnProcName;
                           /* Remote process name */
};
```

Operations on CCBs are:

InitCCBList()
    Initializes the CCB list.

GenerateNewCCB(LocalID: int, ConnID: int, DestName: InternalMSGName *) returns CCB *
    Allocates a new CCB from the free storage pool, fills in the LocalID, ConnID, and FrnProcName fields and adds the CCB to the CCB list. A pointer to the new CCB is returned.

GetOldCCB(ConnID: int, DestName: InternalMSGName *) returns CCB *
    Searches the CCB list for a CCB with the given ConnID and FrnProcName. If such a CCB is found, a pointer to it is returned. Otherwise, the value vNoCCB is returned. This call is used when the LocalID for the connection is not known.

GetSpecCCB(LocalID: int) returns CCB *
    Searches the CCB list for a CCB with the specified Local ID. If such a CCB is found, a pointer to it is returned. Otherwise the value vNoCCB is returned.

ReleaseCCB(TheCCB: CCB *)
    Removes TheCCB from the CCB list and releases the storage used by it.

3.2.4 MSG Primitive Routines

3.2.4.1 Host Table

The MSG Primitive Routines' Host Table is identical to the Local MSG's Host Table, described in Section 3.2.3.1.

3.2.4.2 Generic Names Table

The Generic Names Table for the MSG Primitive routines is similar to that for the Central and Local MSGs (described in Section 3.2.2.2), but here only the arrays GnNmCVec[], GnNmVec[] and GnCdVec[] are used. Primitive routines use the same generic table-managing module (generic.c) as the Central and Local MSGs.

3.2.4.3 Pending Events

In the MSG Primitive Routines, Pending Events (PEs) are used to keep track of outstanding MSG primitives issued by the user. They are implemented in the module pmpe.c. The structure of a PE is:

```
struct PE
{
  int peLink;        /* Link for completed PE queue */
  int peID;          /* PE ident */
                     /*-(incarnation #,,PE index) */
  int peType;        /* Type code of PE */
  int peState;       /* State code of PE */
  int peSignal;      /* Signal code */
  int *peDispositionAddr;
                     /* Address to return disposition */
  int peRcvBufSize;
                     /* Size of return buffer */
                     /*-peAlarmCode overlays */
  int *peSizeRcvdAddr;
                     /* Addr to return size of rcvd msg */
                     /*-peAlarmCodeAddr overlays */
  char *peRcvBuffer;
                     /* Address of receive buffer */
  int peSrcLen;      /* Size of SourceName guy */
  char *peSourceName;
                     /* Address to return source name */
  int *peSpecialHandlingAddr;
                     /* Addr to return special handling */
  int peTimeout;     /* Timeout of this PE */
  struct CCB *peCCB;
                     /* Associated CCB (if any) */
  struct PE *peTimeoutQLLink;
                     /* Left link of timeout queue */
  struct PE *peTimeoutQRLink;
                     /* Right link of timeout queue */
};
```

Operations on PEs are:

InitPEs()
Initializes the PE vector.

ReleasePE(OldPE: PE *)
    Returns OldPE to the free state, where it may be reallocated by AllocatePE().

FindPE(ID: int) returns PE *
    Looks in the PE vector for a PE with the given ID. If found, a pointer to the PE is returned. Otherwise, vNoPE is returned.

PendingPEsExist() returns int
    If one or more pendings PEs exist, true (-1) is returned, else false (0).

GenerateNewPE(PETypeCode: int) returns PE *
    Allocates a new PE and fills in the Type field with PETypeCode. Initializes the Signal and Disposition Address fields to null, and returns a pointer to the new PE. If no PE can be allocated, the value vNoPE is returned.

SignalPE(ThePE: PE *, Disposition: int)
    Delivers the disposition value Disposition to the address specified in the peDispositionAddr field of ThePE, then depending on the signal value, either releases the PE (null signal), adds it to the completed PE queue (request signal), or simply returns (unblock signal).

AddPEToCompletedQueue(ThePE: PE *)
    Internal routine called by SignalPE. This routine adds ThePE to the completed PE queue, where it will be removed by ReqSig. The completed PE queue is pointed to by the global variable CompletedPEQueue. Queues are described in Section 3.2.1.2.

GetCompletedPE()
    Called by ReqSig, this routine returns the first PE on the completed PE queue, or vNoPF if the queue is empty.

AllocatePE() returns PE *
    Allocates a free Pending Event from the PE vector, returning a pointer to it. If there are no free PEs in the PE vector, vNoPE is returned. This is an internal routine, called by GenerateNewPE.

3.2.4.4 Connection Control Blocks

These keep track of information needed by the MSG primitive routines to maintain direct connections. Only connections which were initiated by the user process have CCBs maintained by the MSG primitive routines. The Local MSG creates a CCB and holds any requests to open a connection received from another process until the user process requests that the connection be opened. CCBs are kept on a list identical to the Local MSG's CCB list (see Figure 9).

The structure of a CCB is:

```
struct CCB
{
  struct CCB *ccbNextCCB; /* Link to next CCB in chain */
  struct CCB *ccbPrevCCB; /* Link to previous CCB */
  int ccbConnID;          /* User connection ID */
  int ccbState;           /* Connection state */
  int ccbLocalID;         /* My ID for connection */
  long *ccbRmSockAddr;    /* Where to deliver remote */
                          /*-socket */
  struct PE *ccbPE;       /* PE for open/close */
  struct InternalMSGName ccbFrnProcName;
};
```

Operations on CCBs are:

InitCCBList()
Initializes the CCB list. As for the Local MSG, all CCBs are kept on this list through their entire life.

GenerateNewCCB(<u>ThePE</u>: PE *, <u>ConnID</u>: int, <u>DestName</u>: InternalMSGName *) returns CCB *
Allocates a new CCB from the free storage pool, fills in the ConnID, PE and FrnProcName fields from the arguments, fills in the LocalID field by incrementing the global CCBID and using the new value, adds the CCB to the CCB list and returns a pointer to it.

GetOldCCB(<u>ConnID</u>: int, <u>DestName</u>: InternalMSGName *) returns CCB *
Searches the CCB list for a connection with the specified connection ID and remote process name. If found, a pointer to it is returned, else the value <u>vNoCCB</u> is returned.

ReleaseCCB(<u>TheCCB</u>: CCB *)
Removes <u>TheCCB</u> from the CCB list and returns the storage used by it to the free storage pool.

AddtoCCBList(<u>TheCCB</u>: CCB *)
    Adds <u>TheCCB</u> to the end of the CCB list. This is an internal
    routine, called by GenerateNewCCB.

4. PROGRAM COMPILING, LOADING, AND MAINTENANCE PROCEDURES

4.1 Support Software Requirements

The source programs for MSG are a collection of program modules, written in the programming language C. MSG requires a non-standard compiler due to the use of long (> 8 character) identifier names. This compiler, called npjcc, has a special preprocessor which handles identifiers up to 24 characters long, but is otherwise identical to the ncc compiler. Those identi iers which are not unique in their first 7 characters are defined (in the .names declaration files) to unique 7-character names.

To generate a new version of MSG, the modules must be compiled and loaded with npjcc. Shell files are provided for this purpose, and these are listed in Section 4.2.

The modules which comprise the MSG source programs are kept on-line in the UNIX file system. Each module is stored in a separate UNIX file. The constituent modules are:

cmsg.names	Name redeclarations for Central MSG modules.
lmsg.names	Name redeclarations for Local MSG modules.
lmutil.names	Name redeclarations for lmutil.c
msg.names	Name redeclarations for the data structures defined in msg.h.
pmsg.names	Name redeclaration for MSG Primitive routines.
util.names	Name redeclarations for util.c, llist.c and queue.c.
ccb.h	Defines structure and states of Connection Control Blocks.
cmsg.h	Defines externals used by all Central MSG modules.
errcodes.h	Defines error codes which are sent in the "Reason" field of MSG-to-MSG messages.
generic.h	Definitions for generic name routines.
hcb.h	Defines the Host Control Block (HCB) structure, and its states.
llmsg.h	Defines op codes and lengths of Local MSG-to-Local MSG messages.
lmsg.h	Defines externals used by all Local MSG modules.

lmutil.h	Defines structure of (UNIX) port header.
lpmsg.h	Defines op codes and lengths of Local MSG-to-user process and user process-to-Local MSG messages.
mcb.h	Defines the Message Control Block (MCB) structure.
mmmsg.h	Defines op codes, header lengths, and structure of MSG-to-MSG messages.
msg.h	Defines full and internal MSG name structures, UNIX signals, and other constants used in MSG.
msgpe.h	Defines Pending Event (PE) structure and states.
msgs.h	Defines structure of user process-to-Local MSG, Local MSG-to-user process, and Local MSG-to-Local MSG messages.
netopen.h	Defines structure of open parameter block for opening network connections.
pcb.h	Defines Process Control Block (PCB) structure.
prmcodes.h	Defines (MSG) signal types, connection types, and other special parameter codes.
retcodes.h	Defines codes which are returned by an MSG primitive call, or in the disposition field.
streamio.h	Defines structure of data streams implemented in streamio.c.
util.h	Defines linked list and queue structures (LE and QE).
clmsg.c	Routines that handle the sending of messages to Local MSGs from the Central MSG.
cmsg.c	Central MSG Initialization and main loop.
cmsgi.c	Central MSG routines that handle incoming messages from the network.
cmsgnt.c	Central MSG routines that open network connections to other MSGs.
cmsgo.c	Central MSG routines that handle outgoing traffic from Local MSGs.
getconfig.c	Routine to read the configuration file.
generic.c	Routines to handle generic codes and names.
hosts.c	Host table initialization.
incnum.c	Incarnation number initialization for all MSG components.
inipmsg.c	Initialization routine for MSG Primitive routines.
llist.c	Routines that implement linked lists.
lmsg.c	Local MSG initialization and main loop.
lmsgi.c	Local MSG routines that handle incoming messages from other Local MSGs.

lmsgo.c	Local MSG routines that handle outgoing messages from the user process.
lmutil.c	Utility routines used by Local MSG and Central MSG, but not the MSG Primitive routines.
pmlmint.c	Routines that handle pipe I/O between the user process and the Local MSG.
pmpe.c	Implements Pending Events for MSG primitive routines.
pmusr.c	Contains the user-callable MSG primitive functions.
ptmo.c	Handles Pending Event timeouts for MSG primitive routines.
queue.c	Routines that implement queues.
streamio.c	Stream I/O routines used by getconfig.
string.c	String functions.
util.c	Utility routines used by all MSG components.

4.2 Procedures

Shell files are used to compile and load UNIX MSG. These shell files assume that the C-language source files are in a subdirectory named src, and that the object files and executable binaries are to be placed in a subdirectory of the same (immediate) parent directory named obj. The files are listed below:

To create pmsg.o (MSG primitive routines):

```
sh comp inipmsg pmsg
sh comp pmusr pmsg
sh comp pmpe pmsg
sh comp pmlmint pmsg
sh comp ptmo pmsg
sh recomp ccb pmsg
sh recomp generic pmsg
sh recomp incnum pmsg
sh recomp hosts pmsg
sh recomp util pmsg
sh ncomp getconfig
sh ncomp queue
sh ncomp streamio
sh ncomp string
cd ../obj
echo "In loading phase"
ld -r inipmsg.o pmusr.o pmpe.o pmlmint.o ccb.o ptmo.o\
      generic.o incnum.o hosts.o util.o queue.o\
      getconfig.o streamio.o string.o
mv a.out pmsg.o
cd ../src
```

To create the local MSG:

```
sh comp lmsg lmsg
sh comp lmsgi lmsg
sh comp lmsgo lmsg
sh recomp ccb lmsg
sh recomp lmutil lmsg
sh recomp hosts lmsg
sh recomp generic lmsg
sh recomp incnum lmsg
sh recomp util lmsg
sh ncomp getconfig
sh ncomp llist
sh ncomp queue
sh ncomp streamio
sh ncomp string
cd ../obj
echo "[Loading LMSG"]
npjcc -n lmsg.o lmsgi.o lmsgo.o lmutil.o ccb.o\
         hosts.o generic.o incnum.o util.o\
         queue.o llist.o getconfig.o streamio.o\
         string.o -ln
cd ../src
```

To load the Central MSG:

```
sh comp cmsg cmsg
sh comp cmsgi cmsg
sh comp clmsg cmsg
sh comp cmsgo cmsg
sh comp cmsgnt cmsg
sh recomp hosts cmsg
sh recomp lmutil cmsg
sh recomp util cmsg
sh recomp generic cmsg
sh recomp incnum cmsg
sh ncomp getconfig
sh ncomp llist
sh ncomp queue
sh ncomp streamio
sh ncomp string
cd ../obj
echo "[Loading CMSG]"
npjcc -i cmsgo.o cmsgi.o clmsg.o cmsgo.o cmsgnt.o\
         lmutil.o util.o llist.o queue.o hosts.o\
         generic.o incnum.o getconfig.o\
         streamio.o string.o -ln
cd ../src
```

The shell files comp, recomp and ncomp are listed below:

comp:

```
if ! -newer $1.c -than ../obj/$1.o exit
echo $1.c
npjcc -c -O -D $2 $1.c
mv $1.o ../obj
```

recomp:

```
if -newer $1.c -than ../obj/$1.o echo $1.c
npjcc -c -O -D $2 $1.c
mv $1.o ../obj
```

ncomp:

```
if ! -newer $1.c -than ../obj/$1.o exit
echo $1.c
ncc -c -O $1.c
mv $1.o ../obj
```

4.3 Verification

At present there is no formal verification procedure for UNIX MSG. However, newly generated versions of MSG are tested prior to release with an extended version of the M1 process (see Appendix D of the MSG System/Subsystem Specification and the M1 user manual). Although this process does not exhaustively test all MSG functions, it does exercise the principal MSG operations and provides some boundary-error testing, and so represents a reasonably effective regression test for new versions of MSG.

4.4 Special Maintenance Programs

As noted in Section 4.3, the processes M1 and M2 are used in the verification procedure for MSG. The executable M1 and M2 files are kept on line as _m1_ and _m2_, respectively, and the source program for these processes are kept on line as _m1.c_ and _m2.c_.

4.5 Other Special Maintenance Procedures

None.

4.6 Error Conditions

Refer to the MSG User Manual for the UNIX implementation.

4.7 Listings

Listings for the MSG source programs are supplied as an appendix to this document under separate cover.