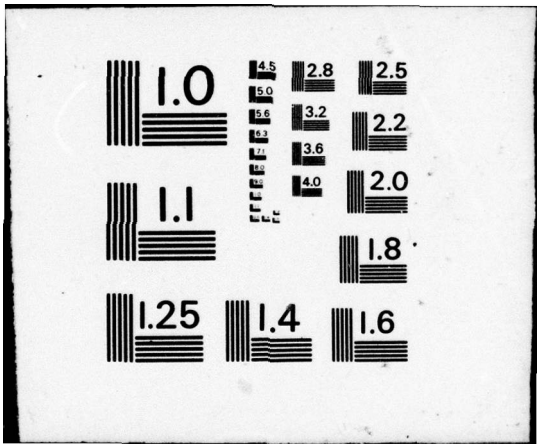
AD-A034 133 MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD F/G 9/2
NATIONAL SOFTWARE WORKS.(U)
SEP 76 R MILLSTEIN F30602-76-C-0094
UNCLASSIFIED CADD-7603-0411 RADC-TR-76-276-VOL-1 NL

1 OF 3
ADA034133


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

(12)

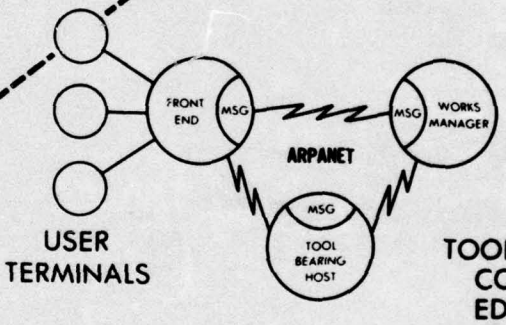
FG.

RADC-TR-76-276, VOLUME I

SEPTEMBER 1976

NATIONAL SOFTWARE WORKS

STATUS REPORT NO. 1



TOOLS:
COMPILERS
EDITORS
DEBUGGERS

CO-SPONSORED BY

ADVANCED RESEARCH PROJECTS AGENCY
INFORMATION PROCESSING TECHNIQUES OFFICE
1400 WILSON BOULEVARD
ARLINGTON, VIRGINIA

U.S. AIR FORCE
AIR FORCE SYSTEMS COMMAND
ROME AIR DEVELOPMENT CENTER
GRIFFISS AIR FORCE BASE, N.Y.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD) under ARPA Order No. 3061.

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

This report has been reviewed and is approved for publication.

APPROVED: Richard A Robinson

RICHARD A. ROBINSON
Project Engineer

APPROVED: Robert D. Krutz

ROBERT D. KRUTZ, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

NATIONAL SOFTWARE WORKS, STATUS REPORT NO. 1

Contractor: Massachusetts Computer Associates
Contract Number: F30602-76-C-0094
Effective Date of Contract: 1 July 1975
Contract Expiration Date: 30 June 1977
Short Title of Work: National Software Works, Status Report No. 1
Program Code Number: 6P10
Period of Work Covered: Jul 75 - Feb 76

Principal Investigator: Robert Millstein
Phone: 617 245-9540
Project Engineer: Richard A. Robinson
Phone: 315 330-7746

Approved for public release,
distribution unlimited.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and by Rome Air Development Center. It was monitored by Rome Air Development Center under Contract F30602-76-C-0094 and by the Office of Naval Research under Contract N0014-75-C-0073.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-76-276, Volume 1	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): NATIONAL SOFTWARE WORKS, STATUS REPORT, NO. 1	5. TYPE OF REPORT & PERIOD COVERED: Interim Report, 1 Jul 75 — 29 Feb 76
	6. PERFORMING ORG. REPORT NUMBER: CADD-7603-0411
7. AUTHOR(s): Multiple; Robert Millstein	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0094 NEW; N00014-75-C-0073
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Massachusetts Computer Associates, 26 Princess Street, Wakefield MA 01880	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702E, 63728F; 55540830
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington VA 22209	12. REPORT DATE: Sep 76
	13. NUMBER OF PAGES: 211
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Rome Air Development Center (ISCP), Griffiss AFB NY 13441	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:
Richard A. Robinson (ISCP)
Related NSW Documentation - User's Guide, Manager's Guide, Tool User's Guide, Tool Installation Guide, etc. (see reverse)

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

Software Systems
Software Engineering
Computer Networks

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

The National Software Works (NSW) is a facility, resident on the ARPANET, intended to support the construction, use, maintenance, modification, verification, and storage of programs and bodies of information on which these programs operate. It is principally aimed at the construction of programs and at providing software tools which can be used in the construction activity.

NSW is intended to facilitate both the administrative and technical aspects of these activities. Thus, it provides mechanisms for the exercise of fiscal and

OVER

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

093 745

access control in the operation of a programming project, and also access and storage conveiences to programmers for the management of their files.

The salient factor in the conception of NSW is the expectation that the hardware, software and human resources needed for the execution of a task may be geographically and administratively dispersed, although connected through the network. Tools whose use is to be coordinated may be resident at different computer installations, possible under the control of different organizations, each with its own rules of operation.

Block 18. (continued)
This work was supported by the Advanced Research Projects Agency of the Department of Defense and by Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F30602-76-C-0094 and by the Office of Naval Research under contract number N0014-75-C-0073.

TABLE OF CONTENTS

Chapters 3 and 5 of this report were produced by Bolt Beranek and Newman and by Massachusetts Computer Associates, Inc. Chapter 3 is available separately as Bolt Beranek and Newman Report No. 3237 and Massachusetts Computer Associates Document No. CADD-7601-2611. Chapter 5 is available separately as Bolt Beranek and Newman Report No. 3266 and Massachusetts Computer Associates Document No. CADD-7604-0111.

The remaining chapters of this report were produced by Massachusetts Computer Associates. Chapter 4 is available separately as CADD-7602-2011. Chapter 6 is available separately as CADD-7601-1511. Chapter 7 is available separately as CADD-7602-0211.

Chapter 1: Introduction

1. A General Introduction

1.1 Purpose

The National Software Works (NSW) is a facility, resident on the Arpanet, intended to support the construction, use, maintenance, modification, verification, and storage of programs and bodies of information on which these programs operate. It is principally aimed at the construction of programs and at providing software tools which can be used in the construction activity.

NSW is intended to facilitate both the administrative and technical aspects of these activities. Thus, it provides mechanisms for the exercise of fiscal and access control in the operation of a programming project, and also access and storage conveniences to programmers for the management of their files.

The salient factor in the conception of NSW is the expectation that the hardware, software, and human resources needed for the execution of a task may be geographically and administratively dispersed, although connected through the network. Tools whose use is to be coordinated may be resident at different computer installations, possibly under the control of different organizations, each with its own rules of operation.

1.2 Design

NSW as an entire system contains large collections of information about its users and the resources belonging to the system; it also contains the programmatic objects whose execution constitutes the operation of the system.

The software animating the NSW is called NSWExec; it is partitioned into independent processes on different processors in the network. These processes have individual names, such as Works Manager (WM), Front End (FE), Foreman, File Package (FP), Works Manager Operator (WMO), etc.

NSWExec appears, operationally, as a state-of-the-art time-sharing monitor. That is, it functions as a keeper and supplier of computational resources, and as a mediator between the user and these resources.

The essential functions of a time-sharing monitor which NSWExec is to provide in the larger network-wide environment are:

Logging in and out -- permitting the user to make, and break, contact with the monitor, and authenticating his right to use its services.

Maintaining a file system, with access protection and provisions for shared use.

Handling I/O with the user's terminal.

Interpreting and honoring the user's requests for resource usage ("Executive Commands").

Setting a specified program into operation ("running a tool") at the user's behest, and linking his terminal to the program in case the tool runs interactively.

Since the users, the resources, and the NSWExec software may all be dispersed throughout the network, creating analogues of these basic functions of a time-sharing monitor in this environment has raised some complex design problems. This overview will be organized as a consideration of these functions in the network environment, with a description of the design solutions adopted in the NSW. The functions we shall discuss are:

Maintaining a physically dispersed but conceptually integrated file system, with adequate access controls.

Managing communication between the separate components of the system.

Catering to the user at his on-line terminal -- connecting him to NSWExec, accepting and interpreting his raw input, protecting him from intervention by non-NSW programs (local host executives, etc.).

Responding to the user's requests for resource use and disposition -- i.e., the normal "Executive Commands", permitting operations on files, inspection of current information about resources and circumstances, invocations of tools, etc

Initiating execution of a tool, and providing a File-System interface so that the tool may obtain the input files it needs from the NSW File System, and deliver the output files it produces into the NSW File System, in a manner compatible with the file-system conventions of the host system where the tool is resident.

2. FILE SYSTEM

2.1 Design Considerations

As does any contemporary Operating System, NSW provides a file system to its users, with naming conventions, protection, access controls, and facilities for entering, deleting, copying, and renaming individual files. However, it had been determined at the beginning of the design that NSW would not "own" any on-line storage device, dedicated to the storage of NSW files.

A principal element of the NSW concept is to both facilitate and constrain file access and file sharing by the members of a programming project, in a manner which will allow the implementation of a wide variety of management policies. To this end, NSW has its own file-naming conventions and mechanisms for verifying access rights which are rather different from those of the hosts' operating systems.

In any case, the user must not be required to have any knowledge of the individual file systems on the hosts; rather, he must be able to use a uniform file-system vocabulary in any reference to his files, regardless of what component of NSW he is communicating with.

2.2 Design Solutions

The files in the NSW File System actually live in the various file systems of Arpanet hosts: on any host which can provide storage for NSW files, NSW "owns" one or more directories (accounts), with the maximum protection available, in which it keeps its files. Hosts providing file storage are called "Storage Hosts". There is also a "principal NSW host" in the Arpanet. This is the host on which the central elements of the NSWExec software are executed -- in the current implementation, a TENEX.

NSWExec contains an information retrieval system, resident on the principal NSW host in the Arpanet. The data base of this information retrieval system does not contain the NSW files themselves, but rather it constitutes the catalogue of the File System. Every file name known to the NSW File System has a record in this catalogue; part of the information in the entry for a particular file gives the location (identity of the host, plus file identification within that host's system) of any existing copies of the file itself. Note that the existence of multiple copies of a file is information which is not normally available to users.

Some of the operations which the user might wish to perform on files can be done merely by making changes in the catalogue, such as deleting a file, renaming a file, or removing a semaphore (access lock) which he had had set on a file; the user, of course, does not directly access the catalogue. But others require operations on the bodies of the files themselves, such as making a copy of a file within the NSW file system, importing a file from outside the NSW system, or exporting a file to a destination outside NSW.

For these operations of making physical copies of files, NSWExec calls upon a "black box" called the File Package, whose job is to understand file transmission across the Arpanet so thoroughly that it can accomodate any likely formats and perform any reasonable conversions necessary to cause a copy of a file at one place in the network to appear at another place. The specifications of the File Package are described in Chapter 4.

The portion of NSWExec software which includes the file-catalogue information retrieval system is called the Works Manager (WM). It has a number of other functions, which we shall discuss at the appropriate places in this Overview. The host on which the WM runs -- called above "the principal NSW host" -- is termed the "WM host", in distinction to other hosts participating in NSW, such as a "Front End host", or a "Tool-bearing host".

The two NSWExec components mentioned in this section -- the Works Manager (including especially the file catalogue system) and the File Package -- are clearly new programs which have had to be written for NSW.

3. COMMUNICATION SYSTEM

3.1 Goals

For acceptable operation the network connection between the user's terminal and whatever system he is communicating with should be a fast, character-by-character, full-duplex link. But such links are expensive in Arpanet, in terms of traffic loads and response time, so they should be direct, and used only when a person is at one end of the connection.

Nonetheless, there will need to be frequent communication between the dispersed software components of NSW, and this communication should be as efficient as possible.

Normally, a program on a time-sharing system can be executed only by a logged-in user of that system. Within NSW, the user should only have to log in to NSWExec itself, and not to any of the individual Tool-Bearing Hosts (TBHs).

3.2 Decisions

The individual components of NSW software will be configured as independent, coordinate, concurrent processes (even if they happen to reside on the same host).

The standard communication between two NSW processes will be by unitary messages, expressed in an 8 bit message format, and dispatched through a message-handler (named MSG), which is itself an independent process on each host participating in NSW.

A process, then, does not call another as a subordinate, or subroutine. Rather, it sends a message requesting a service, and later receives a message in response.

Two NSW processes may establish a direct network connection, if desirable for terminal response or for transmitting large volumes of data. But the initial communication between them, and the agreement to set up the direct connection, are accomplished via MSG.

In the Arpanet implementation, the several MSGs are privileged processes on their hosts, with exclusive rights to reserved network sockets. This permits bypassing the local host's login procedure when executing a process on that host.

MSG is a new program, written for NSW. The specifications of MSG are described in Chapter 3.

4. USER-TERMINAL CATERING

4.1 Situation

The user must be able to get in touch with NSWExec in a reasonably straightforward fashion, preferably without having to log in to any other systems along the way.

As discussed above, the direct connection between the user and NSWExec should be a fast, character-by-character full-duplex link (for most terminals), so that the user will receive rapid and convenient response to his typeins. It should include

Character echoing, perhaps with substitution for the input character, and suppression of echoing for passwords, and

Basic editing facilities, such as the ability to backspace (delete) a character or a word, retype what has been typed for inspection before confirming, or kill what has been typed so that the user can start over.

Any process the user may be connected to is actually running on some Arpanet host, under the host's own operating system. Operating systems generally have some reserved character which, when received from the terminal, causes them to interrupt communication between the terminal and the running process in order to allow the user to communicate directly with the operating system. The NSW user must be protected from the consequences of mistakenly typing such a character.

Conversely, if the NSW user is in communication with some process other than NSWExec, there must be some action he can take to temporarily suspend his communication with that process in order to communicate with NSWExec. Such actions (e.g., typing a special character) must be intercepted before they are transmitted to the connected process.

But the user will, in general, be only indirectly connected to the WM host, so that, if these terminal-catering functions were to be performed at the WM host, the response would be unacceptably slow.

Aside from these user-catering functions, the principal content of the communication between the user and NSWExec will be the the user's calling for the execution of executive commands (see below), and WMExec's displaying the response to these commands on the user's terminal. The amount of information necessary to specify such commands is not large, and the display of NSWExec's responses does not require two-way communication.

4.2 Decision

The user-catering function of NSWExec will be placed in a separate process, called the Front End (FE), to which the user will always be connected, and which will run on a machine as "close" to the user as possible.

In the initial system, the FE process resides on the principal NSW host; a user can achieve connection to this FE by either:

Logging in to this host in the normal fashion, CONNECTing to the appropriate directory, and running the program NSW; (this mode of operation will always be present as a fall-back option)

Executing, from his local host or TIP, an Initial Connection Protocol to a reserved socket on the NSW host.

In an early stage of development, the FE will also run on a dedicated minicomputer, connected to the Arpanet either as a local host (through an IMP) or directly as a "smart TIP". The user will be have a broad-band connection to this minicomputer.

However the user's connection to the FE process is achieved, he will immediately be permitted (in fact, required) to LOGIN to NSWExec, identifying himself and giving a password. When the LOGIN is accepted by NSWExec, he will be able to issue any Executive Commands.

The FE process provides echoing on the user terminal, recognition and completion of abbreviated command words (if the user desires), editing functions on the user's typein, and more sophisticated display-control functions.

In the standard mode of operation, when the user is communicating with NSWExec or with "integrated" tools, the user's interactions with the terminal are driven by a "grammar" contained in the FE process, which elicits from the user the information needed to specify the operation he wishes performed. This information is then packed into an message and expedited through MSG to the appropriate recipient (Works Manager or tool).

An "integrated" tool is one which in fact handles its communications with the user in the above fashion -- via MSG.

A tool which has not been "integrated" is called an "old" tool. It will be put in contact with the user through a direct TELNET connection between the FE process and the tool process. Thus another feature of the FE is that it can establish and maintain this TELNET connection.

In the standard mode of operation, no danger exists that the user might send to the tool some special character which would place him unwittingly in contact with the tool-host's Operating System. In the TELNET-connection mode, either the FE or the Foreman (see below) must filter out any such characters.

In either mode of operation, a reserved special character will be recognized by the FE process, having the effect of temporarily suspending communication with the tool, and returning communication to NSWExec.

For the duration of his session with NSW, all communication between the user and the system will thus be mediated by the FE, whether the user is conversing with the Works Manager or with one or more tools. This provides a consistent style of interaction with all elements of NSW, except perhaps from some "old" tools which will have their own conventions which the user must obey.

The FE Process is new software, designed and programmed by Charles Irby at SRI/ARC, who also designed and implemented the language for specifying grammars, the compiler for that language, and the interpreter for the compiled grammars.

5. EXECUTIVE COMMANDS

5.1 The component of NSWExec which implements the user's Executive Commands is called the Works Manager (WM). It resides on the principal NSW host, which is therefore called the WM Host.

The formats of the Executive Commands are specified in the Executive Grammar, which is always available to the FE Process. When the user has specified a command to his, and the FE's, satisfaction, it is packaged into a message which amounts to a call on some procedure within the WM; this message is then sent from the FE to the WM via MSG.

5.2 Executive Commands are essentially requests for the use of computing resources (including requests to inspect the status of resources). Hence the WM is fundamentally a purveyor and allocator of resources.

The WM maintains in its data base lists of the rights, privileges, and responsibilities of the users known to the system. When the user logs in, his right to use NSW is authenticated by checking this information. Whenever he requests the use of any resource, his right to use it is verified against these lists.

As mentioned above, the WM maintains in its data base the catalogue of the file system, and uses this to control the existence of copies of files on different hosts in the network. The catalogue, together with the user's rights information, allows the WM to control access to the files in the system.

The WM also maintains information on the tools available within NSW, which enables it to cause a tool process to be created and run on its appropriate host. This information, together with the user's rights information, allows the WM to control access to the tools.

5.3 As part of its resource-managing responsibilities, the WM provides to the (human) managers of programming projects within NSW facilities for admitting new users to the system, and specifying the rights the new user shall enjoy.

These management facilities will in fact be embodied in a separate Management Tool Kit, access to which will be restricted by the rights of the user seeking to execute it, as with any other tool. The Management Tools are described in Chapter 7.

5.4 The types of Works Manager commands are discussed in Chapter 4., The problems involved in making the WM more reliable, larger scale, and more efficient are descussed in Chapter 6.

6. TOOL INTERFACE

6.1 Design Considerations

In a contemporary interactive computer system, a tool runs under the control of the Operating System on its computer; the Operating System provides to the tool:

Means for communicating with the user's terminal;

Means for using the file system;

Other miscellaneous services more directly associated with the hardware, such as memory allocations, interrupt servicing, date-and-time and elapsed-time information, etc.

In the course of its operation, the tool will interact with the file system in several different ways:

It will need to open for reading (or modification) some already-existing files (input files).

It will need to create scratch files for temporary storage of information during its operation (and perhaps for re-start after a crash).

It may produce new files (or modifications of input files) which are to be delivered to the file system after the tool-run is completed (output files).

In NSW, where the tool, the user, and the "Operating System" (NSWExec) are, in principle, all on different network hosts, several considerations apply:

Communication between the tool and the user is handled through MSG (perhaps plus a TELNET connection), as discussed above; hence tool/user communication must be diverted from the host OS's terminal-handling mechanisms to some other process.

The tool's input files must come from the NSW File System, and not from the host's own file system, hence requests for input files must be diverted from the host OS to the WM. However, once the tool has obtained the file from the NSW File System, it must be able to work within the local file system for operations within the file -- reading or writing at particular locations within the file.

But to use the NSW file system for the storage of scratch files would be grossly inefficient, requiring frequent WM calls, and frequent updating of the WM's File System catalogue; hence, the tool should continue to use the host file system for these.

To keep the tool operation integrated within NSW, the tool's output files must be submitted to the NSW File System for storage. Hence, calls to the host OS to close, or deliver, output files must be intercepted and re-directed to the WM.

The last category of local OS services -- the miscellaneous ones -- must clearly be left intact, since it would be either impossible or expensive to provide them from the WM.

Since it is intended to be an easy task to adapt an existing program to run as an NSW tool, it is obviously desirable to minimize the amount of programming required to do this.

For instance, a tool should not have to include the software necessary to send and receive MSG messages.

6.2 Design

A new program, called the Foreman must be written to run on each host which will provide tools.

The Foreman has at its disposal a number of empty file directories (accounts, workspaces) within the local file system, which it will provide to tools running on that host.

When the WM has decided to run a tool on a particular host, it sends a message to the Foreman on that host, asking it to load and start up an instance of the tool process. The SF will select one of its local directories and assign it for the tool to run "out of" (or "in").

When the tool wants to Open an input file, it has presumably gotten the NSW Filename of the file from the user. It passes this name to the Foreman, requesting the file. The Foreman then sends a message to the WM, requesting that a copy of the file be sent to the local directory assigned to the tool. When the copy has arrived in that directory, the Foreman returns to the tool the local directory name of the copy, which it will have "opened" for the tool in the local file system.

And similarly, when the tool wants to Close an output file, it passes the local directory name of the file, together with the NSW Filename the user has given, to the Foreman. The Foreman, in turn, sends a message to the WM, "Delivering" the file to the NSW File System -- that is, requesting the WM to make a copy of the file in one of the NSW's own directories on some host (perhaps the same host, if it is also a NSW Storage Host).

When the tool is ready to stop running, it notifies the Foreman so that control of the communication link to the user is not returned to the local OS. The specifications of the Foreman are described in Chapter 5.

Old tools which are to be made to run under NSW will need to be modified only to the following extent:

Points of communication between the tool and the user terminal must be detected and modifiedin one of two ways:

If it seems feasible to structure the communication in message blocks, these messages should be composed for transmission via MSG.

If, however, it is necessary to maintain TELNET-style single-character communication with the user, "system calls" for implementing this must be replaced by accesses to the TELNET connection to the FE.

Each file used by the tool must be identified as an input file, an output file, or a scratch file.

All places where the tool Opens an input file -- i.e., requests by name a (presumably) pre-existing file from the local file system -- must be identified and replaced by calls on the SF.

All places where the tool Closes an output file --i.e., delivers a file name and contents for storage in the local file system -- must be identified and replaced by calls on the SF.

The tool must notify the Foreman before terminating its execution.

Chapter 2: Works Manager Procedures

1. Introduction

The Works Manager (WM) is the central software of NSWExec. Its job is to authenticate users' interactions with NSW, to carry out executive commands, and to control access to all NSW resources.

Operationally, the Works Manager is a "server process", which is brought to life when a Works Manager call is made by a Front End (FE) process or a Foreman process: there exists no single Works Manager process which remains continuously alive while dealing with multiple petitioners, as is the case, for example, with the MSG and FE processes.

From outside, the Works Manager appears as a collection of separately-callable procedures, each performing a specific function. Coordination of the separate procedures and synchronization of separate incarnations of the Works Manager process are effected by jointly-accessed, interlock-protected, data structures. Each Works Manager call, either from a Front End or a Foreman, is a call on a specific one of these procedures.

The principal shared data structure is the Catalogue in the NSW Information Retrieval System, which contains all the long-lived data about all elements of NSW.

Furthermore, whenever NSW is in operation, there are tables of current data, residing in the Works Manager Host, which depict the momentary state of NSW -- e.g., a list of users currently logged in, a list of the tools currently running, etc. Almost every Works Manager call will result in some change being made to one or more of these tables. It is these "hot" tables which give the appearance of continuity of service by "the" Works Manager on behalf of a user.

The purpose of this chapter is to list these Works Manager procedures and give a brief description of their effects.

Some of these procedures can meaningfully be called only from a Front End process (e.g., LOGIN, LOGOUT), and others only from a Foreman process (e.g., OPEN, DELIVER); the remainder may be called from either source. It will be indicated for each procedure, explicitly, from where it may be called. Procedures marked with an asterisk (*) may only be called from the Front End. Procedures marked with a plus sign (+) may only be called by a tool. Unmarked procedures may be called either by a tool or from the Front End (except ENDTOOL which is called by the Foreman).

2. WORKS MANAGER PROCEDURES

2.1 Connection

* LOGIN (project, node-name, password)
--> userid, node-profile, user-profile, system-message, login-message, qhave-mail

LOGIN connects a user to the NSW, establishing him as an active user with all the rights implied by the node at which he has logged in. Mistakes (i.e., non-recognition by the WM) in arguments will be handled by HELP returns. The user will then be permitted to retype the incorrect argument or abort and re-start the login.

project: STRING, node-name: STRING, password: STRING

This triple is collected from the user for the initial LOGIN call; it identifies and gives access to a node on the NSW project tree. The user is then considered to be logged-in "at" (or even "as") that node. All rights to access files, use tools, use WM procedures, and spend money are associated with the login node.

userid:

Internal WM identifier of a logged-in user. It is assigned to the user at login by the WM, and its thereafter regularly used in all messages between the FE and the WM, so that each can be sure which user the message refers to.

node-profile: BIT-STRING, user-profile: BIT-STRING

These are encoded instructions to the FE (and perhaps also to the WM), determining the style of communicating with the user; they include specifications for lengths of heralds and prompts to be displayed, degree of command-word recognition and completion desired, lengths of lists to be displayed, etc. The information in node-profile is peculiar to the node, while the information in user-profile applies to the person "owning" the node in NSW records, regardless of which of his nodes he may log in at (a person may own several nodes -- for example, a project manager will own the top node in his project, but might also set up some subsidiary nodes for his personal work).

system-message: STRING

This would be an important operational message from NSW (or perhaps Project) management, to be displayed to all NSW users at their next login ("system news").

login-message: STRING

This would be an operational message from the WM or Works Manager Operation (WMO) reporting on the status of previously submitted batch jobs, the status of the files used by tools which crashed in a previous session but which have subsequently recovered, etc. It informs the user of changes to the user environment which have occurred since his last logout.

qhave-mail: BOOLEAN

This, if TRUE, will cause the FE to inform the user that there exists new mail addressed either to him personally, or to his present login node; to read his mail, the user should call a Readmail tool.

* LOGOUT (userid, qfast)
--> cost

LOGOUT disconnects a user from NSW. Normally, if any interactive tools are still running for this user, he will be asked to terminate those tooluses in the way appropriate for each tool, and re-call LOGOUT. Alternatively, the user may ask the NSW to terminate these tooluses for him (qfast set to TRUE); in this case, output files which the tool has already DELIVERed will be in the NSW File system, and any other files will be lost. If the TBH has gone down while the user was running, the WM will try later to recover and save the local workspace in which the tool was running. At a subsequent LOGIN the user will be told (via login-message) about the saved workspace, and he will be given an opportunity to DELIVER the files to the NSW File System. Batch Tools are, of course, asynchronous with respect to user-NSW connection, and are not affected by logout.

cost: INTEGER

cost is returned by several WM procedures. It is to be interpreted as the cost in cents of the use of a tool or of an entire session, as appropriate. The user is given the opportunity to gripe about the cost by returning a non-null message when invited to protest.

A WM-procedure RELOG was originally planned, which would enable the user to move his login location from one node to another. This has been replaced by a Front End command MOVELOG which executes successive calls on the WM-procedures LOGOUT and LOGIN.

* REATTACHTONSW (project, nodename, password)
  --> userid, node-profile, user-profile, LIST [(tool-info)]

This procedure is intended to allow a user to resume his session in the event that his FE-machine goes down, by re-contacting NSWExec through another FE process. This is not a high priority procedure and will not be available in early versions of the WM. The LIST of "tool-info" in the returned items would contain, for each tooluse the user had initiated, the information necessary for the new FE to set up its tables as if it had been the one the user had been using, presumably: tool-process-ID, tool-name, tooluse-name, (perhaps) tool-grammar.

2.2 Tools

* RUNTOOL (userid, tool-name, tooluse-name)
  --> tool-process-id, tool-grammar

RUNTOOL verifies that the user has access to the tool called tool-name. It creates an instance of the tool process, and establishes a communication path. It returns to the FE a process identification for the FE to use in calling the tool, along with the tool grammar. The tooluse-name argument is provided so that several active instances of the same tool can be distinguished.

tool-name: STRING

The name, e.g., NLS, TECO by which a tool is known to the WM. Retrievable under this tool-name in the WM Catalogue is a large block of data called the Interactive Tool Descriptor. This descriptor supplies whatever information the WM needs for successfully running the tool and servicing its file requests. More specifically, it lists the ARPAnet hosts (called Tool-Bearing Hosts (TBHs)) and process identifications of potential instances of the tool so the WM can cause an instance of the tool process to be readied for execution and it lists the file-attributes required for input files and those to be attached to output files (see OPEN and DELIVER).

tooluse-name: STRING

The name by which a particular instance of a given user's active tool is known. This argument is necessary to distinguish between, e.g., different concurrent uses of NLS.

tool-process-id: MSG<process name>

(see MSG: The Interprocess Communications Facility for the National Software Works, Massachusetts Computer Associates, Inc., CADD-7601-2611, Bolt, Beranek & Newman, Inc., Report No. 3237.)

The tool-process-id is a name which the FE and WM can use for communication with the tool.

tool-grammar: ?

The tool-grammar is an encodement of the Command Meta Language (CML) specification of the commands provided to the user for interacting with the tool. When the FE process is on the WM Host TENEX, what is passed is the local name of the .REL file which embodies this grammar. When the FE process is on a separate machine, the grammar itself will be passed, in some format yet to be specified (perhaps BIT-STRING?).

* ENDTOOL (userid, tooluse-name)
  --> cost, LIST [NSW-filenames of files with semaphore left set by tool]

ENDTOOL is called from the Foreman when a tool indicates it has finished running; this procedure causes the WM to detach the tool from the FE and terminate the tool process. The "return items" shown above are actually sent to the FE (rather than returned to the Foreman), along with a message instructing the FE to remove this tooluse from its list of active tools and to break its communication link with the tool. All semaphores set during the tool's running are unset unless the Tool Descriptor indicates that this tool is one which understands use of the semaphore. If so, a list of files with semaphore set is sent to the FE so that the user can either confirm for each file that he wants to leave the semaphore set, or indicate that he wants it unset.

* RERUNTOOL (userid, tooluse-name)
  --> tool-process-id, tool-grammar

RERUNTOOL reestablishes the connection between a user and a tool which was running on a TBH which had crashed and has subsequently come back up. This procedure is not defined yet and will not be available in early versions of NSWExec.

2.3 Files, No Movement

DELETE (id, filespec, qhelp)
  --> NSW-filename

DELETE verifies that filespec designates a unique file to which the user (identified explicitly by userid, or implicitly if DELETE is called by a tool (first argument 0)) has DELETE access. This access is blocked by a set semaphore. If any assistance is required it is obtained via a HELP return (if qhelp is T or if DELETE were called by a batch tool) or by a direct FE HELP call (otherwise). Once a unique file has been found, its catalogue entry is marked. It will no longer be accessible to OPEN, COPY, RENAME, EXPORT, etc., but the actual file catalogue entry and file copies are not immediately deleted. The NSW-filename of the deleted file is returned. This return could be a HELP return, requiring confirmation before the actual delete occurs. Alternatively, since the file does not immediately disappear, an UNDELETE operation could be supported.

id: userid | 0

WM procedures which can be called from either the FE or a tool require "id" as their first argument. If, in an actual call, the first argument is non-zero, then it is a userid, and the call is from a Front End. If it is zero then the call is from a tool. WM procedures which show "userid" as first argument can only be called from the FE. If any other first argument is shown (except for the procedures LOGIN and REATTACHTONSW which are only FE-callable), then the WM procedure can only be called by a tool.

NSW-filename: STRING

The NSW-filename is the full identification of the file in the NSW File System, which could amount to a rather long string of text. However, the user will never have to type in a full filename; instead, he will use either a "filespec" or an "entry-name", depending on the intended use of the file (see these terms below).

A (full) NSW-filename consists of two parts: the name-part, and the attribute-part, separated by a slash (/). The name-part is a sequence of name-components, separated by periods (.); the order of the name-parts is significant. The attribute-part is a list of attributes, separated by semi-colons (;); the order of the attributes is not significant.

An example of a full NSW-filename might be:

```
IVTRAN.PHASE1.PARSE.SYMBOL-HASH/
     UT:BCPL-SRC;CR:ILLIAC+BOLDUC;DTC:1975:08:25:16:03:38
```

Name-parts do not necessarily designate unique files. NSW files have attributes and certain of these attributes (those supplied by tools - syntactically indicated by UT:) may be used for disambiguation. Thus it is entirely possible for a user to have a file with name-part A.B and attribute UT:FORTRAN-SRC and another file A.B with attribute UT:360-FORTRAN-REL. The NSW-file-names of these two files are unambiguous and consist of name-part/tool-supplied attributes. E.g., A.B/UT:FORTRAN-SOURCE and A.B/UT:360-FORTRAN-REL. The tool supplied attributes consist of those file attributes which are supplied by tools through WARRANT, DELIVER.

filespec: STRING

A filespec is an abbreviated form of an NSW-filename, used in contexts where the name of an existing file is required -- i.e., COPY and DELETE accesses. A filespec need contain only enough parts of the NSW-filename to unambiguously denote the file. As explained below under "scope", an initial segment of the name-part can be automatically supplied, and need not be typed by the user. Any sequence of consecutive name-components which are not necessary for identifying the particular file may be replaced by three periods (...). Also, an attribute-part may be typed in a filespec, to distinguish between two files which differ only in attributes (e.g., the source-language and the relocatable binary forms of the same program).

Thus, the file named by the example under "NSW-filename" above, would be retrieved under the filespec

IVTRAN...PARSE/UT:BCPL-SRC .

Any time a filespec is used, if it does not happen to designate a unique file, the WM will send to the Front End for display to the user an indexed list of the full file-names of all files which match the filespec; the user may indicate which one he intends by responding with the index number.

More specifically: If the filespec matches a great many files, the WM Information Retrieval System will protest and refuse to retrieve them; the user will be asked to submit a more reasonable filespec. If the filespec matches few enough files to retrieve, but more than some user-settable limit ("maxlist"), the user will be informed of the number of files matched, and asked if he wants to see the list of names. Only if the number retrieved is less than maxlist will the list be displayed automatically. In any case, the user has the option, in response to any of these messages, of entering a different filespec.

qhelp: BOOLEAN

qhelp is used when a tool calls the WM and does not want the WM to directly contact the user at the FE for assistance. In this case qhelp is set to FALSE.

RENAME (id, filespec, entry-name, qhelp)
--> old-NSW-filename, new-NSW-filename

RENAME verifies that filespec designates a unique file to which the user has DELETE access. This access is blocked by a set semaphore. If any assistance is required it is obtained via HELP return or direct FE call as above. RENAME forms a new NSW-filename using entry-name and the tool-supplied attributes of the old file. It verifies ENTER access and unambiguity. As usual assistance is sought should there by any difficulty. The NSW catalogue is then altered to reflect the new name-part and both old and new NSW-filenames are returned.

entry-name: STRING

An entry-name is an abbreviated form of an NSW-filename used in contexts where a new filename is to be created. As described below under "scope", the contents of the user's ENTER scope will be prefixed to the entry-name as typed. Aside from this scope abbreviation, however, the user must type the entire name-component of the filename -- that is, no ellipses (...) are permitted. No attribute-part is permitted, either, since the user may not assign attributes to files (his identity as creator of the file, and the date-and-time of creation attributes will be appended automatically).

Referring again to the NSW-filename example above, if we assume the user had an ENTER scope of IVTRAN.PHASE1, the filename shown could have been created (minus the UT:BCPL-SRC attribute, which could only have been appended by a tool), using the entry-name

PARSE.SYMBOL-HASH .

\+ SETSEMAPHORE (filespec, qhelp)
--> NSW-filename

The WM verifies that the tool can use SETSEMAPHORE, that filespec designates a unique file to which the user has DELETE access, and that the semaphore is not already set. Assistance is obtained via HELP return or direct FE call as above. If all is well, the semaphore is set and the NSW-filename is returned.

The semaphore is set by a tool on behalf of a user who is writing into the file in order to warn other potential users that the file may be undergoing change. The semaphore is either 0 - meaning not set - or it is project + node-name indicating the setter of the semaphore.

UNSETSEMAPHORE (id, filespec, qhelp)
--> NSW-filename

The WM verifies that filespec designates a unique file to which the user has DELETE access. Assistance is obtained as usual. If all is well, the semaphore is unset and the NSW-filename returned.

READSEMAPHORE (id, filespec, qhelp)
--> NSW-filename, project-node

The WM verifies that filespec designates a unique file. Assistance is obtained as usual. If all is well, the STRING project + node-name is returned if the semaphore is set. If the semaphore is not set, the empty STRING is returned.

\+ WARRANT (attcode, NSW-filename)
--> new-NSW-filename

WARRANT adds the attributes referenced in the Tool Descriptor by attcode to the file whose current name is NSW-filename. Since tool-supplied attributes are part of NSW-filenames, the new NSW-filename is returned. WARRANT is not currently implemented.

attcode: INTEGER

An index into the Tool Descriptor, where a large list of required or known attributes can be referenced without a large amount of net transmission.

* DISPLAY (userid, access-type, filespec)
--> LIST [NSW-filenames]

DISPLAY lists file catalogue entries for the set of files which match filespec.

access-type: COPY | DELETE | ENTER

Denotes a particular kind of access to the NSW file system.

2.4 Files, Movement

COPY (id, filespc, entry-name, qhelp)
--> src-NSW-filename, dst-NSW-filename

COPY verifies appropriate accesses: COPY access for the source file, ENTER access for the destination file, plus DELETE access for the destination file if the copying would "overwrite" an existing file. It creates a new NSW catalogue entry and a new copy of the source file.

EXPORT (id, filespec, external-name, password, qhelp)
--> src-NSW-filename

EXPORT verifies COPY access and sends a copy of the source file to the location designated by external-name.

external-name: STRING

Either an ARPAnet pathname with password or a device pathname with password. An external-name is needed for copying files from a source outside of NSW (see IMPORT, TRANSPORT) or copying to a destination outside of NSW (see EXPORT, TRANSPORT). An external-name argument is always accompanied by a password argument (which is a STRING) for gaining access to the external directory, device, etc.

IMPORT (id, external-name, password, entry-name, qhelp)
--> dst-NSW-filename

IMPORT is the inverse of EXPORT.

TRANSPORT (id, src-external-name, password, dst-external-name, password, qhelp)

TRANSPORT is an extended FTP for NSW users. It is not currently implemented.

\+ OPEN (input-attcode, filespec, qset, qhelp)
--> NSW-filename, local-filename, new-filespec

OPEN is used by a tool to obtain a copy of an NSW file. The WM verifies that there is a unique file designated by filespec to which the user has COPY access and which has the attributes implied by input-attcode. Assistance is obtained as usual. Should the user also have DELETE access to the file, then the WM will set the semaphore on the file if either the Tool-Descriptor indicates that it should be set, or if qset is TRUE. If the semaphore is already set (and the user has DELETE access rights), then this tool's access to this file is blocked unless the user, in response to a message to the FE, indicates that he is willing to use a copy of the filed version, even though someone else may be planning to replace it soon. In any event, if the semaphore is set, the user is informed that it has been set, and by whom. The WM makes a copy of the file into the workspace used by the tool, performing whatever conversions are necessary and possible. The NSW-filename of the copied file and the local filename of the new copy are returned. If in the course of disambiguating filespec, the user supplies a new filespec, then that is returned also.

\+ DELIVER (output-attcode, local-filename, entry-name, qhelp)
--> NSW-filename

DELIVER is used by a tool to insert a file into the NSW file system. ENTER access and unambiguity are verified with assistance sought as usual. An entry is made in the NSW file catalogue and an NSW-owned copy is made of the file designated by local-filename. The attributes implied by output-attcode are appended to the file. The original file is left in the tool's workspace. The NSW-filename of the new entry is returned.

+ READDEVICE (local-filename, device-code, qhelp)

READDEVICE is used by tools to input via local tape, card reader, paper tape reader without making NSW files. The WM could figure out from the userid which FE the user was at and therefore what the external-name of the appropriate device is. Alternatively, the node-profile (user profile?) could contain the association betweeen device-code and actual external-name for the device. After that, this procedure is just like IMPORT.

device-code: STRING

crd = card reader
pun = card punch
ptr = paper tape reader
ptp = paper tape punch
mt7 = 7 track mag tape
mt9 = 9 track mag tape
dta = DEC tape

+ WRITEDEVICE (local-filename, device-code, qhelp)

WRITEDEVICE is the inverse of READDEVICE. Neither READDEVICE nor WRITEDEVICE are currently implemented.

2.5 Project Management

These project management procedures (and the display procedures in section 2.6) are only temporary. They will be superseded by the Management tools currently being designed and implemented. In fact, they should not be construed as even giving a flavor of the project management facilities which NSW will eventually have; they are too primitive to even do that.

* ADDNODE (userid, son-node-name, son-password)

ADDNODE creates a new node in the project tree. This new node is a son of the node at which the user logged in. Its name is the STRING son-node-name and its password is the STRING son-password.

* DELETENODE (userid, son-node-name, qtree)

DELETENODE checks to see that son-node-name designates a son of the node at which the user logged in. If so, that node (the son) is deleted. If qtree is TRUE then the entire subtree (if any) headed by son-node is also deleted. Otherwise, the sons of son-node (grandsons of login node) are made sons of login node.

* ADDRIGHT (userid, son-node-name, rights)

ADDRIGHT checks to see that son-node-name designates a son of the login node. The rights are checked to verify that the login node possesses them. If so, the son node is given the additional rights. Otherwise, an error is signalled, since a node cannot give rights that it does not have itself.

rights: procedure-rights|tool-rights|file-rights
procedure-rights: PROCEDURE, proc-names
tool rights: TOOL, tool-names
file rights: (COPY|DELETE|ENTER), keys
proc-names, tool-names, keys: STRING,...

A right authorizes access to some NSW resource. Before a WM procedure consumes a resource on behalf of some user, that user's login node is checked to validate access to that resource. Thus, a user's access to the system is defined by the rights stored at a node.

Procedure-rights is the STRING PROCEDURE followed by the names of WM procedures which the user (and tools on his behalf) are allowed to invoke. The reserved string ALL designates access to every WM procedure.

Tool-rights is the STRING TOOL followed by the names of tools which the user is able to run. Again, ALL denotes every tool.

File-rights is one of the STRINGs COPY, DELETE, ENTER followed by keys which define the parts of the file system to which the user has access. In general a user will have file-rights of each access type. Again, ALL is used to denote any file.

key: STRING

A key is, syntactically, an initial segment of the name part of an NSW-filename followed, optionally, by a slash(/) and one or more attributes separated by semicolon (;). See the description of NSW-filename above. A user has access to a file if he has a key which matches the NSW-filename of the file. A key matches an NSW-filename if (1) the name-part of the key is an initial segment of the name-part of the filename, and (2) the attributes of the key are attributes of the filename.

* DELETERIGHT (userid, son-node-name, rights)

The WM verifies that son-node-name designates a son of the login node. If so, rights are removed from the son-node.

* CHANGERIGHT (userid, son-node-name, delete-rights, add-rights)

  CHANGERIGHT does both ADDRIGHT and DELETERIGHT.

* ADDSCOPE (userid, scopes)

  ADDSCOPE adds scopes to the information stored at the login node.

  scopes: (COPY|DELETE|ENTER), scope,...

  scope: STRING

    Scopes are a method of abbreviating NSW-filenames for user convenience. There are three types of scopes -- COPY, DELETE, and ENTER -- corresponding to the three access-types. A user may have any number of COPY and DELETE scopes active, but only one ENTER scope. Whenever a filespec is typed by the user, the filespec, together with all of either his COPY or DELETE scopes (depending on the context of use) are submitted to the Information Retrieval System. When an entry-name is typed by the user, his ENTER scope is prefixed to the entry-name typed, in order to construct the full name-part of the file to be entered into the Catalogue.

    Syntactically, a scope looks like a key: a sequence of name-components separated by periods (.), follwed, optionally, by a slash (/) and one or more attributes separated by semicolon (;).

    Scopes are set and changed by the user to make it convenient to reference files in a relatively small region of filename-space, presumably because he expects to do most of his work with those files. If he wishes to access a file outside of his current scope, he may prefix a filespec or entry-name with a dollar-sign ($) (to be read as a barred "S", meaning: DON'T SCOPE), to override the automatic scoping mechanism.

    A scope cannot be set unless it is implied by a key which the user has. A key implies a scope if (1) the types are the same and (2) the name-part of the key is an initial segment of the scope and (3) the attributes of the key are attributes of the scope.

* DELETESCOPE (userid, scopes)

* CHANGESCOPE (userid, delete-scopes, add-scopes)

  These are analagous to DELETERIGHT and CHANGERIGHT.

* CHANGEPASSWORD (userid, new-password)

This changes the password of the login node to new-password.

2.6 Display

The results of these procedures are given by direct display at the FE rather than by returning values from the function calls.

* DISPLAYNODE (userid, node-name, qtree)

* DISPLAYRIGHT (userid, node-name)

* DISPLAYSCOPE (userid, access-types)

These procedures are similar in that they all show part or all of the information stored at a node. They differ only in details.

DISPLAYNODE shows all of the node designated by node-name. If qtree is TRUE then the entire subtree headed by node-name is displayed. Otherwise only node-name is shown.

DISPLAYRIGHT shows all of the rights possessed by the node designated by node-name.

DISPLAYSCOPE shows all of the scopes of type access-types where access-types is a list containing one or more of COPY, DELETE, ENTER. See also DISPLAY in section 2.3..

* SHOWJOB (userid, job-number)

When a user submits a batch job it is assigned a job-number (INTEGER) by NSW. The user can subsequently check the status of the job with SHOWJOB.

2.7 WMO and IBS Support

There are currently eight procedures for WMO and three for IBS (Interactive Batch Specifier). These procedures cannot be used from the FE or by a tool. They are only of internal NSW use; hence they will not be described in this section. For completeness, they are listed with their arguments and results (names and types).

```
GETDESCRIPTOR
                processor       INTEGER (IBS)
                tool-name       STRING
=>              qavail          BOOLEAN
                jcl             STRING
```

```
//
VERIFY
                file-spec       STRING (IBS)
                userid          INTEGER
                qset            BOOLEAN
=>              qthere          BOOLEAN
                item-index      INTEGER
                number-recs     INTEGER
                max-length      INTEGER
                qctl            INTEGER
                qsem            INTEGER
                NSW-file-name   STRING
                mail            INTEGER
//
ENTER
                qref            BOOLEAN (IBS)
                file-name       STRING
                attrbs          STRING
=>              qdisp           BOOLEAN
                item-index      INTEGER
                NSW-file-name   STRING

SENDBATCH
                job-number      INTEGER (WMO)
                tbh-name        STRING
                wsd-name        STRING
                device-name     STRING
                file-name       STRING
                argument-vector INTEGER-VECTOR
=>              error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
                local-file-name STRING | NULL-STRING | EMPTY
//
DELIVERBATCH
                job-number      INTEGER (WMO)
                tbh-name        STRING
                wsd-name        STRING
                device-name     STRING
                argument-vector INTEGER-VECTOR
                local-file-name STRING | NULL-STRING | EMPTY
=>              error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
//
RESERVEBATCH
                job-number      INTEGER (WMO)
                tbh-name        STRING
                wsd-name        STRING
                device-name     STRING
                file-name       STRING
                argument-vector INTEGER-VECTOR
=>              error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
                local-file-name STRING | NULL-STRING | EMPTY
```

```
//
DELETEBATCH
                job-number      INTEGER (WMO)
                tbh-name        STRING
                wsd-name        STRING
                device-name     STRING
                file-name       STRING | EMPTY
                local-name      STRING | EMPTY
=>              error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
//
FINDTBH
                job-number      INTEGER (WMO)
                processor       INTEGER
                space           INTEGER
                time            INTEGER
                tool-name       STRING
                device-type     STRING
=>              error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
                tbh-name        STRING
                wsd-name        STRING
                devie-name      STRING
//
EXECUTEJOB
                job-number      INTEGER (WMO)
                tbh-name        STRING
                wsd-name        STRING
                device-name     STRING
                argument-vector INTEGER-VECTOR
                jcl             STRING | NULL-STRING | EMPTY
=>              error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
                jcl-file-name   STRING | NULL-STRING | EMPTY
                job-name        STRING | NULL-STRING | EMPTY

JOBALLDONE
                job-number      INTEGER (WMO)
                tbh-name        STRING
                wsd-name        STRING
                device-name     STRING
                time            INTEGER
                charges         INTEGER
                id              INTEGER
= >             error-code      INTEGER
                error-number    INTEGER
                error-message   STRING | NULL-STRING | EMPTY
```

```
//
JOBINQ

                    job-number          INTEGER (WMO)
                    tbh-name            STRING
                    wsd-name            STRING
=>                  error-code          INTEGER
                    error-number        INTEGER
                    error-message       STRING | NULL-STRING | EMPTY
                    status              INTEGER
                    time                INTEGER
                    charges             INTEGER
                    report              STRING
                    device-name         STRING
```

Chapter 3: MSG Design Specifications

1. Introduction

1.1 Overview

The National Software Works (NSW) provides software implementers with a suitable environment for the development of programs. This environment consists of many software development tools (such as editors, compilers, and debuggers), running on a variety of computer systems, but accessible through a single access-granting, resource-allocating monitor with a single, uniform file system. By its very nature, the NSW consists of processes distributed over a number of computers connected by a communications network. These processes must communicate with one another in order to create a unified system. This paper describes the communication facility (named MSG) which was developed to provide interprocess communication for the implementation of the NSW. The communications network is currently the ARPANET. However, we have designed the MSG facility to be as independent as possible of the ARPANET implementation so that the concepts may be carried over to implementations on other networks.

We begin by describing the more important of the processes which comprise NSW and discussing the pattern of communication which those processes require. We then proceed to abstract from those patterns a model of interprocess communication which is sufficient for NSW. Finally, we develop the details of the MSG facility itself.

It is our hope that both the description of the process of defining MSG as well as the description of the structure of the protocol will be of interest to protocol developers for the ARPANET and other networks.

1.2 NSW Components

The monitor of NSW is the Works Manager. It is responsible for servicing requests for system resources - e.g., running a tool, opening a file. The Works Manager verifies that each such request is valid (using in this verification a rather elaborate access data base which serves as a domain for automated project management machinery). The Works Manager then allocates to each valid request the necessary resource. This allocation generally involves either the creation of a tool (e.g., editor, compiler) instance - i.e., the creation of a new NSW process - or the movement of a file (which movement may be either inter- or intra-host).

For each user of NSW an interface to the other components is provided by a Front End, which may be local to the user. In the sequel we will talk as if the Front End were local, so that communication to the user is synonymous with communication to the Front End. This is not, however, an NSW system requirement. The Front End filters the user's input stream, discarding bad characters (e.g., control-C should not be sent to TENEX tools) and interpreting system-wide control characters - delete line, retype line, escape to the Works Manager, etc. In addition, the Front End may provide local parsing of the Works Manager command language and, conceivably, even tool command languages.

Just as users see the NSW environment through the Front End, so also do tools see an extended local system environment through a Foreman component. Tools are software systems which are written for a given host - e.g., MULTICS. To become NSW tools they must be inserted into a slightly different milieu. This different milieu is provided by a Foreman component on the tool's host. The Foreman provides the tool with access to NSW resources, such as NSW files. Thus a tool gets NSW resources by making a local call on the Foreman, which then forwards the request to the appropriate NSW component. From the viewpoint of other NSW components, then, it is the Foreman rather than the tool with which most communication must occur.

The final component of interest here is the File Package. There is an instance of the File Package on each tool-bearing host. These File Packages are responsible both for local file system manipulation - e.g., delete, local file copy - as well as inter-host file transfers and reformatting.

4.3 Patterns of communication

We will now describe the anticipated patterns of communication between the NSW processes. These communications factor into six types:

- Front End - Works Manager
- tool/Foreman - Works Manager
- Works Manager - File Package
- Front End - tool/Foreman
- tool/Foreman - tool/Foreman
- File Package - File Package

The other possible pairs - e.g., Front End - File Package, File Package - tool/Foreman - do not represent communication paths in NSW.

. Front End - Works Manager

Communication between these two kinds of process consists of user requests for NSW resources (Front End to Works Manager) and Works Manager responses to such requests (Works Manager to Front End). Examples of such requests are: run a tool, copy a file, delete a file, etc. These requests are relatively infrequent - a user may make only a few per hour. Each request is short - almost all requests can easily be encoded in 1000 bits. The response to each request is also short - again, less than 1000 bits. The time required to process a request is generally brief - certainly on the order of milliseconds as compared to the minutes between requests. There is no necessity for a request to be processed by the same instance of the Works Manager that processed any previous request (since all instances of the Works Manager share the same common data base). Hence a communication link need not be retained between a Front End and a Works Manager between resource requests. Thus we can characterize Front End - Works Manager communication as a sequence of unrelated elements, where each element is a short request, a brief delay, a short response, and a long delay until the next element of the sequence.

. tool/Foreman - Works Manager

These communications are exactly analogous to Front End - Works Manager communications. A tool (on behalf of a user) requests an NSW resource of the Works Manager. Examples of such requests are: open a file, create a subsidiary tool process, deliver a file, etc. As above, these requests are generally less than 1000 bits, are processed by the Works Manager in

milliseconds, have responses of less than 1000 bits, and are relatively infrequent. The only difference between this pattern and the preceding pattern is that tool requests are more frequent than Front End requests, although the time between such requests is still measurable in minutes.

. Works Manager - File Package

These communications are again analogous to the above. Indeed, these requests (of the Works Manager to the File Package) occur in order to service a Front End or tool request of the Works Manager. For example, when a tool asks the Works Manager to open a file, the Works Manager must then ask a File Package process to make a copy of that file, possibly across the ARPANET. The time to make a cross-net copy of a file may be measured in seconds (even in minutes for large files), but such long copies are expected to be infrequent. Thus, the same pattern of a short request (not related to previous requests), a brief delay, a short response, a long delay holds for Works Manager - File Package communication also.

. Front End - tool/Foreman

Communication between these processes consists of user commands to tools and tool responses to users. In some cases these communications will fit into the same pattern as the three previous cases. Often, however, the pattern is different. Consecutive requests are related and must be serviced by the same tool. The time between the user's command and the tool's response may be greater than the time between the response to the previous command and the issuing of the next command. Also, the frequency of user commands to tools may be much greater than the frequency of either user or tool requests to the Works Manager. In addition, the length of a Front End - tool/Foreman communication may be large. For example, in a typical session a user might request the use of a text editor (Front End - Works Manager communication), get a particular file to edit (tool/Foreman - Works Manager communication), and then insert two hundred lines of program text into that file. Thus Front End - tool/Foreman communication is expected to vary from the infrequent, short request pattern to frequent, long transmissions of information.

. tool/Foreman - tool/Foreman

These communications are relatively infrequent. No tool currently installed in NSW needs to talk directly to another tool. Nevertheless, debugging tools for NSW as well as multi-process tools have been proposed and are being implemented.

Such tools require communication facilities. We expect that their patterns of communication will be analogous to Front End - tool/Foreman communications.

. File Package - File Package

Some very small fraction of these communications will consist of short, infrequent messages - e.g., a source File Package telling a destination File Package the length and encodement of a file - but the bulk of such communication will consist of files being transferred. Thus, we can characterize this pattern as infrequent transmissions of many bits.

3.4 Model of Communication

From these expected patterns of communication we can abstract a model of the kind of interprocess protocol that NSW requires. We have, roughly speaking, three patterns of communication:

- Infrequent short transactions between previously unrelated processes (Pattern 1):
  Front End - Works Manager
  tool/Foreman - Works Manager
  Works Manager - File Package

- More frequent, longer transactions between related processes (Pattern 2):
  Front End - tool/Foreman
  tool/Foreman - tool/Foreman

- Infrequent, very long transactions (Pattern 3):
  File Package - File Package.

1.5 Modes of Communication

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

- generic addressing;
- specific addressing;

and three different modes of communication:

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

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. Pattern 1 communication is always initiated by the transmission of a generically addressed message.

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.

Message exchange is provided by MSG to support the requirements of pattern 1 communication and some pattern 2 communication. 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.

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 pattern 3 communication, such as file transfers between hosts, and some pattern 2 communication, 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 to a process 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.

All modes of interprocess communication supported by MSG follow the same basic pattern, which is roughly as follows:

1. One process tells MSG about a message or alarm to be sent or a connection to be opened. It also specifies a destination address and a signal by which MSG can

inform it that the message or alarm has been sent or the connection opened.

2. Another process which matches the above destination address tells MSG that it is ready to receive the same type of communication. It also specifies a signal by which MSG can inform this process that the message or alarm has been received or the connection opened.

3. MSG sends the alarm or message or opens the connection. It also signals the source process that the message or alarm has been sent or the connection opened and signals the destination process that the message or alarm has been delivered or the connection opened. After it receives the signal, the process receiving a message or alarm always knows the specific address of the sender.

1.6 Sequencing of Messages

Normally MSG does not guarantee that messages sent from one process to another process will be delivered to the destination process in the order in which they were sent. However, since it is expected that NSW processes may frequently desire message sequencing, it is possible for a process to ask MSG to sequence certain messages.

To achieve sequencing a process can specify when it sends a message that the message is to be sequenced. MSG will guarantee that a sequenced message from process A to process B will be delivered to process B only after all previous sequenced messages from process A have been delivered to process B. A process may, if it chooses, intermix sequenced and unsequenced messages.

Several of the situations which motivate the presence of the alarm communication mode within MSG also require that a process receiving messages be able to distinguish messages sent before an alarm was sent (or received) from those messages sent afterwards. That is, it is often important for a pair of processes to synchronize a message stream with respect to an alarm.

To facilitate such message-stream/alarm synchronization, MSG supports the concept of message stream markers. A stream marker is an attribute of a message. When sending a message a process may specify whether or not the message is to carry a stream marker. MSG guarantees that a message M, sent from process A to process B, which carries a stream marker will be delivered to process B only after all messages sent by A prior to M have been delivered to B and before any messages sent after M by A. Furthermore, MSG will notify the receiving process B whenever it delivers a message that carries a stream marker. The notification will be part of the information normally supplied by MSG to the receiving process.

When it is necessary to achieve message stream synchronization after an alarm, a pair of processes can use the MSG stream marker. This can be accomplished by placing a stream marker on the first message sent after the alarm (was sent or received). Although stream marked messages are provided by MSG to simplify message-stream/alarm synchronization by MSG processes, it is important to note that MSG itself places no constraints upon how processes use stream marked messages.

1.7 Host Incarnations

The NSW is expected to provide continuous, 24 hour a day, 7 day a week service. However, the various computer systems which support NSW processes may not provide such continuous service. Proper NSW operation requires that MSG be able to determine whether a name for a process refers to a process that MSG is currently managing or to an obsolete one which MSG managed during a previous period of MSG service by the host computer system in question. (The term "incarnation" is used synonymously with "period of host MSG service" in the remainder of this document.) To enable MSG to distinguish current from obsolete processes, an MSG process name (more precisely, a specific address) includes an indication of the host incarnation under which the process exists (or existed).

1.8 Organization of this Document

The remainder of this document specifies MSG in detail. There are four parts to the specification:

i. MSG process environment.
Section 2 defines in detail the environment MSG provides to MSG processes. In particular, it defines the set of primitives that MSG provides to such processes.

ii. MSG-to-MSG protocol.
NSW is a multi-computer system. Parts of MSG will reside on the various computer systems that comprise the NSW. The inter-computer protocol used by the components of MSG in order to support the MSG primitives is specified in Section 3.

iii. MSG-to-MSG Protocol for the ARPANET.
The initial implementation of the NSW will make use of the ARPANET as an inter-computer communication medium. Section 4 specifies how the ARPANET host/host communication facilities are to be used to support the MSG-to-MSG protocol.

iv. MSG-to-MSG Transmission Formats for the ARPANET.
Section 5 defines the formats to be used for the transmission of MSG-to-MSG protocol messages between ARPANET hosts.

2. MSG process environment

This section defines in detail the environment MSG provides to processes. This section covers those aspects of the MSG process environment which are common to all hosts; it is not a process-implementer's guide to MSG on any particular host. Such a guide must also discuss aspects of the process environment which are peculiar to that host.

2.4. Hosts

NSW is implemented as a number of processes running concurrently on a number of different computer systems, called hosts. MSG on each host can be thought of as an extension of that host's operating system, creating a new operating system that satisfies the MSG design. Because MSG specifies only a fraction of the host environment for a process, it is generally true that MSG processes will be sensitive to the type of host on which they run.

NSW will operate continuously, but individual hosts may not be continuously part of it. This can occur because a given host is not scheduled for continuous NSW service, or because the host has failed. We define a particular period of NSW service by a host as a host incarnation, designated by:

```
<host incarnation name> ::=
      <host designator><incarnation designator>
```

where <host designator> is an integer which uniquely designates a particular host computer and <incarnation designator> is an integer which designates this particular period of NSW service by this host.

2.2. Processes

The form of an MSG process is strongly host-dependent, since the MSG design specifies only a part of the operating system under which the process runs. An MSG process is what one generally thinks of as a process, i.e. a collection of programs, local memory, etc. to which the operating system allocates system resources such as CPU time. MSG processes must, however, have the following properties:

1. The process can make at least some MSG primitive calls.
2. The process has a unique MSG process name through which it can be addressed by other processes.

2.3. Process names

A host incarnation supports a number of MSG processes. Each process has a name of the form

<process name> ::= <host incarnation name><generic designator>
<specific designator>

The host incarnation name is the incarnation name of the host under which the process is running. The generic designator is a character string which characterizes a process in terms of its functional relationship to other processes. This characterization determines whether a process could be chosen to perform a certain function. For example, processes with generic designator WM are candidates for messages which invoke Works Manager functions. The specific designator is an integer. A process name is always unambiguous; at all times it either corresponds to a single process or is invalid.

2.4. Process addressing modes

There are two fundamental modes by which one process may address another process: generic and specific. A specific address is always a process name. Generally process A will use a specific address for process B because process A has had some prior communication with B, either directly or through some intermediary process.

A generic address, however, is of the form:

<generic address> ::= <host designator><generic designator> |
<generic designator>

Unlike specific addressing, which uniquely determines the destination process, generic addressing implies a selection by MSG of a destination process from a class of processes. This selection allocates the destination process to the communication implied by the generically addressed message. This is distinct from process allocation, in which MSG creates and terminates processes.

The class of processes from which MSG can pick a destination process for a generically addressed message is defined as follows:

1. If the generic address is of form
   <host designator><generic designator>
   then the process selected must be on the designated host. If <host designator> is not specified in the address, then the process may be on any host.
2. The <generic designator> field of the process name must match the <generic designator> field of the generic address.
3. The process must have a Receivegeneric primitive call pending.

2.5. Modes of information transfer

MSG supports three basic modes of information transfer between processes: messages, alarms, and direct connections.

A message is a string of bits created in the local memory of a sending process. MSG sends the message to a receiving process by duplicating the bit string in a specified portion of the receiving process's local memory. MSG itself imposes no further structure on messages, nor does it interpret the contents of messages. Messages are the only mode of communication which can be generically addressed.

An alarm, like a message, is a string of bits created by one process and addressed to another process. As with a message, MSG transmits the bit string to the receiver process, which has designated beforehand where the bit string is to be put. In other ways, however, alarms differ from messages. First, an alarm is a fixed-length bit string and is shorter than most messages. Second, MSG will transmit an alarm independently of any message traffic between sender and receiver processes. In fact, MSG will give alarms priority service over messages. It is anticipated that alarms will be used to transmit information about unusual or exceptional conditions, while messages and direct connections will be used to support normal communication.

A direct connection is a one- or two-way dedicated channel between two processes. MSG assists the processes in opening and closing the connection, but does not intervene in the actual use of the channel.

Messages are further differentiated by whether they are addressed to a specific process or to a generic class of processes. Processes use different primitive calls to send and receive generically-addressed messages than they use to send and receive specifically-addressed messages.

For a specifically-addressed message it is further possible to specify either (but not both) of two types of special handling: sequencing and stream marking. Normally MSG will not guarantee to deliver messages in the order in which they were sent. Sequenced messages, however, from process A to process B will be delivered to B in they same order in which they were sent by A. A stream marker message from A to B will not be delivered to B until all other messages from A to B have been delivered. Furthermore, it will be delivered to B before any other messages to B sent subsequently by A.

In all cases, MSG will inform the receiving process of any special handling given any message it receives.

2.6. MSG primitive operations

Each host supports a set of MSG primitive operations for the processes that run under it. The method of calling these primitives will be host dependent. Every primitive call produces some time later a reply (return) from MSG. We divide the set of primitive calls into two classes, differentiated by the meaning of the reply MSG makes to the primitive call. For one class of primitive call the MSG reply signifies that the primitive operation is complete. For the other class of primitive call, however, the MSG reply signifies only that the parameters of call were reasonable enough for MSG to deduce what operation to perform and that MSG has agreed to attempt to perform this operation. When this primitive operation is finally complete or has been aborted, MSG will signal the process, using a signal specified in the primitive call. We call this uncompleted primitive operation a pending event, where the event in question is the completion or aborting of the operation. A pending event has the form:

<pending event> ::= <primitive><signal><disp><timer>

where
<primitive> is the primitive operation to be performed
<signal> is a means by which MSG can signal the process that the primitive operation is complete
<disp> is a pointer to a field in the process's local memory
<timer> is a timer which tells MSG when it can abort the operation.

Every host will offer processes a set of signals for use in primitive calls that produce pending events. We shall discuss signals at greater length later in this document. The disp field, which MSG will have set before it sends the signal, tells the process whether the primitive operation completed normally or was aborted.

The set of all pending events for a process is called that process's pending event set. When the process makes a primitive call of the second class, a pending event is added to its pending event set. When MSG completes or aborts a pending event, it sets the appropriate disp field, sends the signal, and then deletes the pending event from that process's pending event set.

A process should ensure that no two elements simultaneously in its pending event set have the same signal, but MSG will not enforce this restriction. The simplest way for a process to ensure this is never to reuse a signal in a primitive call until

that signal has been received from the old call. It should be emphasized that the signal for an operation is the only reliable way for a process to ensure that this operation has completed.

2.6.1 Primitives that create pending events

Many of the following primitives contain the parameter dt. This is used to create the <timer> field of the pending event, and either specifies a time interval in local host clock units or indicates that a default value should be chosen by MSG. Unless the default is specified,
<timer> = tc+dt where tc is the local host clock time when the primitive was called.

1. Sendspecificmessage(msgarea,pnam,signal,disp,dt,sphndl)
   where
   msgarea points to a message to be sent
   pnam is a process name
   sphndl specifies special handling for the message
   0 - ordinary handling
   1 - sequenced message
   2 - stream marker message

   This causes the message pointed to by msgarea to be sent to process pnam. At the very minimum, completion of this primitive operation means that the msgarea has been read by MSG, the disp field set, and the pending event deleted from the sender's pending event set. Local hosts may opt to guarantee more, such as that when the primitive is completed the foreign host has accepted the message.

2. Sendgenericmessage(msgarea,genadr,signal,disp,dt,qwait)
   where
   msgarea points to a message to be sent
   genadr is a generic address
   qwait is a boolean

   This is like Sendspecificmessage except that here a generic address is specified instead of a process name, there is no special handling, and there is the extra parameter qwait. Unlike a Sendspecificmessage, a Sendgenericmessage may cause MSG to create a destination process. Qwait is a boolean; setting it false will cause MSG to accept the primitive only if there is a process available with a Receivegeneric primitive pending.

3. Receivespecificmessage(msgarea,srcnam,signal,disp,dt,sphndl)
where
msgarea points to a block of local memory in which MSG will put a message
srcnam points to a field of local memory which MSG will set to the process name of the sender
sphndl points to a field of local memory which MSG will set to the special handling class of the message being received:
0 - ordinary handling
1 - sequenced message
2 - stream marker message

If the primitive completes normally, i.e. if the specified signal is received and the disp field does not indicate an error, then msgarea will contain a message which was sent by a Sendspecificmessage primitive call by some process. Srcnam will contain the name of the process that sent the message, and sphndl will show if the message was sequenced or was a stream marker.

4. Receivegenericmessage(msgarea,srcnam,signal,disp,dt)
where
msgarea points to a block of local memory in which MSG will put a message
srcnam points to a field of local memory which MSG will set to the process name of the sender

This is like Receivespecificmessage except that here the message received was sent by a Sendgenericmessage primitive instead of a Sendspecificmessage primitive. There is also no special handling field.

5. Sendalarm(acode,pnam,signal,disp)
   where
    acode is an alarm code
    pnam is a process name

   This sends the alarm code acode to the process named pnam. When this primitive completes, the disp field will indicate one of the following outcomes:
   1. OK. Either the alarm was delivered to the process or it was queued and will be the next alarm to be delivered to the process.
   2. Rejected. Process pnam is not accepting alarms at all now, or another alarm is already queued for this process, or some error has occurred.

6. Enablealarm(acode,srcnam,signal,disp)
   where
    acode,srcnam point to fields of local memory

   This enables the process to receive an alarm. When the alarm is received, acode will be set to the alarm code and srcnam will be set to the name of the alarm sender. In order for an alarm to be received, not only must an Enablealarm primitive be pending but also the iaccept boolean state for this process must be true. This boolean value is changed by the primitive Acceptalarms.

7. Openconn(conntype,connid,pnam,signal,disp,dt)
where
 conntype is a connection type
  TELETYPE
  BINARY SEND-RECEIVE(s)
  BINARY SEND(s)
  BINARY RECEIVE(s)
  where s is a byte size
 connid is a connection identifier
 pnam is a process name

This opens a connection of type conntype to process pnam. The connection will be identified by connid. In order for the primitive to complete normally, process pnam must also execute an Openconn primitive addressed to this process, with the same connid and a compatible conntype. Some hosts may return a host-dependent identifier for the connection.

8. Closeconn(connid,pnam,signal,disp,dt)
where
 connid is a connection identifier
 pnam is a process name

This refers to the connection created before by the primitive Openconn(conntype,connid,pnam,...). If the connection was never opened, Closeconn will abort with an error code in the disp field. If the corresponding Openconn is still pending, the Openconn also will abort. Whatever the outcome, however, when the Closeconn primitive completes, the connection, if it ever existed at all, will be closed.

9. Terminationsignal(tsignal,disp) where
 tsignal is a signal

If this primitive ever completes, i.e. if tsignal is ever received then it should be taken as a request by MSG for the process to terminate. The disp field may be used, at host option, to specify why the termination is being requested.

2.6.2 Primitives that do not create pending events

1. Stopme()

   This primitive indicates that the process wishes to terminate. Control will never return from this primitive. The process will be terminated by MSG as soon as possible. Well-behaved processes will ensure that their pending event sets are empty before issuing this primitive.

2. Rescind(rsignal)
   where
   rsignal is a signal

   This is used to delete a pending primitive operation. The parameter rsignal must be the signal of a pending event, i.e. an uncompleted primitive operation. If the Rescind call returns successfully then the corresponding primitive will not occur and rsignal will not be sent. The Rescind may fail because the primitive operation is partially complete and it is too late to stop it, or because rsignal no longer corresponds to a pending event. The latter case generally means that the corresponding primitive has already completed. It is a host option what primitives may be rescinded at all.
   Some hosts may wish to return an event handle with rescindable primitive calls. In this case, the call will be Rescind(event handle).

3. Acceptalarms(qaccept)

   Each process has a boolean state value, iaccept. If an alarm is sent to a process whose iaccept state is false, the Sendalarm will fail with a disposition indicating that the process is not accepting alarms. If, however, iaccept is true then the Sendalarm will either match an Enablealarm, be queued, or be rejected because another alarm is already queued for this process. Acceptalarms sets iaccept to the value of qaccept.

4. Resynch(pnam)

   If MSG had been rejecting sequenced messages to process pnam due to failure of a sequenced message transmission, then MSG will now stop doing so.

2.7. Signals

Each host provides for processes running under it a set of signals. A signal is a means by which MSG can inform a process that some event has occurred, in particular that MSG has completed some primitive operation.

Different hosts will offer different signals, but all signals must satisfy certain criteria:

1. At any point in time, the process can determine whether or not the signal has been received.
2. Signals must be distinguishable, i.e. if one of several possible signals has been received, the process must be able to determine which one.
3. Signals are local. A signal to one process does not directly affect any other process.

The restrictions listed above allow hosts to specify a wide variety of signals for processes. It is not the function of this section to further specify what signals will be available on any host. We list here some examples of signals that a host might provide. These are strictly examples; they imply no MSG requirement that these particular signals be supported:

1. Block/Unblock
   The process waits and control does not return from the primitive call until the event has occurred.
2. Flag
   MSG sets a field in the process's local memory nonzero when the event has occurred. This field could be the <disposition> field itself.
3. TENEX PSI on channel n
   On TENEX, MSG sends an interrupt on PSI channel n when the event has occurred.
4. Flag plus TENEX PSI
   MSG sets a field in the process's local memory nonzero, then sends an interrupt on an agreed-upon PSI channel which is the same for all signals of this type. This differs from example 3 in that here different signals cause interrupts on the same channel. Because TENEX queues PSIs on a channel only one interrupt deep, some PSIs may be lost if MSG sends several signals of this type sufficiently close to each other in time. With care, a process can handle the resulting race without undue difficulty.

2.8 Information transmittal

The sending of messages and alarms and the opening and closing of connections all involve a pairing of compatible primitive operations in the pending event sets of (usually) different processes. Such a pairing defines an interchange of information between two processes which MSG must cause to happen. The possible pairings are:

1. Specifically-addressed message
   This pairs the primitives
   Sendspecificmessage(ma,pb,...) in process pa
   Receivespecificmessage(mb,snam,...) in process pb

   This causes the message pointed to by ma to be transmitted by MSG to process pb and put into the memory area pointed to by mb. In addition, snam in process pb will be set to pa so that the receiving process will know the name of the sending process.

2. Alarm
   This pairs the primitives
   Sendalarm(acode,pb,...) in process pa
   Enablealarm(cdval,snam,...) in process pb

   This pairing is possible only if the boolean state variable iaccept in process pb is true. This causes the alarm code acode to be transmitted from process pa to process pb and put into field cdval. In addition snam will be set to pa, the name of the sending process.

3. Generically-addressed message
   This pairs the primitives
   Sendgenericmessage(ma,genadr,...) in process pa
   Receivegenericmessage(mb,snam,...) in process pb

   This is like a specifically-addressed message pairing except that here genadr is a generic address which matches process name pb instead of being pb directly.

4. Opening a connection
   This pairs the primitives
    Openconn(ta,connida,pb,...) in process pa
    Openconn(tb,connidb,pa,...) in process pb
   where
    connida = connidb
    ta and tb are compatible connection types:
     1. ta = tb = TELETYPE
     2. ta = tb = BINARY SEND-RECEIVE(s)
     3. ta = BINARY SEND(s)
        tb = BINARY RECEIVE(s)
     where s is a byte size.

   This opens a connection of the indicated type between processes pa and pb. The connection will be hereafter identified to both processes as connida (= connidb).

5. Closing a connection
   This pairs the primitives
    Closeconn(connid,pb,...) in process pa
    Closeconn(connid,pa,...) in process pb

   This will close for both processes the connection between them which is identified by connid.

These pairings define tasks that MSG is to perform, but they allow MSG hosts a great deal of freedom in scheduling computer time and resources to the multitude of concurrent operations they must perform. We must, however, specify a few more rules:

1. Fairness. MSG will not grossly favor any one process, mode of communication, or particular operation over any other. Exceptions are:
   a. Alarms will be favored over messages.
   b. Transmission of messages with special handling attributes may be delayed until other related messages have been transmitted.

2. Access to communication. A process must always be able to have in its pending event set:
   a. One message send primitive.
   b. One message receive primitive.
   c. One alarm send primitive.
   d. One alarm enable primitive.
   e. One primitive to open or close a connection.

3. Efficiency. Within limits set by the above rules, MSG will arrange its workload so as to perform it in a reasonably efficient manner.

2.9 Sequencing of messages

As noted in Section 1.6, MSG normally does not guarantee that a collection of messages sent from one process to another process will be delivered to the destination process in the order in which they were sent. Some applications will require that the messages between two processes be sequenced. In such cases, the communicating processes could observe a private protocol to insure proper sequencing of messages. However, since it is expected that processes may frequently desire message sequencing, it is possible for a process to ask MSG to sequence certain messages.

To achieve sequencing a process can specify when it sends a message that the message is to be sequenced. MSG will guarantee that a sequenced message from process A to process B will be delivered to process B only after all previous sequenced messages from process A have been delivered to process B. A process may, if it chooses, intermix sequenced and unsequenced messages.

The sending and receiving disciplines required of MSG to support sequenced messages are discussed below. Processes should be aware that a cost is associated with the use of the message sequencing option; that cost will be reduced message throughput.

MSG cannot guarantee that every message will be delivered. (The destination host may be temporarily inaccessible, the destination process may spontaneously disappear, the message may be timed out, etc.) When MSG is unable to deliver a normal, unsequenced message, the sending process is signalled and notified (via the disposition information normally supplied by MSG) that the message could not be delivered. The sending process can then take whatever action it feels is appropriate with respect to the message in question.

Sequencing introduces an additional complexity here since a sequenced message is not independent of other messages in the sequence. To illustrate the nature of the problem, suppose that process A has attempted to send process B the sequenced messages M1, M2, M3, M4, M5. Furthermore, suppose that MSG successfully delivers M1 but is unable to deliver M2. What should MSG do with M3, M4, and M5? In particular, its inability to deliver M2 does not necessarily mean that MSG will be unable to deliver the remaining messages in the sequence. Delivery of M3, M4 and M5 without M2 may confuse process B; processes A and B are communicating via sequenced messages presumably because sequencing is important. Therefore, MSG will not attempt to deliver the remaining pending sequenced messages.

If MSG cannot deliver a sequenced message from process A to process B, it will stop the flow of sequenced messages to process B from process A until process A takes some explicit action to "resynchronize" the message sequence. MSG does this by marking process A as being out of synchrony with process B after a sequenced message from process A to process B fails. MSG will then abort all pending sequenced Sendspecificmessage primitives in process A's pending event set which are addressed to process B. Furthermore it will reject all such primitive calls subsequently made by A until A resynchronizes the message sequence with B by executing the primitive Resynch(B).

As noted in Section 1.6, in situations in which an alarm is transmitted or received, it is often important for a pair of processes to identify a point in a stream of messages between them corresponding to "where" the transmission (or receipt) of the alarm occurred. To facilitate such message/alarm synchronization, MSG supports the concept of message stream markers. A stream marker is an attribute of a message. When a process sends a message it can specify whether or not the message is to carry a stream marker. The default is no stream marker.

MSG guarantees that a message M, sent from process A to process B, which carries a stream marker will be delivered to process B only after all messages sent by A prior to M have been delivered to B (or have been determined by MSG to be undeliverable) and before any messages sent after M by A. Furthermore, MSG will notify the receiving process B whenever it delivers a message that carries a stream marker. The notification will be part of the information normally supplied by MSG to the receiving process. We emphasize that MSG itself places no constraints upon how processes use stream markers. However, we expect that standards regarding their use will be adopted for NSW.

MSG observes a queuing discipline with respect to Receivespecificmessage primitives. The Receivespecificmessage primitives executed by a process are to be satisfied in the order in which they are issued in the sense that the first Receivespecificmessage should be satisfied by the first message MSG accepts for the process, the second by the second message, etc. We note that this does not necessarily imply that the signals associated with a collection of pending receives will be delivered to the receiving process in the order in which the receives were satisfied.

In addition, we note that this MSG receiving discipline does not imply that messages from a given sending process will be delivered in the order in which the sending process sent them. If in-order delivery is required, the sending process must request "sequenced" or "stream marker" handling. When sequencing for a message is requested, the sending MSG observes a sending discipline whereby it transmits the message only after the receiving MSG has accepted all previous sequenced messages (from the sending process to the receiving process). Similarly, when stream marking for a message is requested, the sending MSG observes a sending discipline whereby it transmits the message only after the receiving MSG has accepted all previous messages from sender to receiver and additionally transmits no further messages from sender to receiver until the receiving MSG accepts this message. These sending disciplines, together with the receiving discipline described above and always followed by MSGs, is sufficient to insure in-order delivery of sequenced and stream marked messages.

2.10. Process creation and termination

To create a process MSG performs the following operations:

1. MSG assigns a process name to the process and creates an empty pending event set for it.
2. MSG creates the process on the host operating system.
3. MSG starts the process in some host-dependent agreed-upon initial state.

An MSG host may create processes for one of only two reasons:

1. In order to fulfill its obligation to find a destination for a generically addressed message.
2. As part of system initialization or restart.

To terminate a process, MSG performs the following operations:

1. MSG marks the process for termination in such a way that it will no longer be a candidate for any communication from other processes and such that it is blocked from issuing any more MSG primitives.
2. MSG completes or rescinds all elements in the process's pending event set.
3. MSG deletes the process from the host.
4. MSG forgets about the process.

2.11 Summary of terms

We present here a brief summary of the terms defined in this section:

1. Host incarnation name

   <host incarnation name> ::=
   <host designator><incarnation designator>

2. Process name

   <process name> ::=
   <host incarnation name><generic designator><specific designator>

3. Generic address

   <generic address> ::= <host designator><generic designator> |
   <generic designator>

4. Generic designator

   <generic designator> ::= character string

5. Specific designator

   <specific designator> ::= integer

6. Host designator

   <host designator> ::= integer

7. Incarnation designator

   <incarnation designator> ::= integer

3. MSG-to-MSG Protocol

This section specifies the inter-host MSG protocol which supports the primitives provided to processes managed by MSG. The concern in this section is the information communicated between MSGs rather than how it is communicated. This section assumes the existence of a bi-directional communication path between each pair of MSG host systems. Issues such as how these MSG-to-MSG paths are supported by ARPANET communication capabilities or how MSG-to-MSG messages are delivered are the subjects of Sections 4 and 5.

3.1. Transaction Identifiers.

The completion of an inter-host MSG transaction (such as the transmission of a message or an alarm) generally requires a protocol exchange that involves several inter-MSG messages. When an MSG initiates an inter-host transaction on behalf of a process it manages, it generates an identifier for the transaction which it places into the inter-MSG message which initiates the transaction. In addition, the initiating MSG generally places the name of the initiating process into the inter-MSG message.

When an MSG responds to an inter-MSG message that initiates a transaction, the responding MSG includes the transaction identifier chosen by the initiating MSG in its response. If the transaction in question is one that requires further interaction between the MSGs, the responding MSG generates a second identifier (its identifier) for the transaction and places it into the response message. All subsequent inter-MSG messages which refer to the transaction will include both transaction identifiers.

3.2. On the use of "source" and "destination".

Most inter-MSG messages are transmitted to support interactions between a pair of processes. Consequently, most of these messages include the names of two process and many include two transaction identifiers. In the specification that follows, we adopt the convention of using "source" when referring tc a process or transaction identifier managed by the initiating MSG and "destination" when referring to a process or transaction identifier managed by the responding MSG. "Source" is then relative to the initiator of the transaction; it is not relative to the sender of a particular message in the series of protocol messages needed to carry out the transaction.

3.3. MSG-to-MSG Protocol Items.

In the specifications of inter-host MSG protocol items that follow, the items are grouped according to the primitives they support. In these specifications all information exchanged between MSGs is explicitly represented as parameters of the various protocol messages. In some cases some parameters may be implicit from the protocol exchange context and are therefore redundant. Section 5 defines the transmission formats for the protocol items in detail.

4. MSG-to-MSG protocol for interprocess messages (SendSpecificMessage, ReceiveSpecificMessage, SendGenericMessage, ReceiveGenericMessage)

MESS (source-process, destination-process, source-ID, destination-ID, handling, length, message-data)

This initiates an inter-MSG message transaction. It indicates that the source-process has requested that a message (defined by length, message-data) be delivered to the destination-process. The source ID is the identifier selected by the source MSG to identify the message transaction. The destination MSG should include source-ID in all communication concerning this message transaction. The destination-ID is empty if it is unknown; it takes on meaning for interactions requiring more than a simple request and acknowledgement (see descriptions of MESS-HOLD, HOLD-OK, MESS-CANCEL and XMIT below). The destination-ID is an identifier selected by the destination MSG for the message transaction. The handling parameter specifies the special handling (if any) required by the receiving MSG in order to properly deliver the message. Examples of special handling include: include a synchronization marker with message; MESS-HOLD not an acceptable response (see below); MESS-HOLD acceptable and this MESS is an implicit HOLD-OK (see below).

Protocol requires the destination MSG to promptly acknowledge MESS with one of the following three messages.

MESS-OK (source-process, destination-process, source-ID)

This response to MESS indicates that the destination MSG takes full responsibility for buffering the message data and subsequent delivery of the data to the destination-process. This reply implies that destination-process is currently a valid name.

It does not imply that the message data has been actually received by destination-process, nor does it guarantee that destination-process will ever accept the data.

MESS-REJECT (source-process, destination-process, source-ID, reason)

This response to MESS indicates that the destination MSG will not accept the request for the transaction identified by source-ID. Reason indicates the reason for rejection. Possible reasons include: no such process, no buffer space, too many messages already queued for this process, etc. The reason supplied might be one which attempts to stimulate retransmission by the source MSG if the rejection is known to be of a temporary nature.

The following four MSG-MSG protocol items provide an important extension to the basic message transmission discipline of MESS, MESS-OK, and MESS-REJ described above. These additional protocol items are motivated by the need for flexible flow control within MSG. Their inclusion introduces complexity to the protocol. However, the flexible flow control they support is sufficiently important to justify this complexity.

MESS-HOLD (source-process, destination-process, source-ID, destination-ID)

This response to MESS indicates that the destination MSG will not accept the message data associated with the specified message transaction but that it will remember that the message transaction has been requested and at some time in the future will ask the initiating MSG to retransmit the message data. The destination-ID is the identifier selected by the destination MSG for the message transaction. Both source-ID and destination-ID should be included in any subsequent MSG-to-MSG communication concerning this message transaction.

Protocol requires that the source MSG acknowledge the MESS-HOLD promptly with one of the following two messages.

HOLD-OK (source-process, destination-process, source-ID,
destination-ID)

This reply to MESS-HOLD indicates that the source MSG agrees to buffer the message associated with the transaction specified by source-ID and destination-ID. The destination MSG will remember the pending message transaction and request transmission of the message when it is able to accept the message data.

MESS-CANCEL (source-process, destination-process, source-ID,
destination-ID, reason)

This reply to MESS-HOLD indicates that the source MSG is unwilling to buffer the specified message. In addition, it may be used by a source MSG to indicate that it has ceased buffering a message which it had previously agreed to buffer.

XMIT (source-process, destination-process, source-ID,
destination-ID)

This is used by a destination MSG to request a source MSG to transmit a message previously buffered. The XMIT signals that the message will, in all probability, be successfully accepted. On receiving a XMIT, the source MSG is expected to transmit the message identified via a MESS message (using the specified source-ID and destination ID to identify the transaction in question). All legal responses to a MESS request are appropriate for the redelivery.

A destination MSG can send a MESS-REJ rather than an XMIT in order to abort a message transaction for which the message is buffered at the source. It might choose to do this if the destination-process terminates without requesting the message.

We note that since a destination MSG can utilize the MESS-HOLD option, it may be important to provide processes managed by MSG means to declare that a MESS request be accepted or rejected immediately (i.e. not held) by a destination MSG. This concept is not currently supported at the process-MSG interface level; should it become important to do so, the "handling" parameter of the MESS item will be used to support the concept at the inter-MSG protocol level.

2. MSG-to-MSG Protocol for Interprocess Alarms (SendAlarm, EnableAlarm)

ALARM (source-process, destination-process, source-ID, alarm-code)

This initiates an inter-MSG alarm transaction. It indicates that the source-process has requested that an alarm be transmitted to the destination-process. A few bytes of data (alarm-code) are to be conveyed to the destination-process along with the alarm. The ALARM message should bypass the flow control mechanism applied to normal interprocess message transactions (MESS). Source-ID is the identifier selected by the source MSG to identify this transaction.

Protocol requires that one of the following two messages be sent promptly to acknowledge the ALARM.

ALARM-OK (source-process, destination-process, source-ID)

This response to an ALARM request indicates that the alarm request has been accepted by the destination MSG. It does not mean that the alarm has been received by the destination-process; it may be the case that the alarm is never actually delivered to the destination-process.

ALARM-REJECT (source-process, destination-process, source-ID, reason)

This response to an ALARM request indicates that the destination MSG refuses to accept the alarm. Reason indicates the reason for rejection (e.g. incorrect destination process name, process not accepting alarms, another alarm is already queued, etc).

3. MSG-to-MSG Protocol for Direct Access Communication (Openconn, Closeconn)

Because of the symmetric nature of the following three protocol messages, we change our conventions with respect to "source" and "destination". In the description of these three items, "source process" always indicates the process local to the sending MSG and "destination process" always indicates the process at the receiving MSG. The same convention is used for the transaction ID fields.

CONNECTION-OPEN (source-process, destination-process,source-ID, destination-ID, user-connection-ID, type, source-socket)

This message indicates that the source process desires to establish a direct communication path to the destination-process of the "type" specified. The source-ID is the identifier selected by the source MSG to identify the operations concerned with establishing and breaking the connection(s). Destination-ID is empty when unknown.

[For implementations which make use of the ARPANET, the source-socket specifies the socket(s) at the source MSG host which is (are) to be used in establishing the connection which implements the communication path. Protocol states that the ARPANET RFCs required to establish the connection(s) are to be exchanged immediately after both source and destination MSGs have agreed to the connection (by exchanging matching CONNECTION-OPEN messages).]

CONNECTION-CLOSE (source-process, destination-process, source-ID, destination-ID, reason)

This protocol message indicates that the sending MSG wants to close the connection identified by source-ID and destination-ID. Protocol specifies that the receiver should close the connection and acknowledge the request with a matching CONNECTION-CLOSE. CONNECTION-CLOSE may be sent to abort a connection which has not yet been completely opened. Reason indicates the reason the connection is being closed. Possible reasons include: process requested close, byte size mismatch, type mismatch, and entry timeout.

CONNECTION-REJECT (source-process, destination-process,
destination-ID, reason)

This item is used to reject a CONNECTION-OPEN or a CONNECTION-CLOSE request. It does not require an acknowledgement. Reason indicates the reason for rejection. Possible reasons include: no such destination; no such connection. The transaction identifier returned is the "source-ID" for the request being rejected.

4. MSG-to-MSG Protocol for Obtaining Process Status (Get-status primitive)

An MSG primitive to be used to obtain information regarding the status of an MSG process is to be specified in the future. The "get-status" primitive will not be required in the first MSG implementation. The following describes, in general terms, three protocol items which are intended to support the "get-status" primitive.

SEND-STATUS (source-process, destination-process, source-ID)

This protocol message requests the status of the destination-process on behalf of the source-process. Source-ID is the identifier selected by the source MSG for the status transaction.

Protocol requires that one of the following two messages be promptly sent in acknowledgement of SEND-STATUS.

STATUS-OK (source-process, destination-process, source-ID,
status-words)

This returns the status information requested by the source MSG. The information to be included in the status report has not yet been completely specified. We expect that it will include the state of destination-process including pending Sends and Receives as well as pending alarms.

[Note: It may not be desirable to allow a process to obtain detailed status information about processes with which it is not actively communicating. The precise access controls (if any) that are required for the Get-status primitive will be defined in the future.]

STATUS-REJECT (source-process, destination-process, source-ID, reason)

This response is used to indicate the rejection of a SEND-STATUS probe request. Reason indicates the reason for the rejection.

5. Miscellaneous MSG-to-MSG Messages.

The following MSG to MSG messages are provided because they have proven useful in communication system implementations and for experimental extensibility.

NOP

This message is a no-operation. It has no effect and is immediately discarded by the receiving MSG. No reply is required.

ECHO (data-byte)

This protocol message requests the receiving MSG to echo the data-byte. It can be used to see if a remote MSG is actively functioning. Protocol specifies that the data-byte of an ECHO message be promptly returned to the sending MSG in a matching ECHO-REPLY message.

ECHO-REPLY (data-byte)

Reply to ECHO.

EXPERIMENTAL (command, length, data)

This message provides for experimentation and extensibility within the MSG-to-MSG protocol. The command specifies the function requested; the length specifies the number of bytes in the EXPERIMENTAL protocol message; data is information relative to the function requested.

4. MSG-to-MSG Protocol for the ARPANET

4.1 Implementation of MSG-to-MSG paths by ARPANET connections.

Section 3 introduced the notion of "MSG-to-MSG paths" across which inter-host MSG messages are sent. A single such MSG-to-MSG path exists between each pair of host MSGs.

MSG-to-MSG paths are virtual entities in the sense that they are implemented by ARPANET host/host protocol connections. At any given time, a given MSG-to-MSG path may be implemented by zero, one or more pairs of ARPANET host/host connections. The standard byte size for ARPANET connection which implement MSG-to-MSG paths is 8 bits.

The set of ARPANET connections which implement an MSG-to-MSG path are equivalent in the sense that any legal inter-host MSG message can be sent over any one of the ARPANET connections in the set.

To send a message to another MSG, an MSG selects one ARPANET connection from the set that implements the MSG-to-MSG path and transmits the message over the connection. If no such ARPANET connection exists, the sending MSG must act to establish one.

4.2 Establishing the ARPANET connections.

A pair of ARPANET connections which supports an MSG-to-MSG path is established via an ICP to a "well known" contact socket in the normal way. The contact socket for MSG is 27 (decimal) = 33 (octal).

After a new pair of connections is established by an ICP, the pair of MSGs must engage in a synchronization exchange before they can use the connections to carry the inter-MSG messages defined in Section 3. The purpose of this MSG-MSG synchronization is to allow the two MSGs to exchange their current "incarnation" numbers and any other information pertinent to subsequent interaction via the connection pair.

An MSG incarnation number identifies a particular period of MSG service. (We frequently use the term "MSG incarnation" to mean such a period of MSG service.) A period of MSG service ends and a new period of MSG service begins when an MSG re-initializes itself. This typically occurs after its host has restarted or the MSG itself has crashed and been restarted. An MSG is expected to know its current incarnation number and to change its incarnation number when a new period of service begins. (An MSG could do this by storing its incarnation number in a file which is preserved over host and MSG crashes. When a new period of service begins, the MSG could increment the stored incarnation number and use the number obtained to identify the new period of service.)

As noted in Sections 1 and 2, MSG process names include an incarnation number component which serves to identify the incarnation of the MSG that generated the process name and is responsible for managing the process. The MSG incarnation number component of a process name is used to determine whether the process named is one that currently exists or is an obsolete one which was managed by the MSG during one of its previous periods of service.

The MSG-to-MSG protocol for the synchronization exchange is:

1. The MSG that initiated the ICP initiates the synchronization exchange by using the send connection of the pair to send the message:

   SYNCH (my-incarnation, your-incarnation, version, data)

   where:

my-incarnation identifies the current incarnation of the initiating MSG.
your-incarnation is empty.
version identifies the version of the MSG-to-MSG protocol to be used on this connection.
data is other synchronization information. (To be defined in the future.)

2. The other MSG responds to the SYNCH by using the send connection of the pair to send the message:

   SYNCH (my-incarnation, your-incarnation, version, data)

   where:
   my-incarnation identifies the current incarnation of the responding MSG.
   version identifies the version of the MSG-to-MSG protocol to be used on this connection.
   your-incarnation echoes the incarnation number specified in the initiating MSG's SYNCH message.
   data is other synchronization information.

After the synchronization exchange is completed, the connections may be used to carry any of the inter-MSG messages defined in Section 3 until the connections are closed (see Section 4.3 below).

An MSG may wish to ascertain that the entity at the other end of a new connection pair is indeed another MSG before it commits any of its host resources to acting upon protocol messages received over the new connection. Section 4.4 below defines a procedure which MSGs may use to reliably authenticate one another.

4.3 Breaking the ARPANET Connections.

A pair of ARPANET connections to another host represents a resource which an MSG may not want to keep open indefinitely in the absence of MSG traffic. If an MSG were to close a connection pair unilaterally, messages in transit from a remote MSG could be lost or garbled. A protocol mechanism is defined for closing pairs of connections in an orderly manner that eliminates the possibility of such lost or garbled messages.

The protocol for closing a pair of connections is:

1. MSG sends an MSG-to-MSG "CLOSE" message over the send connection of the pair that is to be closed and then closes the send connection of the pair;

2. Upon receipt of an MSG-to-MSG CLOSE message an MSG is expected to: close the connection which carried the message; return a CLOSE message on the send connection of the pair (when it is convenient to do so); and close the send connection.

The protocol exchange defined above is the mechanism for breaking pairs of connections. At present, we refrain from specifying in detail a policy which defines when MSG may use this mechanism.

An MSG that does not wish to communicate with the entity that has initiated an ICP should respond to the initiator's SYNCH message by initiating the CLOSE protocol exchange. An MSG might choose to do this if the synchronization data supplied by the initiating MSG is incompatible or if the initiating entity can not properly be authenticated as another MSG.

4.4 Authentication of MSGs.

As noted in Section 4.2 above, it may be important for an MSG to be able to reliably authenticate the entity at the remote end of a pair of ARPANET connections as another MSG before host resources are committed to requests made by that entity. The problem here is one of mutual authentication. Each entity must authenticate the other as an MSG.

[In the absence of an authentication procedure, there is no way for an MSG to determine whether the entity at the remote end of a connection is another MSG or a bogus process which follows the MSG-to-MSG protocol. Failure to distinguish between an MSG and a process masquerading as an MSG could result in the inadvertent disclosure of private information or unaccountable use of expensive resources.]

The use of passwords is one approach to MSG authentication. Only an MSG would know the password and thus be able to properly identify itself to another MSG. We reject the password mechanism as unreliable and operationally impractical for the following reasons:

1. Use of a password requires that the password be stored in the sending program or be accessible to it in some way thereby increasing the likelihood that the privacy of the password will be compromised.

2. If a password is compromised, it must be changed at both sending and receiving hosts; this represents a synchronization problem.

3. Truly secure authentication would probably require passwords for each pair of hosts; this would require N*N passwords for an N host NSW.

The mechanisms to be used for MSG authentication are based upon the properties of ARPANET host/host communication. First, we assume that the ICP is a secure procedure. That is, we assume that a host can guarantee that MSG is the only entity that has access to the MSG ICP contact socket and that MSG is the only entity that has access to the connections resulting from the ICP. This is the standard assumption made in the ARPANET regarding the ICP. Thus, the authenticity of the entity responding to an MSG ICP as an MSG is based upon the security of the ICP procedure.

The authentication problem that remains is that of authenticating the entity that initiates the ICP. This

authentication can be achieved in a manner similar to that of the ICP responder. Just as a single well known ICP contact socket is defined, a collection of well known "ICP-from" sockets (i.e., sockets from which ICPs are initiated) could be defined. (A collection of ICP-from sockets are required due to the nature of the ICP which prevents reuse of the ICP-from socket until the connections resulting from the ICP are discarded.) A host would be required to limit access to the ICP-from sockets (and the connections that result from the ICP) to MSG just as it is required to limit access to the ICP contact socket (and the connections that result from the ICP). If this were to be done, an MSG responding to an ICP could authenticate the initiating entity as an MSG by checking that the socket from which the ICP was initiated was one of the well known ICP-from sockets.

Some hosts find it inconvenient to limit access to a collection of sockets but have no difficulty in controlling access to a connection once it is established. Therefore, a variation of the above approach is used for authenticating initiating MSGs. A single send socket is defined for MSG authentication; access to the MSG authentication socket is limited to MSG. The authentication socket is to be maintained by MSG in a listening state. In response to an RFC for the authentication socket, MSG should open the requested connection (with byte size = 32) and send a specification of the sockets which it is currently using in active MSG-to-MSG connections. The connection should then be closed and the authentication socket returned to the listening state.

An MSG at host A responding to an ICP initiated by a remote entity at host B can authenticate that entity by the following simple procedure:

1. The MSG at A notes the remote sockets, S1 and S2, used in the connections that result from the ICP.

2. It opens a connection to the authentication socket at B, reads the socket specification that the MSG at B sends, and closes the authentication connection.

3. If the remote sockets, S1 and S2, are included in the specification then the entity at B is an MSG; otherwise, it is not. (Note that when the MSG at B initiates an ICP to the MSG at A, it must remember the sockets it uses so that it can include them in the socket specification sent to the MSG at A.)

The reliability of this authentication procedure depends upon the ability of host B to insure that only MSG has access to the authentication socket and to the sockets named in the specification sent over the authentication connection. (This is exactly what host B must do to insure the security of ICPs to its well known contact sockets.) In addition, it requires that the MSG at A have means to reliably determine sockets in use at the remote end of connections. Socket identity is part of the information NCPs must exchange in order to open a host/host connection. Thus, the socket information is available to the NCP at A. The authenticity of the information depends upon the trustworthiness of the NCP at B. We assume NCPs to be secure; if they were not, there could be no reliably secure communication between ARPANET hosts.

The MSG authentication socket is 29 (decimal) = 35 (octal). The specification of MSG sockets returned over the authentication connection may be a range of sockets or a list of sockets. A socket range is transmitted as 3 bytes:

```
byte 1:
0  indicates range spec
byte 2:
Sa
byte 3:
Sb
```

All sockets within the range defined by Sa and Sb (including Sa and Sb) are MSG sockets. A list of N sockets is transmited as N+2 bytes:

```
byte 1:
1  indicates list spec
byte 2:
N  the number of bytes that follow
byte 3:
S1
byte 4:
S2
 .
 .
 .
byte N+2:
SN
```

The MSG sockets are S1, S2, ..., SN.

4.5 Error Control for MSG-to-MSG Paths

ARPANET host to host communication is reasonably reliable. However, communication failures can occur. For example, host/host messages are lost occasionally. A lost host/host message may manifest itself at the MSG-to-MSG path level as a "hung" connection (if the message lost was a host/host allocate) or as a totally or partially lost MSG-to-MSG message (if the message lost was a host/host data message).

In addition, communication between a pair of hosts can be interrupted temporarily. The interruption may be the result of a transient network failure (e.g., the source or destination IMP crashes and is restarted) or a transient host service interruption (e.g., TENEX hosts occasionally experience BUGCHK interruptions and resumptions). At the MSG-to-MSG level this may manifest itself as a spontaneously closed host/host connection. If the connection was being used at the time, this could result in a lost or garbled MSG-to-MSG message.

Mechanisms to insure reliable communication in an environment where messages can be lost are reasonably well understood. These mechanisms typically require positive acknowledgement of all messages and the use of a time out and retransmission scheme. This generally requires that the communicating entities (in this case pairs of MSGs) use unique identifiers or sequence numbers to identify messages in transit and employ techniques for detecting duplicate messages (the message may have made it but its acknowledgement may have been lost). Note that these message identifiers serve to identify individual inter-MSG messages and are therefore different from the transaction identifiers used in the inter-MSG protocol to identify transactions that involve a number of inter-MSG messages.

The question here is:

> Should such a reliable transmission mechanism be used for error control on the MSG-to-MSG paths?

Our position with regard to error control for MSG-to-MSG paths is:

1. The most effective error control mechanism for the MSG-to-MSG application is that described by Cerf and Kahn (i.e., that used in the InterNet or TCP protocol).

2. The overhead incurred by using a TCP-like error control mechanism would not significantly degrade performance for the NSW MSG application.

3. Use of a TCP-like mechanism would approximately double the time and effort required to implement inter-host MSG.

4. The TCP mechanism can be made orthogonal to the MSG-to-MSG protocol and to a properly designed MSG implementation. That is, the information required to enable TCP-like error control would envelope inter-MSG messages. We estimate that 5 or 6 additional 8 bit bytes are required for each inter-MSG message to support TCP-like error control. Furthermore, we believe that the processing required to perform the error control function can occur in series with the "higher level" processing required to implement the MSG protocol.

It is not clear, at present, whether error control stronger than that normally provided by ARPANET host to host communication will be required by the NSW application. Therefore, the initial inter-host MSG specification does not include TCP-like error control for the MSG-to-MSG paths nor does the transmission format for inter-MSG messages include fields for the information required to support TCP-like error control. However, the MSG implementations should be done with the expectation that it may be necessary to add TCP-like error control later, should experience indicate that the lack of error control for the MSG-to-MSG paths is resulting in unacceptable performance.

5. MSG-to-MSG Transmission Formats for the ARPANET

This section specifies in detail the formats for the MSG-to-MSG protocol commands as sent over ARPANET connections. Only the syntax of the commands is specified here; for a discussion of the semantics of the MSG-to-MSG protocol see section 3 of this document.

5.1 General format for MSG-to-MSG messages:

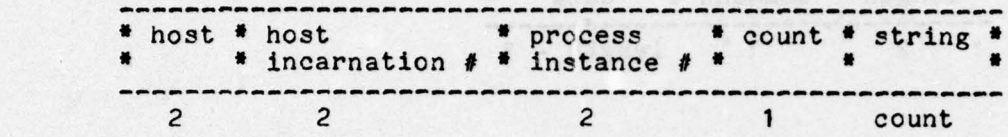
An MSG-to-MSG message is a sequence of 8 bit bytes. The first two bytes contain the length of the message in bytes; the third byte is a command code that identifies an MSG-to-MSG protocol item; and the remaining bytes contain information relative to the command.

length	command	data
2	1	length - 3

5.2. Formats for Message Components

1. Process names:

As described in Section 2, a process name has four components which specify a host, a host incarnation number, a generic process class, and a process instance number. The representation for process names at the MSG-to-process interface is:

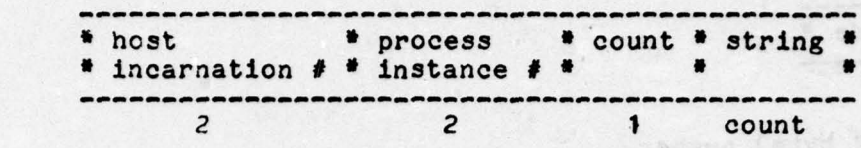


Host is a 16 bit host address. (Whether the host address is an ARPANET host address or an NSW host address whose correspondence to an ARPANET host address is defined by a table MSG maintains is to be decided shortly.) If MSG is modified to allow processes with no generic names, the null generic name will be represented by a zero length string.

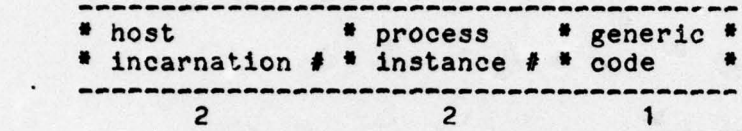
For a generically addressed message the destination process name is only partially specified. Either only the generic process class is specified, or only the host and generic class are specified in a generically addressed message. The other components are left unspecified. "Unspecified" is a special value used in generically addressed messages for host, host incarnation #, and process instance #. Unspecified is represented by two zero bytes.

When a process name appears as the parameter of an MSG-to-MSG message, the host component of the name need not be represented explicitly since it is implicit from the hosts of the sending and receiving MSGs. There are two representations for process names at the MSG-to-MSG level: normal and compact. The only difference in the two is the representation of the generic process class. In the normal represenation the generic class is represented by a string whereas in the compact form it is represented by a one byte generic class code. MSG implementations must be able to deal with both representations for process names. The compact representation is defined to allow for greater transmission efficiency. Use of the generic codes is internal to MSG in the sense that the codes never appear in a process name given by MSG to an MSG process or accepted by MSG from an MSG process. Generic class codes for the NSW will be defined in the near future.

Normal Format: count < 128 (5 + count bytes)



Compact Format: Generic code >= 128 (5 bytes)

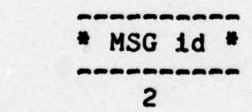


Generic code = 128 + n (n < 128)
where n = integer which specifies a generic class
n = 0 - null (i.e., process has no generic name).

2. Host Incarnation #:

16 bit (2 byte) number.
0 = unspecified (used for generically addressed messages)
1-255 reserved for special use

3. MSG transaction Identifiers (source-id, destination-id)



16 bit (2 byte) number.

4. Alarm code

```
----------
* acode *
----------
    2
```

16 bit (2 byte) number.

5. Failure/Rejection codes

```
-----------
* reason *
-----------
    2
```

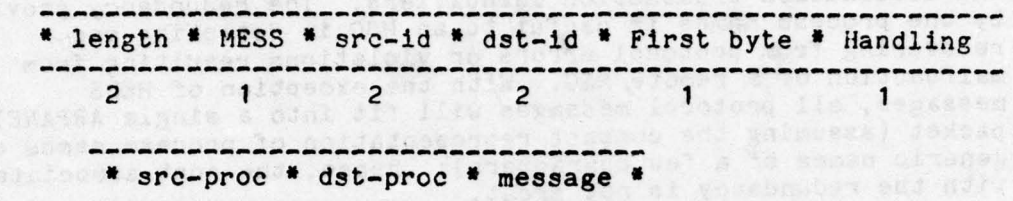
16 bit (2 byte) number.
See descriptions of individual messages for discussion of specific codes. Values have not yet been assigned, nor are those codes given necessarily exhaustive.

5.3 Identifying Transactions.

In the format specifications that follow all inter-MSG messages concerned with inter-process transactions carry the source and destination process names as well as the MSG source and destination transaction identifiers. The redundancy provided by the process names is useful to an MSG in detecting and recovering from protocol errors or violations resulting from malfunction of a remote MSG. With the exception of MESS messages, all protocol messages will fit into a single ARPANET packet (assuming the compact representation of process names or generic names of a few characters); hence, the cost associated with the redundancy is not great.

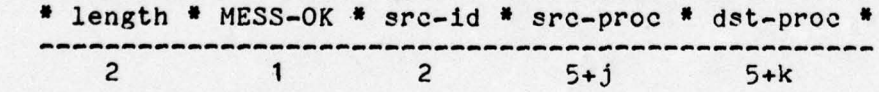
5.4 MSG-to-MSG protocol messages

1. MESS(src-proc, dst-proc, handling, src-id, dst-id, message)



length = 19+j+k+M
j = # chars in source generic name / 0 if compact format.
k = # chars in destination generic name / 0 if compact format.
MESS = 8 (10 octal)
Handling = bit flags (numbered 0-7 from left to right)
bit 0 - generically addressed message
bit 1 - sequenced message
bit 2 - synchronization mark on message
bit 3 - immediate decision on delivery (prohibit HOLD)
First byte - Position of first byte of the message (zero is the position of the first byte of the length field of the MSG-to-MSG message)

2. MESS-OK(src-proc, dst-proc, src-id)



length = 15+j+k
MESS-OK = 9 (11 octal)

3. MESS-REJ(src-proc, dst-proc, src-id, reason)

length	MESS-REJ	src-id	reason	src-proc	dst-proc
2	1	2	2	5+j	5+k

length = 17+j+k
MESS-REJ = 10 (12 octal)
reason = To be specified, but including:
dst-proc unknown
no buffer space
message queue for process full

4. MESS-HOLD(src-proc, dst-proc, src-id, dst-id)

length	MESS-HOLD	src-id	dst-id	src-proc	dst-proc
2	1	2	2	5+j	5+k

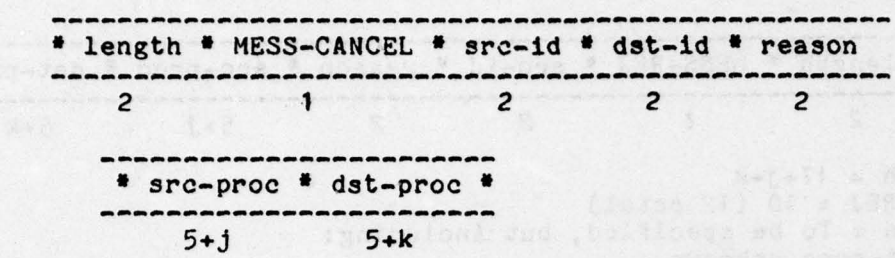
length = 17+j+k
MESS-HOLD = 11 (13 octal)

5. HOLD-OK(src-proc, dst-proc, src-id, dst-id)

length	HOLD-OK	src-id	dst-id	src-proc	dst-proc
2	1	2	2	5+j	5+k

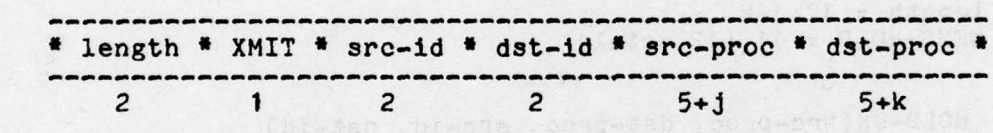
length = 17+j+k
HOLD-OK = 12 (14 octal)

6. MESS-CANCEL(src-proc, dst-proc, src-id, dst-id, reason)



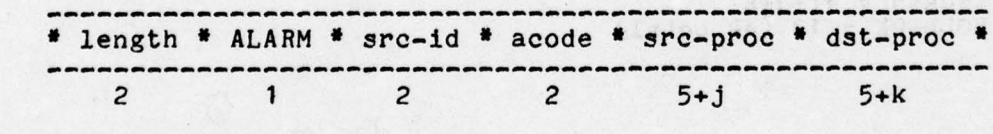
length = 19+j+k
MESS-CANCEL = 13 (15 octal)
reason = To be specified, but including:
- src-proc unknown
- src-id unknown
- message rescinded
- src-proc terminated
- no buffer space

7. XMIT(src-proc, dst-proc, src-id, dst-id)



length = 17+j+k
XMIT = 14 (16 octal)

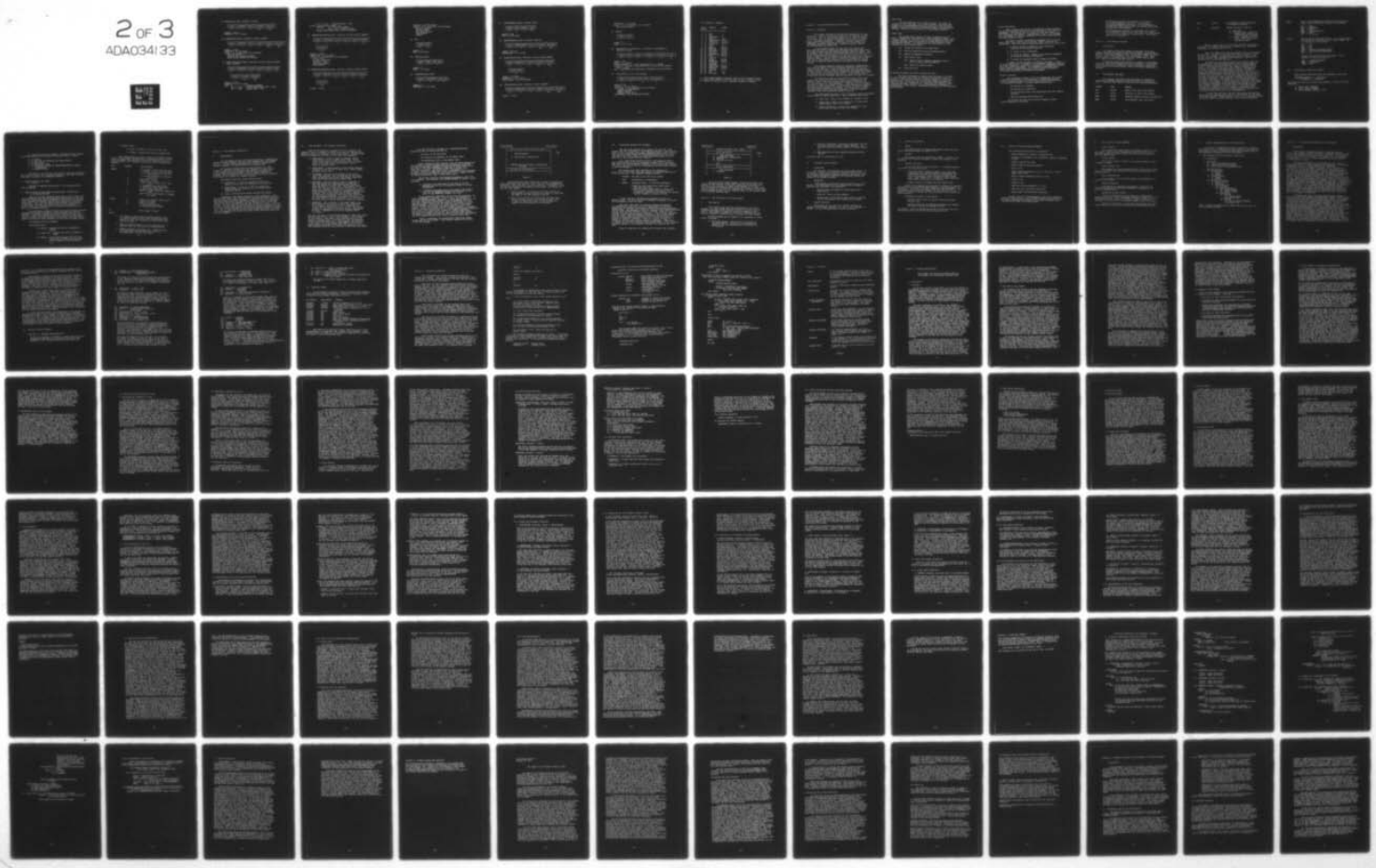
8. ALARM(src-proc, dst-proc, src-id, acode)

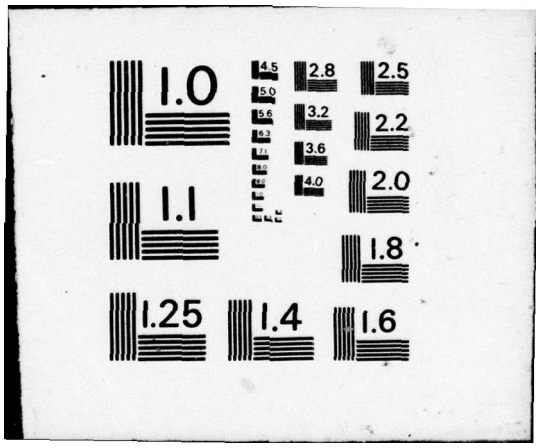


length = 17+j+k
ALARM = 16 (20 octal)

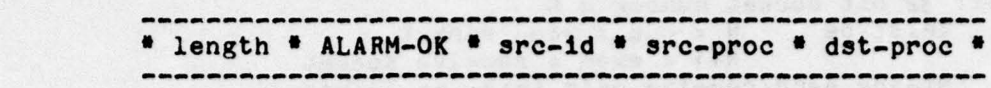
AD-A034 133 MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD F/G 9/2
NATIONAL SOFTWARE WORKS.(U)
SEP 76 R MILLSTEIN F30602-76-C-0094
UNCLASSIFIED CADD-7603-0411 RADC-TR-76-276-VOL-1 NL

2 OF 3
ADA034133



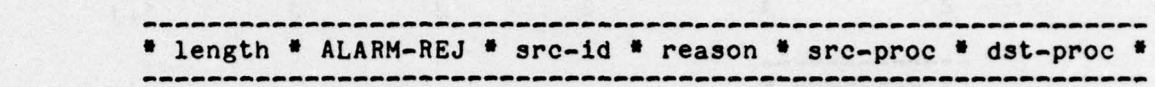

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

9. ALARM-OK(src-proc, dst-proc, src-id)



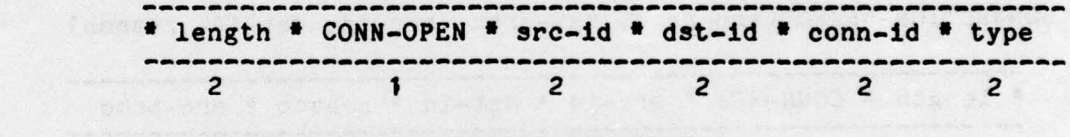
length = 15+j+k
ALARM-OK = 17 (21 octal)

10. ALARM-REJ(src-proc, dst-proc, src-id, reason)



length = 17+j+k
ALARM-REJ = 18 (22 octal)
reason = To be specified, but including:
dst-proc unknown
dst-proc not accepting alarms
alarm already queued for dst-proc

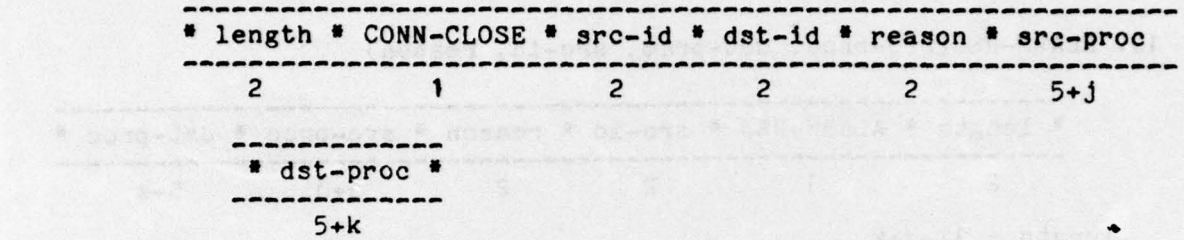
11. CONNECTION-OPEN(src-proc, dst-proc, src-id, dst-id, conn-id, type, socket)



length = 25+j+k
CONN-OPEN = 20 (24 octal)
type: 0 - Teletype (TELNET)
bit 0 + size - binary send/receive pair + size
bit 1 + size - binary send + size

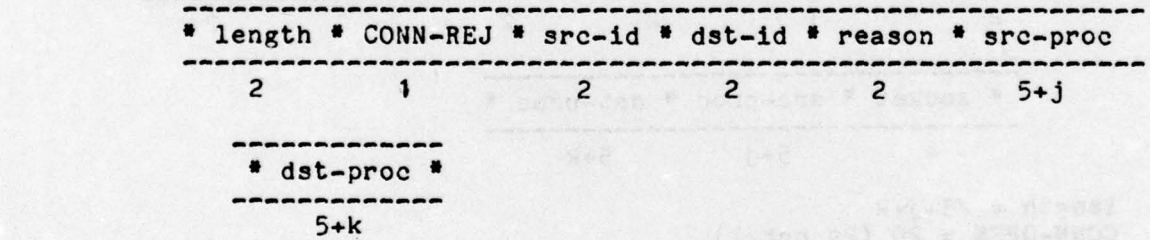
```
        bit 2 + size    - binary receive + size
socket: 32 bit socket number = N
        Teletype        N = odd = send socket
                        N+1 = even = receive socket
        Binary send/receive pair (same as Teletype)
```

12. CONNECTION-CLOSE(src-proc, dst-proc, src-id, dst-id, reason)



length = 19+j+k
CONN-CLOSE = 21 (25 octal)
reason = To be specified, but including:
- normal close
- src-proc terminated
- timeout of open
- byte-size mismatch
- type mismatch

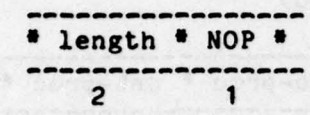
13. CONNECTION-REJECT(src-proc, dst-proc, src-id, dst-id, reason)



length = 19+j+k

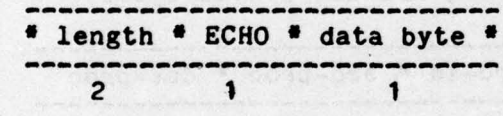
CONN-REJ = 22 (26 octal)
reason = To be specified, but including:
dst-proc unknown
dst-id unknown
byte-size invalid
type invalid
timeout

14. NOP



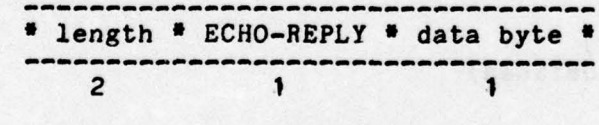
length = 3
NOP = 0 (0 octal)

15. ECHO(data byte)



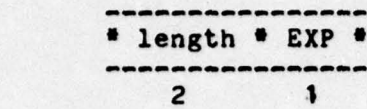
length = 4
ECHO = 1 (1 octal)

16. ECHO-REPLY(data byte)



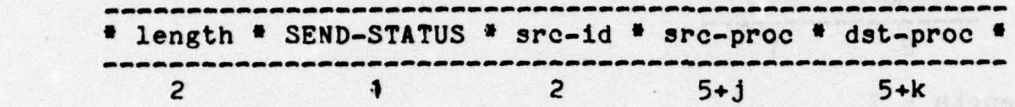
length = 4
ECHO-REPLY = 2 (2 octal)

17. EXPERIMENTAL(command, length, data)



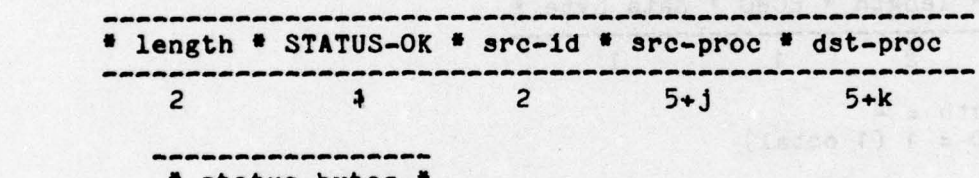
length = 4+N
EXP = 24 (30 octal)

18. SEND-STATUS(src-proc, dst-proc, src-id)



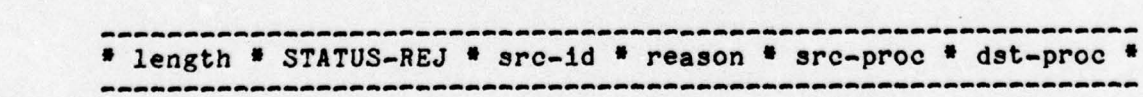
length = 15+j+k
SEND-STATUS = 4 (4 octal)

19. STATUS-OK(src-proc, dst-proc, src-id, status bytes)



length = 15+j+k+N
STATUS-OK = 5 (5 octal)
status bytes = (to be defined)

20. STATUS-REJ(src-proc, dst-proc, src-id, reason)



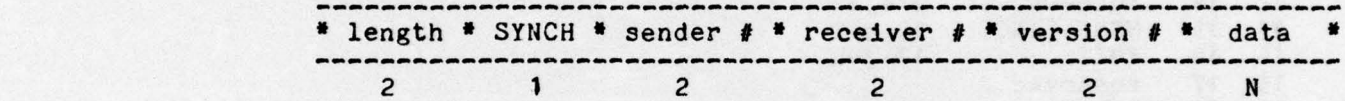
length = 17+j+k

STATUS-REJ = 6 (6 octal)
reason = To be specified, but including:
dst-process unknown

21. CLOSE()

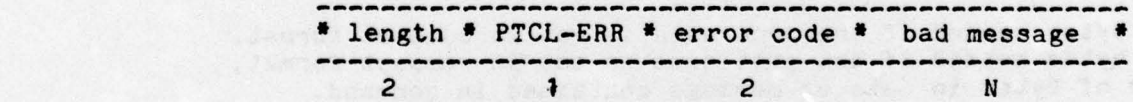
length = 3
CLOSE = 7 (7 octal)

22. SYNCH(sender's incarnation #, receiver's incarnation #, version #, data)



length = 9+N
SYNCH = 3 (3 octal)
sender/receiver #'s = Host incarnation #'s = 2 bytes
version # = version of MSG protocol to be used by the sending MSG = 2 bytes
data = additional synchronization information (to be defined)

23. PTCL-ERR(error code, bad message)



length = 5+N
PTCL-ERR = 25 (31 octal)
error code = To be specified, but including:
command not implemented
command unknown
command syntax error
bad message = The bad MSG-MSG message.

5.5 Summary of Commands

Code Dec	Code Oct	Command	Length
0	0	NOP	3
1	1	ECHO	4
2	2	ECHO-REPLY	4
3	3	SYNCH	9+N
4	4	SEND-STATUS	15+j+k
5	5	STATUS-OK	15+j+k+N
6	6	STATUS-REJ	17+j+k
7	7	CLOSE	3
8	10	MESS	19+j+k
9	11	MESS-OK	15+j+k
10	12	MESS-REJ	17+j+k
11	13	MESS-HOLD	17+j+k
12	14	HOLD-OK	17+j+k
13	15	MESS-CANCEL	19+j+k
14	16	XMIT	17+j+k
15	17	reserved	
16	20	ALARM	17+j+k
17	21	ALARM-OK	15+j+k
18	22	ALARM-REJ	17+j+k
19	23	reserved	
20	24	CONN-OPEN	25+j+k
21	25	CONN-CLOSE	19+j+k
22	26	CONN-REJ	19+j+k
23	27	reserved	
24	30	EXP	4+N
25	31	PTCL-ERR	5+N

j = Extra bytes needed if src-proc name is not in compact format.
k = Extra bytes needed if dst-proc name is not in compact format.
N = Number of bytes in data or message contained in command.

^L

Chapter 4: File Package Design Specification

Section 1: Overview

The primary function of the NSW File Package (FP) is the creation of a copy of an NSW file which will be suitable as input for a tool. That is, the primary concern is to make the output of one tool (e.g., an editor) acceptable as the input to another (e.g., a language processor). Secondary functions include the "importing" of files external to NSW into its file system, and associated peripheral operations: creating listings, reading and writing tapes and cards, etc.

A File Package resides on every NSW Tool Bearing Host. Every such host also includes some NSW-controlled file space; that is, files to which only the NSW Works Manager (WM) has access. The WM contains a File Catalog of NSW files; the NSW file system consists of all files which have entries in the WM file catalog. At one level, an NSW file can be viewed as an NSW name and a list of names of physical copies. The NSW name is (generally) assigned by the NSW user and is syntactically uniform for the entire NSW file system. The list of physical copies includes the complete network address of each.

The multiple physical copies are logically indistinguishable, and the choice of the one actually selected by WM/FP is of no concern to the NSW user. This is clearly the case when physical copies reside on several machines running the same operating system (a "host family"), but what of two "copies" on different host types? We use only the following general notion: At the logical level, the physical copies represent identical sequences of lines (or records).

The WM grants access to given files by given users; once granted it is the job of the FP to make a copy suitable for the desired access. Note that unless a local copy is available two FP's are involved in a copy operation: "receiver" FP (on the host desiring the copy) and "donor" FP (on a host containing an original). The case in which the two FP's reside on hosts in the same family is used to good advantage (see below). Of the two FP's, however, the receiving FP drives the copy procedure: it has the task of creating a copy with equivalent logical structure.

The receiving FP is given a list of physical copies (originals) and the right to select among them. Three situations may arise:

1) Local copy. There is an original on receiver's host.

2) Family copy. There is an original on a foreign host supporting the local file formats.

3) Forced translation. There is an original on a host which does not support local file formats.

Local Copy

The most efficient way to make the copy is by using an original on the local host, for obvious reasons. The local copy procedure can be implemented entirely within the local operating system, but serves to identify procedures common to other modes of copying. Only one FP is involved in a local copy operation.

Family Copy

The analogy with archival of files on magnetic tape is useful. Suppose the host operating system supports archival. Such a process encodes an arbitrary file in serial fashion so that the original file may later be reproduced in programmatically indistinguishable fashion. We characterize the save portion of the operation by the following steps:

A1. Locate the contents of a named file.

A2. Read its physical structure characteristics.

A3. Record its physical structure.

A4. Until end-of-file

A4R. Read a "block" (machine dependent unit) of the file and serially encode it.

A4W. Write the block on tape.

A5. Close the file.

A similar procedure is used to restore the file.

The "save" procedure is exactly what the donor's FP needs to send a family copy, while the "restore" procedure is used by receiver; a one-way MSG direct connection assumes the role of the magnetic tape. Thus, two hosts in the same family can exchange files with as much fidelity as one finds with save/restore. Host families will use a private (i.e., determined by a consensus of implementors in that family) dump/restore encoding for file transfers among members.

Forced Translation

When no family copy exists, the receiving FP must attempt to reproduce the logical structure of a foreign file in the local file encodement. The receiver has the right to select the "donor", or host containing the original. Once selected, donor's file package will be requested to send an encodement of the file, which receiver must then decode and store.

We choose a single "intermediate language" (IL) to be used when a donor must encode the file and its logical structure for translation. There are several properties which such an IL should have.

1) It should preserve a maximum of the information contained in the logical structure.

2) It should be host independent.

3) It should be compact - indeed, files must be compressed for efficient network transmission.

4) It should be reasonably easy to encode and decode.

A chapter of this document is devoted to IL specification. It is our feeling that no physical file encodement adequately satisfies the above objectives; therefore, we propose an intermediate language which is nobody's encodement. Its compression techniques apply equally well to binary files and to files of text, and even to the private intra-family save/restore encodement.

A Note on Devices

The preliminary design of the File Package does not include a detailed presentation on the handling of non-sharable devices. It was a necessary omission, and the subject will be covered fully in a subsequent document. The main topics omitted were:

- How devices are assigned;
- How devices are controlled;
- How the device handler will communicate with the computer operator;
- What the standard device names are.

Two "hooks" are built into the File Package to allow future expansion in this area:

- The physical copy name (see chapter 2, section II) allows host-dependent strings for the file name and location/physical structure. A file residing on a non-sharable device can be named and located using these physical copy name elements.

- The intermediate language for translation (see chapter 5) has constructs which would allow files on a tape volume to be moved as a logically related group. (See subsection II.5 in paricular.)

Section 2: File Package Functions

I. Introduction

This chapter deals with functions performed by the File Package, together with their arguments and results. The functions which can be called externally (i.e., by the Works Manager or another FP) are: copy a file, delete a file, and analyze a file. (Here "file" refers to a physical copy of a file, rather than an NSW file, which is a set of physical copies.)

In the next section we introduce the syntax of the physical copy name, which is the argument to any FP function. In the third section we discuss the delete and analyze functions, leaving the copy function to the next chapter, due to its complex communication problems.

II. The Physical Copy Name

The physical copy name is the main piece of information exchanged in messages between the WM and the FP, and between two FP's. A physical copy name is a list with the following components:

ELEMENT	TYPE	MEANING
HOST	string	Name of host where file resides.
DIR	string	Directory in which file resides.
PSWD	string	Password needed to access the directory.
NAME	string	Host-dependent name (see below).

PHYS	string	Host-dependent location/physical structure information
LS	character	Logical structure. One of: A - Text file F - Formatted text: same as A, except that overprint and line-skips may be present B - Binary: sequences of n-bit bytes, with all 2**n bit patterns allowed.

(Note: A "text file" is a file which contains only graphic characters. No format effectors may be present (i.e. linefeeds, horizontal and vertical tabs, etc.).)

HOST, DIR, PSWD, and LS are all optional (i.e., each element may be null). If HOST is null, the local host is assumed. If LS is null, F is assumed. No assumptions are made for DIR or PSWD: as will be shown, their meaning is host-dependent, therefore the defaults will be host-dependent also.

The NAME will be a string containing the file's name; this is necessarily in host-dependent format. PHYS contains physical structure and/or location information, again in host-dependent format. Since these formats will vary between families, the WM will make no attempt to interpret these strings. Furthermore, an FP on a host not in the same family as the host on which a given copy resides does not interpret the string. The FP, if it chooses to copy that particular physical file, provides the name to an FP on that file's host. That FP, upon receiving the name of the file, can then make use of the host-dependent strings to locate the file in question. (More precisely, it knows how to map the "canonical" name which it receives to a name which makes sense to the host's file system.) In other words, if a physical file resides on a host in host family H, the host-dependent string for that file will contain information which need only be intelligible to any other H-host. Therefore, for each family H in the NSW system, we leave the decision as to the information contained in NAME and PHYS and the mapping from the H-family naming structure to the "canonical" naming structure to the H-implementors.

To show that different families' naming structures may be mapped onto our canonical name form, we give examples for the TENEX and OS/360 families. (These examples are not to be construed as being in any way final or required mappings, merely possible mappings.)

TENEX: File is MTAO:<BABBAGE>DIFF.ANALYZER;12 with password 'CHAS' and is a text file residing on BBN-TENEXB.

```
HOST  = 'BBNB'
DIR   = '<BABBAGE>'
PSWD  = 'CHAS'
NAME  = 'DIFF.ANALYZER;12'
PHYS  = 'MTAO:'
LS    = 'F' or null
```

OS/360: File is NSW.PDS.PROGRAMS(FILEPKG). It is uncatalogued, but resides on a disk with serial number NSWDSK. It is a binary file at UCLA-CCN.

```
HOST  = 'CCN'
DIR   = null
PSWD  = null
NAME  = 'NSW.PDS.PROGRAMS(FILEPKG)'
PHYS  = 'UNIT=3330,VOL=SER=NSWDSK'
LS    = 'B'
```

Suppose the above file were catalogued. Then a variant of the above would be:

```
DIR   = 'NSW.PDS.PROGRAMS'
NAME  = '(FILEPKG)'
PHYS  = null
```

III. The Functions Delete and Analyze

Both functions described below are presented in the form:

```
FUNCTION(argument-1,...,argument-n)
     -> (result-1,...,result-m).
```

The result is a list of at least three elements, of which the first three are:

A: Result code (INTEGER)
B: Error number (INTEGER)
C: Error message (STRING or null)

On success A=B=0 and C is empty. Otherwise, B and C detail the actual error and A describes the recovery type as follows:

0 => success
1 => file not yet available (try again later)
2 => user error
3 => internal FP error
4 => net trouble (unable to establish/maintain a direct connection)
5 => remote FP error

The cases A=4 and A=5 can arise during a file copy operation. The values of B and C are currently undefined in the case of errors. They will be defined in a subsequent document.

1. DELETE (physical copy name)
   -> (A,B,C)

The name is removed from the host's file catalog and the file is deleted.

2. ANALYZE (physical copy name, structure-type, content-type)
   -> (A,B,C,result-list)

Before a file copy operation takes place, the FP which will send the physical file may be requested by the receiver of the file to provide information about the physical structure of the existing file. Invoking the ANALYZE function will provide this structural information. This will aid machines which require pre-allocation of file space, and will help insure that the file created as a result of the copy operation will be physically acceptable to the tool which requested the file.

We conceptually divide the world into record-oriented machines and paper tape-oriented machines. Record machines see files as sequences of logically grouped bytes (these "groups" being called records). Paper tape machines see files as streams of bytes with no logical grouping structure overlaid on the bytes. (The name "paper tape" is not meant to refer literally to that specific device alone.)

The arguments are both integers:

a) Structure type:

0 => Record - analyze the file as a sequence of records

1 => Paper tape - analyze the file as a stream of bytes

2 => Either - the FP which analyzes the file may analyze it as either record or paper tape, whichever is more congenial.

b) Content type:

0 => Text - analyze the file as a text file

1 => Binary - analyze the file as a binary file

The "result-list" is a list of three lists COMMON, RECORD, and PT, which contain (respectively) information common to both a record and paper-tape analysis, information pertaining only to a record analysis, and information pertaining only to a paper-tape analysis.

LIST	ELEMENT	VALUE/MEANING
COMMON	1	0: Argument content-type was text 1: Argument content-type was binary
	2	0: PC-name component LS indicated text (A or F) 1: PC-name component LS indicated binary (B)
	3	Byte size, in bits
	4	Word size, in bytes
	5	Longest line, in bytes (text files only)
	6	0: Sequenced file (text only) 1: Unsequenced file (text only) 2: Can't tell (text only)
	7	Number of pages (empty, if paging not used)
	8	Page size, in bytes (empty, if paging not used)
RECORD	1	Logical (or maximum) record size
	2	Records per block
	3	Number of records
	4	0: Fixed-length records 1: Variable-length records
PT	1	Total number of bytes

Notes:

1. The argument content-type determines whether a text or binary analysis is done, even though content-type may differ from that indicated by the physical copy name's LS value.

2. Page (in COMMON elements 7 and 8) indicates size of physical pages (in the sense of virtual memory).

3. COMMON is always a non-empty list. RECORD is empty if a paper-tape analysis was specified (i.e., structure-type = 1), and vice versa.

Section 3: File Package Communication

I. Introduction

In this chapter we will outline the patterns of communication between the Works Manager (WM) and a File Package (FP), and between two FP's. We only discuss communication in the case where the WM requests creation of a logical copy of an existing file.

We implicitly assume that any given instance of a File Package is capable of handling one file request at a time, where a file request would be a message from the WM or another File Package instance requesting that a file be copied, deleted, or analyzed. At this stage of the design we do not allow file requests to be queued by a given File Package instance.

File motion falls into three distinct cases:

1) Importation: a file from outside the NSW file system is moved into the system as a new NSW file;

2) Exportation: a file in the NSW file system is moved outside the system, i.e., to a host's local operating system;

3) Copy: a file is moved within the NSW file system.

Importation will almost always be a local operation. Every NSW host will have a File Package (which directs the importation) and NSW-controlled file space into which to copy the file to be imported. Thus the WM will relay the importation request to an FP on the same host as the file to be imported, and the resulting copy can be done locally, the only exception arising when all NSW file space on the host is full. In that case, another host must be chosen to accept the copy. It is more likely that export and copy will involve interaction between FP's. (N. B. File motion can only occur between NSW hosts, since NSW will not implement file transfer protocols other than those specified by this document.)

II. Works Manager - File Package Interaction

When the WM receives a request for a file transfer, the requestor can be a Foreman (on behalf of a tool, in the case of a copy) or a user at a terminal (in the other cases). The WM invokes the File Package COPY function, which takes five arguments:

1) SOURCE-LIST: a list of names of physical copies from which the FP can choose one to copy. In the import case, there is only one source file to choose from; in the other cases there may be a set of names of physical files.

2) PHYS-STRUCT: a specification of the desired physical structure of the copy to be made (e.g., record length, blocking factor, etc.).

3) COPY-NAME: the name to be entered in the host's local file catalog. This is specified in the export case, and may be null in the other cases, signifying that the FP should generate a unique name.

4) HOST-PREF: This is a pair of lists, originated in the copy case by the tool/Foreman, by which the tool can specify its likes and dislikes concerning where the copy is to come from. The elements of the first list specify preferred host families (i.e., the FP would try to select a physical file residing on one of these families); the second list specifies unacceptable families. One or both lists may be empty, meaning "no particular preference" (for first list null) and "no unacceptable families" (for second list null). Not all host families need be specified, i.e., the union of the lists need not be the universe of NSW host families.

5) WORKSPACE: This specifies a file space (apart from NSW-controlled file space) in which the receiver FP is to create a working copy (i.e., a physical copy created for the exclusive use of a tool). The Foreman creates the workspace and informs the Works Manager of its name. WORKSPACE is null if no working copy is to be made.

Note that the name of the NSW-controlled file space into which the new copy will go is not a COPY argument. The receiver FP will select a file space, thus relieving the WM of the burden of keeping track of the status of the file space(s) (there may be more than one). This suggests the existence on each host of a "File Package Data Base", a short file which contains the names of all NSW file spaces on that host. This approach allows for easy addition of more NSW file spaces.

The COPY function is invoked via a Sendgenericmessage addressed to any FP on the same TBH as

- the tool, in the copy case;
- the source to be imported, in the import case;
- the destination file, in the export case.

After invoking the COPY function, both the Works Manager and the receiver FP execute Acceptalarms and Enablealarms primitives. The File Package must recognize at least two WM alarm codes (to be specified). One alarm represents an NSW user's typing ^X to abort the processing. The second alarm is analogous to ^T on TENEX: the File Package must print a (necessarily host-dependent) message detailing the status of the transfer (e.g., number of bytes or records transferred, name of donor host, etc.).

The WM then executes a Receivespecificmessage to await the FP's response. This response (the CONFIRM message in figure 1 below) contains:

1) A physical copy name which is the name of the new physical copy (if the FP generated a name) or is null (otherwise).

2) A physical copy name which is the name of the working copy (if the WORKSPACE argument was non-null) or is null (if the WORKSPACE argument was null).

At the successful completion of the file copy operation, the value of WORKSPACE is checked. If it is null, the receiver FP sends the CONFIRM message. Otherwise, a local copy is made into the file space specified by WORKSPACE; a name is generated for this copy (the "working copy"), and the receiver FP includes it in the CONFIRM message. (The WM does not enter the name of the working copy in the NSW file catalog, but merely passes it back to the Foreman. The FP should then execute a Stopme primitive to disappear. (This could easily be modified to be a Receivegenericmessage primitive: the FP would lie dormant until awakened by another generically addressed file request from the WM. This approach might lighten the burden of process allocation by MSG if it turns out that requests for File Package activity occur fairly frequently within NSW.)

Figure 1 summarizes the communication between the Works Manager and the File Package. File transfer begins after receipt of the COPY message.

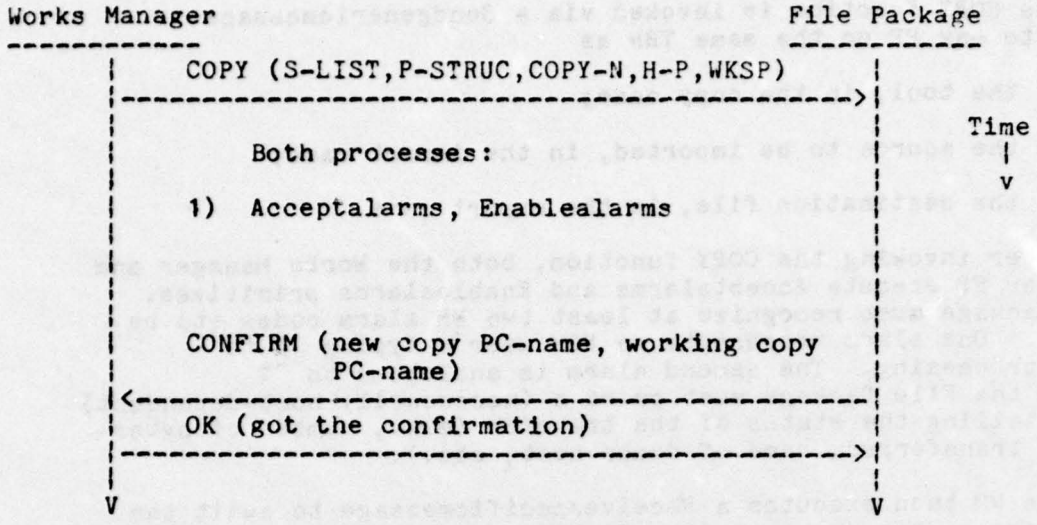


Figure 1.

The astute reader will notice that a number of potentially messy issues have been ignored. Due to the lack of implementation knowledge of MSG, this was a necessary omission; furthermore, we did not want to overspecify the design at this stage. We briefly note some of these issues:

1) Error checking: To insure that the WM received the new physical copy and working copy names, it could echo the names in its "OK" message to the FP.

2) What if the FP does not receive the "OK" after some period of time? This is the thorniest issue, and it would be beyond the scope of this document to suggest any one technique as the solution.

III. Interaction Between File Packages

When the Works Manager first contacts the FP with a COPY request, the WM is speaking to the Resource Allocator. The Allocator, given a list of physical copy names, determines which file to try to copy, and then calls either the Transmission module (to get a copy from one of the host's family) or the Translation module (to get a copy from a host outside the host family).

(If a copy exists on the FP's host, the Localcopy module (see "The Structure of the File Package", section III.3) is called. This module is self-contained; therefore in the rest of this section, "FP" will refer only to the Pure Transmission and Translation modules of the File Package.)

The sending host being determined, the receiving FP (i.e., the one to receive the copy) executes a Sendgenericmessage to an FP on that host. The "SEND-ME" message contains:

- PC-NAME: the name of the file to be copied;
- TYPE: transmission or translation;
- WIDTH: Connection width. If TYPE is translation:

  8 bits for text files (7-bit ASCII with a high-order zero bit)
  n bits for binary files, where n = donor byte or word size (selected by receiver from information returned by its invoking the ANALYZE function (Chapter 2, section III.2)).

It then executes a Receivespecificmessage to await the sender's reply. The donor executes a Sendspecificmessage to transmit its willingness to make the copy.

This done, both donor and receiver execute Acceptalarms and Enablealarms primitives (to allow communication of abnormal conditions) and Openconn primitives to establish a direct connection between them. The receiver's connection type should be BINARY-RECEIVE(m) and the donor's BINARY-SEND(m), where m = WIDTH. They will then transfer the file via the direct connection. The receiver, meanwhile, executes a Receivespecificmessage primitive to await an EOF message from the donor. When this message is received, the receiving FP executes a Sendspecificmessage primitive to inform the donor it is alright to close the connection. Both FP's execute a Closeconn primitive, and the donor executes Stopme to disappear. (The remark in section II about this primitive also applies here.) After informing the Works Manager of the completed transfer, the receiver does the same.

Figure 2 summarizes the communication between File Packages.

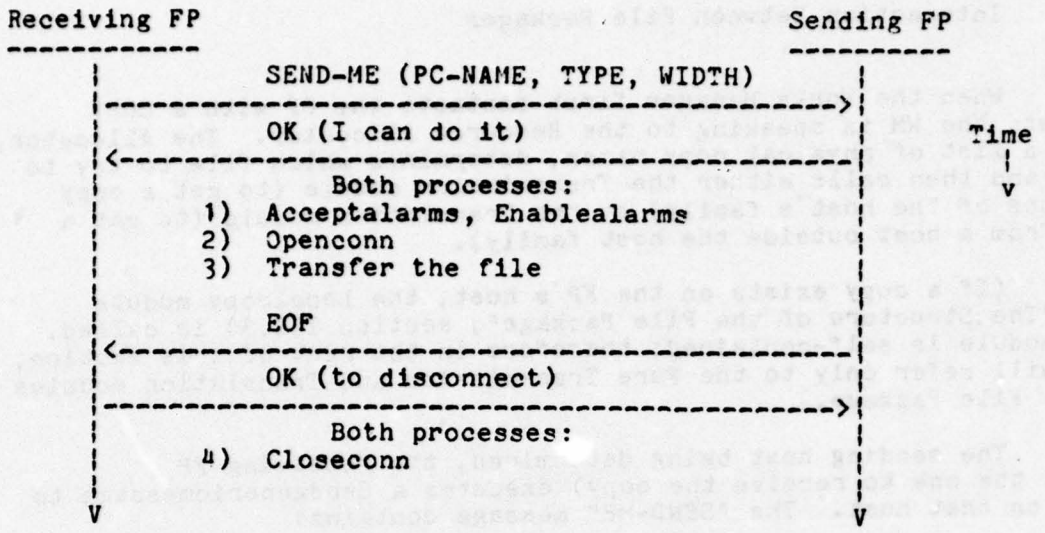


Figure 2.

As in the case of Works Manager - File Package interaction, we have glossed over all issues which arise when we consider what happens if file transfer or process communication errors occur. A file transfer error might be noted by the sending of an alarm, which could be echoed by the other FP, followed by a closing of connections. Another attempt to copy the file might be made, or the receiver might try a different physical copy.

Section 4: The Structure of the File Package

I. Introduction

This chapter deals with the organization of the File Package, i.e., the modules required for the FP to carry out the functions described in the second and third chapters of this report. This chapter is meant as a guide for implementors, and does not represent a mandatory way to organize the File Package.

The File Package can be thought of as consisting of three groups of modules:

1) Top-level control: This consists of modules for FP initialization, reception of messages/alarms passed by MSG, and interpretation of requests for FP functions.

2) Function controllers: There is a controller for each of the FP functions (COPY, DELETE, ANALYZE). The COPY controller has several layers of modules beneath it.

3) MSG interface and local operating system interface modules.

We discuss each of these groups in turn.

II. Top-Level Control Modules

1. Initializer

Any necessary initialization is done by this module. In particular, it makes use of the MSG interface modules to set up and execute a Receivegenericmessage primitive, specifying that when a generic message is received, the Dispatcher should pass control to the Request Analyzer.

2. Dispatcher

The Dispatcher functions in much the same way as a PL/I ON-unit. When MSG signals the File Package that a message or alarm has been delivered, the Dispatcher:

- verifies that the message or alarm comes from a valid source;
- saves the source process address;
- passes control to the appropriate routine, providing it with the message text or alarm code received.

3. Request Analyzer

This module is only called to analyze a request to perform a File Package function, i.e., one of COPY, ANALYZE, or DELETE. It calls the appropriate function controller.

III. Function Controllers

1. DELETE

This module calls the appropriate LOSIMs (local operating system interface modules) to delete a file.

2. ANALYZE

This module calls the appropriate LOSIMs to analyze a file and MSGIMs (MSG interface modules) to report the results of the analysis.

3. Resource Allocator

This is the COPY function controller. To perform resource allocation, it calls modules to:

- refine the list of physical copies, i.e., omit from consideration any copies residing on hosts which the tool/Foreman may have specified as unacceptable, and order the remaining copies according to some scheme (e.g., local copies first, followed by other "preferred host" copies, followed by all others);
- select a file to be copied from the refined list.

Based on the physical file selected for copying, the Resource Allocator calls one of the modules Localcopy, Pure Transmission, or Translation. Localcopy only calls LOSIMs and possibly MSGIMs. Pure Transmission calls LOSIMs and MSGIMs, but there may also be two modules to encode and decode the file according to some private host family protocol.

Translation consists of two modules:

- Encoder, which builds the header record and encodes the file; and
- Decoder, which sets up decoding parameters (by decoding the header record) and decodes the file.

In addition, LOSIMs and MSGIMs are called to read/write the file, send/receive data from the MSG direct connection, etc.

IV. Local OS Interface Modules (LOSIMs)

The following functions will be supported:

- Verify legality of a name of a physical copy
- Connect to a directory (i.e. establish a means of accessing a file)
- Create a unique file name
- Enter a name in the local catalog
- Open a file
- Read a machine-dependent unit of a file (e.g., block, page, track, etc.)
- Write a machine-dependent unit of a file
- Close a file
- Delete a file
- Read the catalog parameters of a file
- Send data over the direct connection
- Receive data from the direct connection

Clearly, not all FP implementations will have a separate module for each function, e.g. in TENEX, sending to/receiving from the direct connection is identical to writing to/reading from a file and can be done by the same module.

V. MSG Interface Modules (MSGIMs)

1. Signal Handler

This module will ensure that no two pending events have the same signal, and will tell the Dispatcher (section II.2) how to associate a signal with the location at which to start processing when the signal occurs.

2. Buffer Handler

This module creates a buffer of a specified length to hold a message or alarm.

3. Build Argument List

This module will encode argument/result lists in some manner (e.g., using PCPB8 data structures) and place the encodement in an assigned buffer.

4. Build Message/Alarm

This is used to set up an MSG argument list preparatory to executing a primitive.

5. Execute Primitive

This executes a specified MSG primitive. There may be separate modules for the different primitives, depending on the local MSG implementation.

6. Add/Delete Process Address

To aid in verification of the source of a communication passed by MSG and to aid in addressing a message or alarm to a given destination process, modules to maintain a list of addresses of processes with which the FP is actively engaged should be provided.

The above modules are in some sense speculative, since they depend heavily on the currently unknown local MSG implementations.

VI. Graphic Representation of the FP Structure

As an aid to understanding the FP structure as presented in sections II-V, we present an indented version of the structure, where the n-th level of indentation represents modules called by the immediately preceding (n-1)-st level module.

1. File Package (called by the WM or another FP)

    A. Initializer

    B. Dispatcher
        i. verify message/alarm source
        ii. save source process address
        iii. pass control to appropriate place

    C. Request Analyzer
        i. DELETE driver
            a. LOSIMs
            b. MSGIMs
        ii. ANALYZE driver
            a. LOSIMs
            b. MSGIMs
        iii. Resource Allocator
            a. Refine PC list
            b. Select PC
            c. Copy Controller
                (1) Localcopy
                (2) Pure Transmission
                    (A) Family encoder
                    (B) Family decoder
                (3) Translation
                    (A) Encoder
                        (i) create header
                        (ii) Encode file
                    (B) Decoder
                        (i) Set up decode parameters
                        (ii) Decode file

(Note: LOSIMs and MSGIMs can be called from all (A), (B), (i), (ii) level copy modules.)

Section 5: An Intermediate Language for File Transfers

I. Introduction

This chapter discusses a grammar to describe text and binary file transfers between NSW TBH's of differing families, using the NSW File Package (FP) as the transfer protocol. The sending FP would encode the file according to the grammar, and the receiving FP would parse it - i.e., it would recreate the file as close to the original as possible.

The main requirements of the grammar are two: first, it should be possible to encode files with a minimal loss of structural information; second, it should capture the structural information of the file system of any TBH. (As a corollary to the latter, it should be easily extendable to include new types of TBH's as they are brought into NSW.) Clearly, the fulfillment of the first requirement is dependent on the fulfillment of the second. In the first pass at defining the grammar, we have not attempted to capture all the structural complexity of, say, the OS/360 file system. Still, the grammar is quite rich and can be easily expanded. The main feature of the grammar which leads to ease of expansion is the separation between structural information and data. The structural information, which at the moment consists of byte size information, sequence number size, and format effector definitions, is contained in a *header* record. This information can be redefined in-stream by a *control* record. The header and control records are separate from the *data* records which contain the actual data of the file. Thus, structural information for a new TBH can be incorporated as new productions in the header and/or control records, leaving the rest of the grammar undisturbed.

In moving text files from paper tape-oriented machines to unit record-oriented machines, the inevitable problem of ASCII format effectors (e.g., horizontal and vertical tabs) arises. The grammar allows the definition of vertical format effectors and the spacing of horizontal tabs (and could easily be expanded to allow for defining more exotic horizontal format effectors). In this way, the receiving FP can build a "dictionary" of format effectors used by the sending (paper tape machine's) FP. When a format effector is encountered during the parsing of the encoded file, the "dictionary" is consulted and the decoder can take the action which is most appropriate for the host operating system and/or the tool which will use the file. For example, horizontal tabs may be expanded into blanks, vertical tabs into blank lines or blank lines with ASA format effectors as the first character, etc. Similarly on transmission from a unit record machine to a paper tape machine, leading blanks may be collapsed by the receiving FP into horizontal tabs, etc. The grammar makes no assumptions and imposes no restrictions on how

the file is to be stored at the receiving end; its purpose is to provide as much information as possible concerning how the sending FP stores the file.

The problem of moving text files with ASA format effectors to paper tape-oriented machines is not quite as severe. There are syntactic tokens (see section III) which enable the sending FP to say "begin the next record, skipping two (or three, or more)". Thus a 133-byte print line with a zero as the first byte would be encoded as "begin next record, skipping two" followed by an encoding of 132 bytes of data. The receiving FP could reproduce the effect by inserting the right number of carriage returns.

Moving binary files between TBH's of differing byte size presents a different problem. The header record allows the definition of the byte size (in bits) of the TBH upon which the file was created. Beyond that, any manner of transmission may be considered. For example, the file may be considered as an unstructured stream of bits, broken up into 8-bit bytes for purposes of transmission. Alternatively, a connection could be opened between the sending and receiving FP's, the width of which is the byte size of the file. The file may (as in the case of text files) be stored in any manner the receiving host chooses, but the byte size information should probably be saved somehow (perhaps in the NSW file catalog). This is an open question which, while it requires further discussion, does not immediately affect the grammar.

In the following section, the productions of the grammar are presented in groups, and the meaning of each group is discussed. Extended BNF is used. When a number is quoted ("255") it represents an 8-bit (i.e., the low-order 8 bits of a byte which is the same size as the width of the connection) control item (see section III). An unquoted number is an 8-bit integer (unless otherwise noted). Where a non-terminal of the grammar is followed by "(p:q)" it signifies the presence of from p to q copies of the non-terminal.

II. The File Transfer Grammar

1. <FILE-XFR> ::= <HEADER><TEXT-FILES>"251"

   An NSW file transfer is defined as a header (the structural information) followed by a sequence of text files (see subsection II.5), followed by an end-of-transfer byte.

```
2.  (a)  <HEADER> ::= "248"<HEADER-INFO>
    (b)  <HEADER-INFO> ::= 0<SEQ-DESC><TAB-DESC>
    (b')                 |1<BYTE-SIZE>
```

The header of a text file (2b) defines the characteristics of the sequence numbers and the horizontal and vertical format effectors. The binary file header (2b') describes only the byte size (in bits) of the computer originating the file.

```
3.  (a)  <BYTE-SIZE> ::= 8|9|...|255
    (b)  <SEQ-DESC>  ::= 0|1|...|8
```

The sequence number descriptor is the number of bytes in the sequence number (zero for unsequenced files). The sequence number may be "identification data" of some sort rather than a true sequence number, in which case the receiving host may choose to discard the sequence numbers if they do not meet the specification of sequence numbers for the tool requesting the file.

```
4.  (a)  <TAB-DESC> ::= <STOPS><VERT-DESC>
    (b)  <STOPS> ::= <NUM-OF-STOPS><STOP>(1:n)
    (b')           |<STOP-INCR>
    (c)  <NUM-OF-STOPS> ::= 0|1|...|127
    (d)  <STOP-INCR> ::= 128|...|247|249|...|255
    (e)  <STOP> ::= 0|1|...|255
    (f)  <VERT-DESC> ::= <VERTICAL>(0:m)
    (g)  <VERTICAL> ::= <CTL-CHAR><STOPS>
    (h)  <CTL-CHAR> ::= 1|...|8|10|...|31
```

The tab descriptor (4a) consists of the horizontal tab descriptor (4b, 4b') followed by a sequence (possibly null) of descriptions of vertical format effectors (4f). The horizontal tab character is understood to be 9 (decimal), but there can be any number (up to 30) different vertical format effectors. Production (4g) allows for the definition of these vertical format effectors by naming the character (4h) and its stops (4b, 4b').

The stops definitions work as follows: (4b) indicates the number of stops on a page (or line, in the case of horizontal tab) followed by that number of line (column) numbers describing the stop locations. If the stops are evenly spaced, (4b') indicates the stop increment plus 127, i.e., 135 implies stops every eight columns or lines.

```
5.   (a)  <TEXT-FILES> ::= <TEXT-FILE>
     (a')                 |<SUBFILES>
     (b)  <SUBFILES> ::= <SUBFILE>(1:p)
     (c)  <SUBFILE> ::= <TEXT-FILE>"255"
```

The file to be transferred may be a single file (5a) or a concatenated sequence of files (5a', 5b), each of which is a single file followed by an end-of-file byte (5c).

```
6.   (a)  <TEXT-FILE> ::= <RECORD>(1:q)
     (b)  <RECORD> ::= <DATA-REC>
     (b')              |<CTL-REC>
     (c)  <DATA-REC> ::= <REC-CTL><SEQ-NUM>(0:1)<ITEM>(1:s)
     (d)  <CTL-REC> ::= "254"<ctl-info>
```

A file is a sequence of one or more records (6a), each of which may be a data record (6b) or a control record (6b'). The latter (6d) is basically undefined at the moment, but may potentially be used to redefine information contained in the header. A data record (6c) consists of a record control byte (or bytes; see below), an optional sequence number (which must conform to the descriptor in the header), and a sequence of items. (If the header has indicated that sequence numbers of length *n* are present, the *n* bytes following the <REC-CTL> item are to be interpreted as the <SEQ-NUM>.)

```
7.   (a)  <ITEM> ::= <STRING>
     (a')             |<REPEAT>
     (a'')            |<FILL>
     (b)  <STRING> ::= <STR-LEN><CHAR>(0:r)
     (c)  <REPEAT> ::= <REP-LEN><CHAR>
     (d)  <STR-LEN> ::= "0"|"1"|...|"127"
     (e)  <FILL> ::= "128"|...|"191"
     (f)  <REP-LEN> ::= "192"|...|"223"
```

An item is a string (7a), a repeated character (7a'), or a fill character indicator (7a''). The string is a length factor followed by that number of characters (7b). The repeated character is a factor followed by the character (7c). The string length explicitly ranges from 0-127. The fill number k is expressed as 128+k, and the repeat factor k as 192+k. (The compression implied by these productions is optional, e.g., a string of blanks may be passed uncompressed if the implementor so desires.)

8. (a) <REC-CTL> ::= "224"|...|"239"|"249"|"250"
|"252"<N1>|"253"<N2>
(b) <CHAR> ::= any 8-bit pattern
(c) <N1> ::= unsigned 8-bit quantity
(d) <N2> ::= unsigned 16-bit quantity (formed by concatenating
#2# 8-bit bytes)

See section III of this chapter for a further description of (8a).

III. Syntactic Tokens

The following is a table of syntactic tokens which represent record and text control information. They do not represent control information in the header (see section II.2-4).

Bit Pattern	Value Range	Meaning
0xxxxxxx	0-127	String of length 0 <= n <= 127
10xxxxxx	128-191	Fill character repeated 0 <= n <= 63 times
110xxxxx	192-223	Repeat next char. 0 <= n <= 31 times
1110xxxx	224-239	Begin new record, skipping 0 <= n <= 15 records
11110xxx	240-247	(Reserved)
11111000	248	Begin header
11111001	249	Begin first record
11111010	250	Form feed
11111011	251	End file transfer
11111100	252	Begin new record, skipping 0-255 records
11111101	253	Begin new record, skipping 0-(2**16-1) records
11111110	254	Begin control record
11111111	255	Delimiter of subfiles

The above set of syntactic control items allow for an easy implementation of a generator and a parser for the grammar. Every control item is one byte long and indicates, explicitly or implicitly, the length of the following data item.

Section 6: Translation Semantics

File translation is a major problem which the File Package must address. When the FP is forced to translate a file, it must map a foreign representation of the file onto its "logical equivalent" in the local file system.

Definition of this equivalence seems beyond the current state of the art. One can, however, state some desired properties. Suppose identical line printers are connected to donor and receiver. Then receiver's copy is logically equivalent to the original only if their listings are visually identical. The same applies to decks of punched cards (both binary and text) which may be read or punched. The hope is that equivalence at this level will extend to tools. Indeed, most tools tend to deal with their text input files in units of lines, irrespective of file organization. Solving the listing problem is very close to solving the "line of text" problem which should provide the desired mapping of the output of one tool into the input of another.

What kind of difference in file systems causes this problem? It is that some file systems represent text files as "unit-records" and others as "paper-tapes". Sequential access to lines of text is available at the file system level in a unit-record machine. That is, a low level access method can easily produce the next line of text; that line will consist of only characters and blanks.

The situation is entirely different in a paper-tape machine. A low-level access method is used to read bytes from the file which must be interpreted in order to produce a line of text in the above sense. Besides characters and spaces, "format effectors" are stored in paper-tape files. Indeed the paper-tape file organization is a descendant of the paper-tape controlled typewriter. Format effectors may be grouped into horizontal ones (e.g., carriage return (CR), backspace (BS), and horizontal tab (HT)) and vertical ones (e.g., line feed (LF), form feed (FF), and vertical tab (VT)). Vertical effectors have no horizontal component, and vice-versa. Vertical motion is always down the page.

Those format effectors which cause horizontal motion to the left have no analog in unit record machines and a mapping onto a reasonable equivalent must be found. Of course, if such motion is paired with vertical motion no problem arises; in fact, sender is requested to treat CR-LF and LF-CR pairs as record delimiters in the intermediate language sense. The basic question is this: if the paper tape sequence "b BS /" is to become the two records:

```
(record)                b
(skip 0)                /
```

should the sequence "b BS BS a"

become

```
(record)                b
(skip 0)               a
```

or

```
(record)               ab          ?
```

For purposes of display the single record version is both correct and optimal, but for tool use, it should be left to FP and tool implementors to make the choice.

A brief discussion of each of the format effectors is in order:

HT - never poses a translation problem as it is equivalent to one or more blanks. However, hosts are free to choose their own tab-stop conventions; therefore, we have included a description of them in the file's HEADER record.

BS - has already been discussed.

CR - unless paired with a vertical format effector is equivalent to back-spacing to column 1.

LF - causes the beginning of a new record beginning at current column. The new record is blank to the left of current column.

FF - easily translates to record environments, but with the same column proviso attached to LF.

VT - is similar to HT. Stops are specified in HEADER record.

The following algorithm sketch may be useful. It describes a procedure by which consecutive characters from a paper tape are encoded into a compact record file - compact because "a BS BS b" produces only one record. It is called by the following control program:

```
CHARACTER PT(LTH)  /*PAPER TAPE*/
INTEGER LTH        /*LENGTH OF PT*/
```

```
/*CHARACTER ARRAY FOR BUILDING RECORD-EQUIVALENT OF PT*/

        CHARACTER C(MAXLRECL,LPP,MAXPAGES,MAXOVPRT)

/*PARAMETERS OF C*/

        INTEGER MAXLRECL,       /*MAX LENGTH OF RESULTANT RECORDS*/
                LPP,            /*LINES PER RESULTANT PAGE*/
                MAXPAGES,       /*PAGE CAPACITY OF C*/
                MAXOVPRT,       /*MAX OVERPRINT CHARACTERS
                                  PER CHARACTER POSITION*/
                BOTTOM,         /*BOTTOM MARGIN OF PAGE
                                  IN LINES (BOTTOM < LPP)*/
                HTSTOP,         /*HORIZONTAL TAB STOP,
                                  IN COLUMNS*/
                VTSTOP          /*VERTICAL TAB STOP,
                                  IN LINES*/

/*INDEX VARIABLES FOR ACCESSING THE ELEMENTS OF C*/

        INTEGER COL,            /*NUMBER OF COLUMN WITHIN LINE*/
                LINE,           /*NUMBER OF LINE WITHIN PAGE*/
                PAGE            /*NUMBER OF PAGE WITHIN C*/


/*PRE-BLANK THE ENTIRE CHARACTER ARRAY, C, AND INITIALIZE
  COLUMN, LINE, AND PAGE TO THE UPPER LEFT-HAND
  CORNER OF PAGE 1.*/

        C(*,*,*,*) = ' '

        COL  = 1
        LINE = 1
        PAGE = 1

        FOR I = 1 TO LTH DO
                CALL UREC(PT(I))
```

The procedure UREC will build the character array C whose slices C(*,LINE,PAGE,OVPRT) represent unit records which result from a compacting paper-tape conversion. For illustration, PT can contain TWO-LINES and THREE-LINES (skip to next multiple of 2 or 3 lines) as well as HALF-PAGE and THIRD-PAGE vertical format effectors.

```
        PROCEDURE UREC(CHAR):

        CHARACTER CHAR
```

```
        PROCEDURE FMFEED:
        BEGIN
                LINE = 1
                PAGE = PAGE + 1
        END FMFEED

/*PROCEDURE OF UREC TO ADVANCE TO A VERTICAL TAB STOP
  WHEN STOPS ARE EVERY "FRAME" LINES ON A PAGE OF "LPP" LINES.*/

        PROCEDURE ADVANCE(FRAME):
        BEGIN
                INTEGER NEWLINE, FRAME

                NEWLINE = FRAME*CEIL(LINE/FRAME)+1
                IF NEWLINE > LPP THEN CALL FMFEED
                        ELSE LINE = NEWLINE
        END ADVANCE

/*IS-GRAPHIC MEANS CHARACTER VISIBLY PRINTS*/
    IF IS-GRAPHIC(CHAR) THEN
        BEGIN
               IF COL > MAXLRECL THEN SIGNAL("LRECL EXCEEDED")
               IF C(COL, LINE, PAGE, MXOVPRT) NE ' ' THEN
                    SIGNAL("TOO MANY OVERPRINTS")
               OVPRT = 1
               UNTIL C(POS, LINE, PAGE, OVPRT) EQ ' '
                    DO OVPRT = OVPRT + 1
               C(COL, LINE, PAGE, OVPRT) = CHAR
               COL = COL + 1
        END

        ELSE

        CASE OF CHAR

        SPACE:          COL = COL + 1
        BKSP:           IF COL > 1 THEN COL = COL - 1
        CR:             COL = 1
        HTAB:           COL = HTSTOP*CEIL(COL/HTSTOP) + 1
        LF:             IF LINE > LPP - BOTTOM THEN CALL FMFEED
                            ELSE LINE = LINE + 1
        VTAB:           CALL ADVANCE(VTSTOP)
        HALF-PAGE:      CALL ADVANCE(LPP/2)
        THIRD-PAGE:     CALL ADVANCE(LPP/3)
        TWO-LINES:      CALL ADVANCE(2)
        THREE-LINES:    CALL ADVANCE(3)

        RETURN

        END UREC
```

Section 7: Glossary

DONOR:	A File Package instance which is the "slave" in a copy operation. The donor reads the file in the local operating system and sends it to the receiver File Package via an MSG direct connection.
FILE SPACE,NSW:	A collection of files to which only the Works Manager has access.
HOST FAMILY:	A group of NSW hosts running the same operating system.
LOGICAL COPY:	The copy (of a file) made by a receiver File Package. Its logical structure is equivalent to that of the physical copy sent by the donor, but the physical structures of the "original" and the logical copy may differ.
LOGICAL STRUCTURE, EQUIVALENT:	The logical structure of two text files are equivalent if the files appear to be the same when listed on a line printer. (No similar definition has been formulated for binary files.)
PHYSICAL COPY:	An NSW file name represents a set of physical copies which are logically equivalent. When it is created, a physical copy may be called a logical copy, to emphasize the logical equivalence between the newly created physical copy and the "original" physical copy.
PHYSICAL COPY NAME:	A list which contains the network address and host-dependent name and structure information of a physical copy. See Chapter 2 ("File Package Functions"), section II ("The Physical Copy Name").
PHYSICAL STRUCTURE:	The physical characteristics of a file, e.g., logical record length, blocking factor, number of (memory) pages, total number of bytes, etc.
RECEIVER:	A File Package instance which is the "master" in a copy operation. It is contacted by the Works Manager and in turn selects and contacts a donor File Package and drives the copy process.
WORKING COPY:	A physical copy of a file created in the work space of a tool.

Chapter 5: Foreman Specification

The Foreman: Providing the Program Execution Environment for the National Software Works

I. Introduction

1.1 Overview

The National Software Works (NSW) is a facility resident on the ARPANET, principally intended to support the construction of computer programs and to provide software tools (e.g., editors, compilers, debuggers, etc.) which can be used in the construction activity. A prominent factor in the conception of NSW is the expectation that the hardware, software, and human resources needed for the execution of a task may be geographically and administratively dispersed, although connected through the network. As such, the NSW is a distributed, multi-computer system.

The major components of the NSW are the ARPANET, a collection of tool-bearing hosts, one or more front-end (FE) systems through which the users access the NSW, and an access-granting, resource-allocating central component called a Works Manager (WM). A tool-bearing host (TBH) is a computer system which houses software development tools made available to users through the NSW, and which additionally may provide storage for NSW user files. To become a TBH, a host must implement a set of supervisory software modules. Included in this set are the modules which handle the NSW communication needs (see MSG: The Interprocess Communication Facility for the NSW, Bolt Beranek and Newman Inc. Report No. 3237 and Massachusetts Computer Associates Inc. Document No. CADD-7601-2611), the modules which handle NSW file transfer requirements (see File Package: The File Handling Facility for the NSW, Mass. Computer Associates Document No. CADD-7602-2011), and the modules which provide the local host functions for invoking and controlling NSW tools, as well as providing a local host NSW tool execution environment. These latter tool-oriented modules are collectively known as the Foreman component of the NSW, and are the subject of this document.

The Foreman specification is especially intended for those persons responsible for implementing TBH Foremen. In many cases, we are merely presenting approaches to problems which must be faced. While we do require that most (if not all) of the functionality we will specify be made available to the tools, the form in which it is presented to the tool is a purely local decision. The only strict requirement is that each Foreman must implement the various functions which can be invoked externally (see Appendix 1). The exact time and sequence for the Foreman itself invoking functions in the other NSW components (e.g., WM)

is discretionary. In most areas, this document also suggests implementation strategies which we feel are worthwhile. However, because of the discretionary nature of the Foreman/tool interface, aspects of this interface mentioned in this document should not be considered as final specification to the tool builder. Each Foreman implementer is responsible for providing the exact details of his tool interface through a host specific tool builder's guide. Potential tool builder's should contact NSW TBH personnel for the tool's implementation host before beginning to integrate their tool(s) into the NSW.

1.2 The Role of the Foreman

The Foreman is the local-to-the-tool component of the NSW system. Each instance of a tool has a Foreman which is responsible for the smooth operation of the tool with the other NSW components. In conjunction with the facilities supported by the Works Manager (WM) component, and to a lesser extent the Front End (FE) component, the Foreman provides the tool instance with its NSW execution environment. Every tool instance runs under the control of a Foreman. Components which do not run under a Foreman are considered part of the implementation of the NSW structure itself.

Note carefully the distinction between the concept of a "tool" and the concept of a "tool instance". A tool refers to a computer program, while a tool instance corresponds to the abstract notion of a process. A tool is a static concept in NSW, while a tool instance is dynamic. The difference is also reflected in the nature of the information maintained by the NSW about each. A tool has a static tool descriptor, maintained by the WM, which describes information pertinent to each use of the tool. Information pertaining to a tool instance is maintained in a dynamic data base, partly within the WM and partly within the Foreman. The dynamic entries referring to a tool instance are maintained for the duration of the tool session only. In this specification we shall often simply refer to "the tool" instead of "the tool instance", in order to avoid the constant repetition of the word instance. However, it should be clear from the context whether we are referring to the static concept of a tool as a program (hardly ever) or the dynamic concept of a tool instance as the execution of that program (almost always).

The Foreman has two well defined parallel interfaces, both of which are described in this specification. One interface is between the Works Manager processes and the Foreman. This interface is organized around the MSG message passing capability, and involves both the WM instructing the Foreman about handling the tool instance, and the Foreman requesting WM services on behalf of its tool. The responses to these commands/requests are obviously also part of the WM/Foreman message interface. The

other well defined interface is between the Foreman and its tool instance. The Foreman has a special relationship with the tool it is monitoring. In addition to having the responsibility for creating and subsequently removing the actual instance of the tool, the Foreman maintains an operating-system-like interface to the tool. It is through this interface that the tool can invoke the various functions provided by the NSW environment to augment the host operating system environment. The tool/Foreman interface can take any of a number of forms, the selection made by the implementer of the Foreman for a particular host. Examples of the various types of tool/Foreman linkage include subroutine call (as in MULTICS), operating system call (SVC in the IBM series, JSYS in TENEX), and short messages (ELF). A host-specific tool implementer's guide will detail the exact nature of the tool/Foreman linkage (for each system) as well as the exact nature of the NSW system calls available to tools on that host. In this document, we attempt to specify the functions which the Foreman is expected to implement, help implement, or provide access to. Some implementation details are given. In addition, we mention an optional Foreman function (encapsulation) which can greatly ease the task of bringing into the NSW selected tools which already exist as local host programs.

This document should be viewed as the first of a series of specifications of functions to be performed by the Foreman. The initial tools do not require sophisticated system support, nor has there been adequate time to fully investigate important areas such as error recovery. Because of this, and because of the phased implementation plan for the NSW, this document is vague in some functional areas, and incomplete in others. (It is clearly noted where we intended to have functions incompletely specified. Other instances of this are oversights, and should be immediately noted in any response to this specification.) Future revisions of the Foreman specification will clarify and further define these areas. In addition, as the NSW concept and the NSW system evolve, extensions to the capabilities supported by the Foreman can be expected. As a statement of intent, we feel that it is a sound design strategy to provide tools with mechanisms for doing (almost) all of the things a user at his terminal could do. The initial specification of the Foreman only partially reflects this goal.

It is important to emphasize that the NSW project is not involved in or based on writing new operating systems. Rather, we are building the NSW by allocating subsets of the resources of the participating machines, and using the existing operating system to help manipulate these resources (and at times transform them into resources more appropriate to the NSW environment). This approach is obvious upon examination of the abstract machine under which tool instances are run. The tool environment is a blend of the environment originally provided by the host operating system, augmented in selected areas with facilities

implemented by NSW components. Sometimes it will be necessary to mold existing local host facilities into new specifications. This is the case when we utilize an existing local file system capability to support access to files under the NSW system. Other times the NSW facilities will be completely new to a host environment, so that adopting NSW standards directly in the operating system is a possibility. Such may be the case with an MSG communication facility. Accordingly, the structure and flexibility of the existing operating system will greatly influence the structure of a Foreman component for any machine. In specifying the role of the Foreman, we have tried to be as independent as possible of the structure of any operating system and avoid reliance on particular features of any system. We hope an implementation is not only possible but reasonably efficient on a wide variety of operating systems.

1.3 Aspects of the Foreman

There are five aspects of the functioning Foreman which are of concern in this document. They are:

* providing for tool startup/control/termination
* providing the NSW runtime environment for tools written to function in the NSW
* (optionally) providing for encapsulation of programs written exclusively for the local host operating system by defining mappings from existing local operating system functions to NSW system functions
* providing for batch type tools
* providing mechanisms for debugging tools and recovering from errors and malfunctions

Additionally, each tool must be prevented from interfering with other tools and other processes running on the host operating system. Toward this end, we envision a protection domain surrounding the tool, and a temporary workspace for file manipulation during a tool session. In some implementations the host operating system may provide support for these requirements (e.g. separate work directories temporarily assigned to the tool for the duration of the tool session, and subsequently cleared and used by other tools). Where the host operating system does not provide such support, the Foremen themselves must assure tool separation and maintain boundaries between workspaces.

1.4 Short Scenario of Beginning an NSW Session

At this point, we provide a scenario of the beginning of a typical NSW session. It serves to illustrate the roles of the various NSW components and sets a proper context for the discussions to follow. The reader is assumed familiar with the documents referred to above, and in addition the document Works Manager Procedures, reproduced within Mass. Computer Associates document CADD-7603-0411 (also available separately as MCA document CADD-7603-0412).

In this scenario we assume that there exists an NSW process in the (local to the user) Front End machine which is receptive to an attention character on a terminal. We also assume that WM command interpretation is done within this FE process. Our scenario begins from the point where the user has a dedicated FE MSG process assigned to handle his terminal port.

The FE process starts by prompting the user for his login information. After accumulating the pertinent information, the FE process sends the data to a Works Manager process using the generic addressing facility of MSG. The WM receiving the login message will verify the login parameters and note the full name of the FE process servicing this user. (The return address on the message indicates the FE process name.) A specifically addressed message from the WM process to the FE process communicates the success or failure of the user login.

If we assume a successful login, then, when the user wants to run a tool, the FE gathers the name of the tool along with any other pertinent information and sends the request generically to any WM. The request is actually one in which the FE process is asking the WM to establish a new instance of the tool on behalf of the NSW user. The receiving WM verifies whether the user can indeed run such a tool, and, if so, retrieves the tool descriptor from an internal WM data base. Based on information accessible to it, the WM formulates a message containing the tool information and sends it generically to a Foreman process on the host which has been selected (by the WM) to run the tool instance. This information includes the nature of the tool (e.g. encapsulated, uses MSG directly, etc.) and the MSG process name of the FE process for this user. The Foreman selects a workspace for the tool and establishes the tool instance in this workspace. The Foreman then returns (to the WM which called it) the MSG name of the tool which was created. This is done using the specific send MSG capability. The tool name may be different from the Foreman MSG name in the case of non-encapsulated tools using MSG facilities. The WM then replies to the original calling FE process with a specifically addressed message which indicates the MSG addresses of the tool process and the Foreman process.

For the case where the tool and FE communicate via MSG messages, our scenario is complete since both the FE and tool/Foreman have each other's specific name and can send messages directly to each other. For the case where the tool is encapsulated, or where the tool requests to use a network Telnet connection to the FE, the Foreman sends a message to the FE indicating that a direct FE-Tool connection is needed. Using each other's specific names, the FE and tool/Foreman use MSG primitives to establish a direct communication path. As soon as the MSG connection requests match, and the connections are established, data can flow from the user to the tool and vice-versa.

1.5 Conventions Used in this Document

To avoid confusion arising from the ambiguous nature of the names of the various functions implemented within and for the Works Manager, the Foreman and the tool, we adopt a convention within this document to help the reader understand which component implements a given function. We are using a prefix of F$ to indicate an externally callable function implemented within the Foreman (e.g. F$BEGINTOOL), a prefix of W$ to indicate a function within the Works Manager (e.g. W$DELIVER) and all capitals with no $ prefix to indicate Foreman-implemented, tool-callable primitives (e.g. DELETELOCAL). The F$ and W$ prefixes are only used as expository aides, and are not part of the actual function name in either the Foreman or the WM. The actual function name is the string remaining after stripping off the F$ and W$, as appropriate. Interprocess request transmission conventions are currently those specified by Jon Postel (SRI) in his network message of 10 March, 1976. These requests utilize a modified PCPB8 format. However, this format is subject to further changes. The exact specification of the arguments to be sent to a Foreman and returned from a Foreman are compiled in Appendix 1 of this document.

II. Controlling the Execution of a Tool

2.1 Initiating an Instance of a Tool

The MSG facility itself is responsible for the allocation of Foreman processes in response to demand for their use. Thus we can begin our specification assuming the existence of the Foreman as an MSG process, and control lying within the Foreman. After initialization, a Foreman must be receptive to a F$BEGINTOOL message from a Works Manager process. The F$BEGINTOOL message is sent as a generic message addressed to a Foreman of the proper variety (i.e. host type), and can be received via the Receivegeneric capability supported by MSG. The Receivegeneric by the Foreman indicates that it is ready to support a new instance of some tool, on command from a Works Manager. The F$BEGINTOOL message contains a host specific name of the tool to be run. (The WM retrieves the host specific name for this tool from a static tool descriptor it maintains for each tool.) On the basis of this name, the Foreman is expected to be able to invoke an instance of the tool, while maintaining control over the running tool.

Prior to initiating the tool, the Foreman selects a workspace in which to run the tool. It then initializes this workspace, usually by clearing it of any remaining files. The set of Foremen processes on a TBH are responsible for managing the set of workspaces the TBH has for NSW tool support. The organization and utilization of the workspaces are left completely to the Foreman. However, the WM must be informed as to which workspace a tool has been assigned, so that it can initiate proper file movement into and out of the tool workspace. The host specific parameters describing the workspace (e.g. in TENEX they are a directory name and (if necessary) a password) are returned to the WM as results of the F$BEGINTOOL. If there is currently no available workspace, then the WM request must be rejected. The WM maintains lists of the TBH workspaces which are running tools, and these play an important role in helping a Foreman recover from system crashes without losing user files left in the workspace.

The F$BEGINTOOL message includes flags indicating whether or not to start execution of the tool and whether or not the tool knows about the NSW (new tool/old tool). The message also includes the name of the process on whose behalf the tool was created (usually the FE representing the user), and the process which serves as the FE to the user (if different from the above). An option of the F$BEGINTOOL message is the inclusion of a list of local to the tool file names which are to be directly accessible to the tool. These files are non-NSW files to which the tool should be allowed direct access without any intervention from the Foreman.

2.2 Removing an Instance of a Tool

Throughout the tool session, the Foreman must be receptive to an F$ENDTOOL specifically addressed MSG message from any Works Manager process. The message will contain a reason for ending the tool session. In all cases, the Foreman will return to the caller the cost incurred by the tool which has been run, after the actual termination of the tool. The Foreman will terminate itself after responding to the F$ENDTOOL request, and the association between the tool/Foreman and the NSW system will be broken.

To help Foremen be receptive to the F$ENDTOOL request, the WM will precede a F$ENDTOOL request with an MSG alarm of code=1. Receipt of this alarm code will signal the Foreman of the forthcoming F$ENDTOOL message. Once a Foreman has received an alarm with code=1, it is expected to immediately begin processing its incoming messages, discarding all except the F$ENDTOOL request. Once a Foreman begins processing a F$ENDTOOL message it need not be receptive to any further messages or alarms. A F$ENDTOOL message which was not preceeded by an alarm with code=1 should be processed nonetheless.

The accounting information returned in response to an F$ENDTOOL is a list of accounting data items. The first entry in the list is an integer which is the cost in cents of running the tool for this session. The remaining items in the list are host specific measures of various resources consumed by the tool. Each TBH implementation registers with the NSW which resource measurements it takes and will return to the WM.

(Note: at some future time, it may be advantageous on certain systems to simply halt the tool instead of terminating it, with the possibility of eventually re-initializing the tool as a different instance of the same tool. If such a mode of operation were possible, and if the WM kept track of the tools assigned to each Foreman, then startup costs for tool invocation might be reduced by selecting a Foreman with the appropriate tool already in place. Since this may not be relevant to all hosts, we recommend that part of the F$ENDTOOL message specify whether or not to try to maintain the tool in the inactive state, and that the reply to F$ENDTOOL correspondingly indicate whether or not this was done.)

2.3 External Control of Execution

If possible, the Foreman should be able to handle F$STARTTOOL and F$STOPTOOL messages. These are used to externally control the progress of the tool through its algorithm. F$STARTTOOL can be used to initially start the tool

in cases where F$BEGINTOOL did not specify immediate startup. A minimal start/stop facility would provide for suspending the execution of the tool while maintaining its complete current state. The tool would be subsequently continued from the point it was stopped, or it would be aborted. If the tool can be stopped, then it must at least be able to be started (continued) again. The suspending and resuming of tools on certain hosts may not be possible because the host operating system does not support such behavior. In these cases, tool execution will automatically begin with the F$BEGINTOOL message, and F$STARTTOOL and F$STOPTOOL messages will be rejected.

A more extensive start/stop facility encompassing stopping as well as starting through an entry vector is often desirable, and we have made provision for this at the WM command language level. The implementation of this extension is highly desirable where it is possible. We specify the characteristics of the richest facility. It is the responsibility of the Foreman implementation to either reject requests for which it has no mechanism, or to map the request into an alternative well documented in the tool builders guide. The facility the WM supports builds upon the (required of all implementations) F$BEGINTOOL and F$ENDTOOL messages. These requests initially create the tool instance and ultimately (forcefully) cause the tool to immediately cease its existence. These requests roughly translate to the user saying to the WM "runtool" and "aborttool". During the execution of a tool, the user can of course request that the WM stop the tool. To do this the WM sends to the appropriate Foreman a F$STOPTOOL message, indicating the reason for stopping. [Note that all messages originating in the WM and destined for a Foreman of an existing tool are sent using the Sendspecific MSG facility, and can be received using the Receivespecific MSG facility. The responses (if any) are sent directly to the WM which made the request. Requests emanating from the Foreman, however, are sent generically to any WM, with the reply obtained as a message specifically addressed to the appropriate Foreman.] During a period of tool inactivity due to the effect of a F$STOPTOOL, the Foreman still must process MSG messages concerning the tool. It must always be able to process an F$ENDTOOL message, and may have to process any replies to outstanding requests made to other NSW processes (e.g. a request to the WM to retrieve a copy of an NSW file). However, any results to be returned to the tool may have to be queued awaiting the command to continue tool execution.

2.4 Entry Vectors

In a complete Foreman implementation, a stopped tool can be started in a number of ways. We introduce the notion of entry vector to unify the start tool concepts. The WM recognizes several standard entry points for its tools. These are an

initial entry point (cold start), a standard re-entry point (warm start), a termination entry point, and a continue from where stopped entry point. (Other system wide entry points will be defined as needed. It may also be possible for tools to implement private entry points invocable via the WM command language.) The meaning of all of the standard entry points is obvious, except perhaps for termination. The intent of the termination entry point is to allow a user to force control to be passed to a final tool cleanup routine, which may also involve saving some of the work of the session. Tools will normally implement commands to do this without WM intervention. However, circumstances may exist (e.g. runaway tool, depletion of resources) where it is useful to force an orderly termination while circumventing the normal tool dispatching. It is expected that the termination sequence implemented by the tool will complete reasonably quickly. If the tool has not signalled its Foreman that tool processing is complete within some host specified time frame, then the Foreman has the right and obligation to forcibly abort the tool execution without further notice.

When installing a tool in the NSW, the tool supplier indicates which entry point functions are available in the tool. This information is maintained by the WM in the interactive tool descriptor, and is used to regulate the type of F$STARTTOOL requests that are sent to the Foreman. In the WM tables the various entry points are statically denoted by a small integer (index). The standard entry points are denoted by the same index for each tool. It is entirely the responsibility of the Foreman and its host operating system to devise a method of converting these indices into actual program entry points (e.g. program counters) for executing the proper function. A Foreman implementation can impose coding standards for these purposes on the NSW tools which want to support the entry point concept.

N.B. When using any form of start/stop facility for tool control, it is the responsibility of the WM command language interpreter to insure the correct sequencing of the start and stop requests. Since in general MSG makes no assurances regarding the order of message delivery, and since WM command language requests are sent generically to any WM, it is most convenient to enforce sequencing at the command language interpreter. All this really means is that to ensure correct behavior, individual tool start and stop requests should not be allowed to be pending simultaneously. A final note on tool control is also in order. The concept of tools controling other tools, as contrasted to the NSW user controling a tool through the WM command language, is currently being defined. We envision the same set of Foreman procedures to be used in tool-tool control. This is discussed further in a subsequent section of this document. However, detailed elaboration of these concepts is postponed to a later date.

2.5 Detailing the Functions

Foreman procedures for tool control: (called by any WM process) The reply to each function invocation includes a "result" variable, which indicates if the operation was successful, and if not supplies a code indicating the reason.

F$BEGINTOOL (program-name, tool-type, entvec, FE-addr, cr-addr, filename-list) -> result, qstart, workspace-descriptor, tool-addr

A request of this type brings the tool instance to life. Program-name is a character string naming the program which forms the body of the tool. Tool-type is a variable indicating the nature of the program as an NSW tool (values defined below). Entvec indicates whether or not to start execution of the tool, and if started, through which entry point. FE-addr and cr-addr are MSG process addresses of the front end process and the creating process respectively. Filename-list is an optionally specified list of non-NSW local files (in the local host syntax) to which the tool is allowed unrestricted access. Some Foreman implementations may not care to protect the access to non-NSW local files. Tools running under such a Foreman would not need a file access list passed to the Foreman at tool initialization time. Tools which do not use non-NSW files would also not need a file list recorded with the WM for the tool initialization procedure. Qstart indicates whether or not the tool execution was begun. A workspace-descriptor is returned to the WM to allow file movement into the tool workspace. Tool-addr is the MSG address of the tool, which is often the same address as the Foreman (see Appendix 2).

F$STARTTOOL (entvec) -> result

The entvec variable indicates how the tool is to be placed back in the executing state: either to be continued from where it was last stopped, or through a specified entry location.

F$STOPTOOL (entvec) -> result

When a WM invokes this function the Foreman stops the execution of the tool and saves its state. We have provided entvec as an argument as a convenience in performing the dual operation of first stopping execution, and then commencing execution elsewhere through an entry point. Attempting to start an already executing tool, or stopping an already stopped tool will elicit an error condition.

F$ENDTOOL (reason, termtype, qmaintain) -> result,
accounting-list, qmaintained

Reason is a code indicating why the tool instance is being removed. Termtype indicates the type of file processing required of the Foreman before completing the F$ENDTOOL. This will be further clarified in later parts of the document. Qmaintain is a boolean indicating whether or not the Foreman should attempt to maintain the tool image for use as another instance. Qmaintained is another boolean which the Foreman uses to communicate to the WM whether or not the image has been maintained. Accounting-list indicates the resource cost and utilization for the tool session.

tool-type: value is an index
=1 -> encapsulated tool
=2 -> tool uses NSW calls, does not use MSG
=3 -> tool uses NSW calls, also use MSG facilities

entvec: value is either empty or an index
empty -> do not start tool (if possible)
=0 -> continue from point where stopped (illegal in F$BEGINTOOL)
=1 -> cold start entry point
=2 -> warm start re-entry point
=3 -> termination routine entry point
=4 -> reserved for expansion
=5 -> reserved for expansion
=6 ... -> tool specific entry points

2.6 Voluntary Tool Termination

As mentioned above, the Foreman must provide its tool with a primitive operation for indicating that the tool has completed execution. The HALTME primitive is the means by which a tool voluntarily relinquishes control for the final time. The Foreman may yet have to save files for the tool (see subsequent sections on the file system and encapsulation) before actually removing the job from the NSW domain. Through the termtype parameter of HALTME, the tool can indicate the type of Foreman file processing it expects. The current choices are:

* termtype=1 -> no Foreman file processing
* termtype=2 -> Foreman asks user which files need saving and saves them
* termtype=3 -> Foreman automatically saves latest copy of modified files

After all peripheral operations by the Foreman are complete, the Foreman notifies the WM of the tool completion by calling the WM W$TOOLHALT procedure. The associated parameters of the WM request include the accounting data list describing the tools resource utilization. The tool can be terminated (in the local host sense) any time after it issues the HALTME primitive. The Foreman terminates itself (in the MSG sense) after receiving the response from the WM to its TOOL-END call. A positive response indicates that the association between the tool/Foreman and the NSW has been broken.

Tool primitive operation:

HALTME (termtype) -> never returns to tool

WM procedure for HALTME support:

W$TOOLHALT (reason, accounting list) -> result

III. Status Probes and Resource Utilization Bounds

The Works Manager, as well as the NSW user through his FE process, may at times wish to closely monitor a tool execution in terms of resource utilization and progress through its algorithm. Toward this end, we specify two functional aspects of a Foreman implementation which can be used to achieve a degree of "tool watching".

Each Foreman must implement an externally invocable function (e.g. requested by a WM process) for probing the status of the tool currently being executed. We have taken the approach of allowing many types of probes. Invoking a status probe is different from invoking almost all other Foreman functions in that it is not done using an MSG message. Rather a status probe takes the form of an MSG alarm signal. The code transmitted as part of the alarm indicates the type of probe. In response to a probe alarm, the Foreman is expected to gather the requisite values and send them via an MSG message (which itself requires no reply) to the invoking process. Status probes are assigned Foreman alarm codes with values between 10 and 20 (octal). Two probe types are initially identified here, with others added as their need arises. One initial probe (alarm code 10) queries the current state of the tool as a program in execution. In response to a state probe, the Foreman returns the tool's current external NSW state (e.g. running, stopped, running at termination code, etc.), the tool's current internal NSW state (e.g. executing, waiting NSW primitive completion, etc.), and the tool's current local operating system state (e.g. running, blocked for I/O, dismissed, etc., and its program counter). The second initially defined probe (alarm code 11) queries the current state of the tool resource utilization for the session. It returns the accounting list referred to earlier. This includes the cost of running the tool so far, and other resource utilization measures maintained by the host operating system.

Currently there are no immediate implementation plans involving these probes other than as an indicator that the host/Foreman/tool complex is still functioning. At this point it is unclear which processes should be allowed to invoke which status probes, and we may have to specify an (as yet undefined) option for results to be prepared for a human rather than for program processing. Additionally, more thought needs to be applied toward determining a useful set of measures in each probe class. Because of these uncertainties, and because the "still working" function can be fulfilled by responding with an "unimplemented function" response, we postpone the exact specification of the values to be returned in a status probe.

A Foreman should also support the enforcement of resource utilization bounds on the tool it is running. These bounds would specify an approximation to the maximum use of a particular

resource or a measure of total resource consumption to which a tool session is limited. Exceeding these bounds would force a cessation of tool execution (i.e. placing the tool in a stopped state) along with the Foreman immediately notifying the Works Manager of the excessive resource utilization. The WM could then apply more resources to the tool session, or cause the tool to execute at its termination sequence, or abort the tool altogether. The initial bounds are passed as part of the F$BEGINTOOL request. We mean these resource bounds to be approximate monitors, and we have no interest in requiring very close scrutiny of the tool execution in order to shut off service the instant the tool exceeds a bound. On the other hand, we would require that the tool be monitored as closely as is reasonable to ensure that it does not consume an arbitrarily large piece of the specified resources (e.g. cpu time).

The WM procedures to support these bounds and the strategies for their use have not as yet been defined. The management tools in the NSW will surely have an impact on the nature of the facility which is actually used. Thus, as with the status probe and other features yet to be discussed, care should be taken not to exclude such behavior, but implementation is not required or yet possible. Future documents will detail the specification of these functions, and we solicit suggestions on their form and use.

Foreman Function:
(invocable via an alarm with code in the range 10-20 octal)

F$STATUSPROBE (type) -> status item list

IV. NSW Runtime Environment

The NSW tool environment differs in a few key areas from the environment provided by the host operating system. The NSW has its own means for inter-component communication and for dynamically creating NSW entities, and maintains its own file system. These facilities are in addition to any similar facilities which the host operating system may already provide, and may be used simultaneously if there is no conflict with providing NSW services. The Foreman and other NSW components provide access to the NSW facilities through enhancements to the set of primitive operations available to tools. There are primitive operations for dealing with each of the areas mentioned:

* NSW file system
* NSW process communication
* NSW process creation

These are discussed individually. The common thread is that the operations are provided in operating system-like fashion to tools, with the exact means of invoking a function or obtaining its result/status dependent on the host implementation. We are concerned nere with the semantics associated with the primitive calls, and the WM-Foreman message exchanges used in implementing these semantics. It is the Works Manager which actually supports the substance of much of the NSW environment. It is the Foreman which provides the local interface to the NSW facilities.

The functionality of each of the specified primitives must be made available to all tools. However, the exact form of the Foreman calls and returns, and the exact nature of the Foreman tool interface is left to the discretion of the Foreman implementer. Thus where it is deemed desirable, certain calls may be subsumed by a parameterization of other calls, or calls may be coupled to perform multiple functions. These are local optimization issues. We are concerned only that it be possible for each tool to somehow perform all of the operations which we feel to be important in the NSW context.

V. NSW File System

5.1 Two File Spaces

A tool running under the NSW system can independently manipulate items in two distinct file spaces. One file space is the sharable NSW global file space managed by the Works Manager and maintained independently of any tools that manipulate the files. The other space is the non-sharable, temporary workspace (local file space) for the copies of the files in use by a tool during the current tool session. A file entered in the central catalog must have a unique global name, and hence is able to be referenced (though perhaps not accessed) by any tool. A file which exists in the workspace for a tool can be referenced only by the tool operating in that workspace, and the mere existence of such a file may be unknown to other tools and even to the WM. It may also be the case that the name of a file in the workspace is not unique in the NSW file system. There is no conflict as far as the WM is concerned since the workspace file is unknown outside of the tool domain, and the tool itself is provided with a means for resolving any local name conflicts. Explicit name conflict resolution must occur whenever a file is to be entered into the global NSW file space.

5.2 Using Local Workspace

There is a considerable cost associated with inserting a file into the global filespace. The cost of synchronizing local activity with the global directory includes name conflict resolution, file copy (possibly network copy) and delay associated with synchronizing the WM and Foreman. Insertion into the local workspace is immediate. The same cost relationship holds when retrieving files for use by the tool. Therefore, it is often good strategy to build tools which utilize the workspace for storing and retrieving as much as possible, often waiting either until it is explicitly desired to synchronize the file use with other tools or until the end of the tool session before delivering selected files. Delivering files to the global file space at the end of a tool session means that only files which actually need to be permanently saved invoke the large system overhead, while files which do not require permanent name status (e.g. files which are subsequently deleted during the course of a tool session, or files which are only intermediate versions of a particular file) incur a minimal overhead. Savings can also be achieved by delivering multiple files in a single WM request, an obvious optimization if files are batched locally.

5.3 Version Numbers

Another aspect of the local workspace is the automatic use of version numbers to distinguish files of the same name. In essence, version numbering adds another field to "local" file names. The Foreman knows about the use of this part of the name field and supports certain default options for it. A version number is a small integer which gets bumped automatically when creating a file with an already existing (local name space) name and a version number is not otherwise specified. The use of version numbers is not part of the global NSW file space. Therefore the user must disambiguate name conflicts when files are moved from local space to NSW global space. In addition to providing automatic local disambiguation through version numbering, the Foreman must allow a tool to specify version numbers when referencing files in local space, as well as provide reasonable defaults for obtaining latest local copy and creating a newest copy in the absence of specified version numbers. If local files are cached for delivery (highly recommended) the Foreman should provide a means for the user to select from among the "new" NSW files only the ones he actually wants preserved. Suitable defaults are required in the absence of this information (e.g. the highest version number of each different file name is delivered at the end of the tool session).

5.4 Maintaining an LND

In the course of implementing the local workspace concept, the Foreman is required to maintain a local name dictionary (LND) for the tool it is running. The LND is used to specify the relationship between the NSW file name and the name of the file (in local operating system terms) which represents the local copy of that NSW file. Other information about the files which are created and maintained during a tool session is also appropriate for the LND. This includes information about version numbers, indications of whether a file has been modified since the copy was obtained (and therefore may need to be delivered), and short abbreviated strings which the NSW user or tool uses to refer to the NSW file. Some of the primitive operations provided to the tool are expressly for the purpose of manipulating the contents of the LND, and hence indirectly manipulate the local workspace. It is imperative that the LND for each tool instance be kept in a "crash-proof" manner (or as near to this as is reasonable and possible), so as to make feasible a recovery procedure in the event of a host system crash. Maintaining the LND in a carefully maintained and identifiable file on the local file system is one technique for achieving this. After a system crash which is not so catastrophic as to destroy the file system also, it would be possible to run a scavenger program in the tool workspace to retrieve the appropriate LND and save some of the results of the tool session. When the host again becomes available, it may also

be possible to re-start (or continue) the use of the tool in the same workspace and working with the saved LND. These issues are further discussed later in this document. In any event, the Foreman should attempt to insure that the effect of the host system crash is no worse for the NSW tool user than for the direct (non-NSW) users of that system.

5.5 File Names in NSW

The general syntax and semantics associated with file names in the NSW are described elsewhere. Here we are concerned with the impact of the Foreman and local workspace concepts on the use of NSW file names. The impact is two-fold: first in the conventions used by tools in providing names as parameters for file system operations, and second as extensions to the name syntax to provide for the manipulation of files in the local workspace.

To discuss file names as parameters to file system operations we must first describe certain aspects of the central NSW filing system as implemented via Works Manager procedures. Generally, the WM file system procedure for retrieving a copy of a file requires only a partial name (filespec) to specify the NSW file for the operation. Specifically, only enough of the name need be specified to disambiguate it. Thus we have the concept of files being retrieved (and saved) using abbreviated names. For example, WALDO.AUTHOR.TEXT might be addressed simply as WALDO. Additionally, when a file name provided to the WM is ambiguous (for retrieval) or already exists (for delivery), the WM often negotiates directly with the user to clear up any uncertainties. This is done only when requested by the tool. Because of these features, most calls involving file names return values which indicate the full NSW name of the file which was actually operated on, as well as any change in the filespec for the file as determined from the interaction with the user. (If the user dialog results in a new filespec, the user will presumably use this new name in future references to the file.) It is a Foreman responsibility to keep an up to date LND reflecting the latest information about NSW file names, as well as to make these names available to the tools which may be unaware of the change or clarification from the user. The names can be provided to the tools either by returning their values directly as part of the tool file operation, or (recommended) by providing functions by which tools can retrieve the names when they are needed. Since the names are kept in the LND anyway, such an operation is rather simple.

To insure that a tool can achieve a maximum utility out of the separation of global and local file spaces and the properties of both, the file manipulation primitives each imply an explicit domain (i.e. global or local space). This allows the tool to

directly control the spaces individually in the manner most appropriate to its particular purpose. Primitives dealing with workspace files also provide for the explicit selection of a particular version of a file, to enable a tool to override any default assumptions. Supporting a reasonable set of default parameters is encouraged, provided the defaults can be overriden where appropriate.

5.6 File Semaphores

Associated with each file in the NSW (global) file space is a semaphore. This semaphore can be set by a tool on behalf of a user (who intends and is able to modify the file) in order to warn other potential users that the file may be undergoing change. In general, this semaphore serves as a loose lock- the NSW system will only warn other users, not restrict their access. Under some circumstances, however, users will be prevented from accessing a file with the semaphore set. Note that the general looseness of the lock does not cause a multiple writers problem. NSW files are not directly modified by tools. Only local workspace copies of NSW files are directly modified. The NSW file is not modified until it is explicitly replaced. Thus a user can obtain an internally consistent copy of an NSW file with semaphore set, since it is only a workspace copy of that NSW file which is actually undergoing modification. The user is warned by the set semaphore that the modification is taking place (by some other user), and that at some time in the future, the NSW file may be replaced by an altered, updated version.

Tools may be divided into two classes: tools which explicitly use semaphores and tools which do not. In the first class go tools which note when a user performs a file modifying action and which are then willing to request the setting of a semaphore. Tools of this class may request that a semaphore be set when a copy is obtained of an NSW file. If this request is made, then the Works Manager presumes that the tool does not want access unless the semaphore can be set. If it is already set (by some other user) then the access request is blocked. In all other cases of copy access, the requester of a file is merely informed that the semaphore is set (and by whom- project and node-name). Delete access is always blocked by a set semaphore.

Whenever a tool of the first class (explicit semaphore users) requests a file, it may also request the setting of the semaphore. As noted above, inability to set a semaphore blocks the access request. Subsequently, a tool of this class may request the setting of the semaphore for a previously obtained file. The semaphore may be read or unset at any time, either by the tool or directly (via a WM command) by the user. If the semaphore has been explicitly requested, then it may remain set after the end of the use of the tool.

Whenever a tool of the second class (non-semaphore user) requests a file, the WM automatically tries to set the semaphore. Failure to do so does not block the access; the user is merely warned that the semaphore is already set. Again, the semaphore may be read or unset at any time. The semaphore is automatically unset when use of the tool is ended.

In all cases, a semaphore is unset whenever an NSW file is replaced (or, obviously, deleted). The tool primitives to be implemented for interfacing to the semaphore procedures of the WM are listed below. See the Works Manager Procedures document for details of the implementation within the WM.

SETSEMAPHORE (filespec, qhelp) -> result, NSW filename
UNSETSEMAPHORE (filespec, qhelp) -> result, NSW filename
READSEMAPHORE (filespec, qhelp) -> result, NSW filename

5.7 File Manipulation Primitives

A tool is provided with distinct sets of primitive operations to individually manipulate the NSW global filespace and the local workspace. An additional set of operations is provided for moving file copies between the two spaces. In this section, we introduce the major file manipulation primitives in each of the three sets.

A tool is provided with primitives for deleting, renaming and copying files within the global NSW namespace. The WM implements procedures which actually perform the DELETEGLOBAL, RENAMEGLOBAL and COPYGLOBAL operations within the global space, so these primitives are merely a packaging operation for calls on those procedures.

Two tool primitives (GET and PUT) are provided which are normally used to relate the files spaces in an automatic fashion. These provide for obtaining a local workspace copy of a global space NSW file (GET), and for depositing a local workspace file as a global space NSW file (PUT).

To support local workspace file access, the Foreman provides primitives for deleting, renaming and copying workspace files, and OPEN and CLOSE primitive operations. DELETELOCAL, RENAMELOCAL, and COPYLOCAL are all implemented totally within the Foreman, and merely result in changes to the LND. The OPEN primitive is used, with appropriate parameters, to gain access to a local workspace file, or to create a new workspace file in cases where one does not already exist. The CLOSE primitive is used to signal the Foreman that the file access is complete, and that the Foreman should assume responsibility for that version of the file. It is the intent that OPEN should prepare a local

workspace file for direct access and modification by the tool, and generally to return to the tool a handle on the file for such access. The type of handle, as well as the types of file data manipulations which are permitted, are local host operating system dependent, based on the file system which underlies the workspace implementation. One aspect of the CLOSE primitive is the invalidation of such a handle so that the Foreman can maintain a consistent copy of the file (via the LND) for possible introduction into the global file catalog. After executing a CLOSE operation on a file, the file data is not accessible to the tool unless it executes another OPEN.

As a general scenario, a tool GETs a local workspace copy of the global space NSW file it wishes to access. The workspace OPEN provides for direct access to the file data. The file actually accessed is always a local workspace file. It may be a copy of a global space file, or it may be a newly created, currently empty local workspace file, or it may be a previously referenced local workspace file which was originally one of the above. We assume that the local host system provides the actual data manipulation primitives for local workspace files. The tool ultimately CLOSEs the file and may subsequently repeat the OPEN-CLOSE sequence some number of times during the tool session. These operations affect only local workspace file images, and new versions of the original file copy may be created. Ultimately the tool decides that some version(s) of the workspace file should be placed into the global catalog, and it instructs the Foreman to do so using the PUT operation. Although these operations give the tool complete control over all phases of its file system interactions, Foreman implementations can and perhaps should often provide simplified means for performing common file functions. As examples, we could consider a Foreman which provides a GET which has the option of automatically opening the retrieved copy of the file, or an OPEN which searches the local space and if unsuccessful tries to GET a copy of the file, or automatic PUTting of the highest new version of each different workspace file upon tool termination. Any such local options will be clearly noted in the host specific tool builders guide.

5.8 Specification of the File System Primitives

In specifying the parameters of the main file manipulation primitives, there are many common arguments. Some are described here as a general introduction to the primitive descriptions.

NSW-filename: The NSW-filename is the full identification of a file in the NSW file system. This is generally a rather long string of text. However, a user will never have to type in a full filename. Instead, he will use either a "filespec" or an "entry-name" (defined below) depending on the intended use of the file. A full NSW-filename consists of two parts: the name

part and the attribute part, separated by a slash (/). The name part is a sequence of name components, separated by periods (.). The attribute part is a list of attributes separated by semi-colons (;). (For a more complete description of NSW-filename and attributes, see the document Works Manager Procedures.)

filespec: This is basically the supplied identifier for an NSW filename. It is an abbreviated form of an NSW-filename, used in contexts where the name of an existing file is required. A filespec need contain only enough parts of the NSW-filename to unambiguously denote the file. Unless changed by the tool, a workspace file copy and all its derivative versions will be referred to for the duration of the tool session by the filespec. A filespec may also contain file attributes as part of the name. (For a more complete description of filespec, see the document Works Manager Procedures.)

entry-name: An entry-name is an abbreviated form of an NSW-filename used in contexts where a new filename is to be created. As described in Works Manager Procedures, the contents of the user's enter scope is prefixed to the entry-name by the WM when a file is delivered. Aside from this abbreviation, however, the user (or tool) must specify the entire name component of the file. (For a more complete description of entry-name, see the document Works Manager Procedures.)

version #: When the local workspace is searched, the version number parameter guides the selection of a particular instance of the files associated with the filespec. Version number is either null (default) or is a decimal integer in the range 1 to 32. Special indicators exist for referencing the highest version, one more than the current highest version, and the lowest version of a filespec. The default value of version number is different for the various primitives and is given along with each primitive description. In general though, defaulted version numbers use the highest version for file access and creates a new highest version for file deposit. When searching the global space, any version number information is ignored.

qhelp: This argument has three possible values and conveys to the WM how the tool would like filename conflicts handled. It is used in conjunction with the various file system primitives. The possible values and their meanings are:

* qhelp=0 -> allow the user to supply help, through a help call on his FE process

* qhelp=1 ->allow the tool to provide help through a help call back to the tool

* qhelp=2 -> do not provide any help but instead report a failure on filename conflict, indicating this as the reason

The qhelp parameter applies only to calls involving the WM and the global NSW file space. Primitives dealing with workspace files need not support these help notions. Version number defaulting provides a form of automatic disambiguation for local space files. The Foreman must not allow the existence of different workspace files named with the same filespec (although different versions of the same file are obviously permitted). The WM interactive tool descriptor provides for a static default of the qhelp parameter, if the tool builder so desires. For these tools, the value of qhelp is obtained (each time) from the WM tables, regardless of the value passed by the Foreman. This is especially useful for encapsulated tools.

qreplace: This argument is a boolean, and it indicates whether or not a file being placed in the global space should force replacement of an existing file of the same NSW name. Qreplace with value true means that such replacement should be automatic, with the old file no longer accessible. Qreplace with value false means that replacement is not automatic and that the WM should revert to the qhelp variable to determine how to deal with the conflict.

success/failure code: An integer value representing the success/failure code for the operation is always returned as a result of each primitive. Each individual primitive has associated with it a set of interpretations of these integers. This code is always returned as a primitive result, but it will not be explicitly shown as a return value in the following primitive descriptions.

The description following each primitive operation reflects the nature of the operation as seen by the tool. In this section we limit ourselves to a description of the tool primitive (i.e. what does the primitive do?), leaving implementation considerations for the next section.

The tool has unrestricted access to all of the files within its workspace. Once a copy of a global space file has been obtained, all references to the local copy are permitted. The global space, however, is tightly controlled, based on the capabilities of the user and the tool he is using. Each operation involving a global NSW file undergoes strict access checking by the WM before it can be accepted. This is not of concern to the Foreman implementation, since the WM provides all of the access checks. The types of checks are mentioned here only to completely describe the primitives that the tool uses.

The WM file system and its access controls are described in the Works Manager Procedures document.

5.8.1 Global NSW Filespace Primitives

1. DELETEGLOBAL (filespec, qhelp) -> NSW-filename

This is the primitive a tool uses to delete an existing file from the global NSW file space. Global space deletion cannot take place until the WM verifies that filespec designates a unique file to which the user has delete access. This access is blocked by a set semaphore. Assistance is obtained as indicated by qhelp. Once a file has been deleted, it will no longer be accessible for GET, COPYGLOBAL, RENAMEGLOBAL, etc. The actual full NSW filename of the deleted file is returned to the tool after a successful DELETEGLOBAL operation.

2. RENAMEGLOBAL (filespec, entry-name, qhelp, qreplace) -> src-NSW-filename, dst-NSW-filename

A tool uses this primitive to change the name of an existing global space NSW file. It renames one global space file to be another global space file. The WM verifies that filespec designates a unique file to which the user has delete access. Enter access is also required for generating the new file name. The new file acquires any tool supplied attributes of the old file. A set file semaphore blocks a RENAME operation. Qhelp and qreplace are used according to their definition. Both source and destination NSW filenames are returned to inform the tool of the operation which actually took place.

3. COPYGLOBAL (filespec, entry-name, qhelp, qreplace) -> src-NSW-filename, dst-NSW-filename

A tool uses this primitive to create a new global NSW file which is a copy of an existing global space NSW file. It is similar to RENAMEGLOBAL with the exception that on completion both the source and destination files exist. For global space copy, enter access is required as well as delete access if copying to an already existing file name. Qreplace is a boolean which when true causes file replacement to be the default on collision of file names. Qreplace with value false means that either help must be obtained or the operation must fail. Both the source and destination full NSW-filenames actually used to complete the COPYGLOBAL are returned for the information of the tool.

5.8.2 Primitives for File Movement Between Spaces

4. GET (filespec, input-attribute-code, qset, qhelp) -> NSW-filename, new-filespec (only if changed), version #

A tool uses this primitive to cause a copy of the global space file denoted by filespec and having the attributes specified by input-attribute-code to be moved into the tool workspace. This working copy can then be manipulated by the tool using workspace file access primitives. When GETting a copy of a file obtained from the central catalog, the WM verifies that the user has copy access to the file, and that the file has the input-attributes specified by the tool. If these conditions are met, the WM initiates the proper actions to have a copy of the file moved into the tool workspace. This file movement may involve a network file transfer. When calling for a file transfer, the WM also insures that any file conversions which are necessary and possible are indeed performed. File conversions are based on the current state of the file and the intended use of the file by the tool (see The File Package document). Qset indicates whether or not the tool desires to set the semaphore associated with the original copy of the file. Note that in the event that the tool does not choose to utilize semaphores (this is indicated in its static tool descriptor) then the WM may automatically set the semaphore regardless of the value of qset. Qhelp indicates how the tool wishes to handle name ambiguity. The full NSW-filename of the file actually copied into the workspace is returned, as is any new filespec for this file (possibly obtained via user help). The version number of the workspace copy is also returned for the information of the tool. GETting a copy of a file for which the workspace already has the matching filespec and NSW-filename causes a new highest version to be created. GETting a copy of a file for which the workspace already has a matching filespec with a different NSW-filename will cause an error return to the tool.

5. PUT (filespec, version #, entry-name, output-attribute-code, qreplace, qhelp) -> NSW-filename

A tool uses this primitive to place a copy of a workspace file into the global NSW catalog. The file is identified by filespec and version number. Entry-name is the full NSW-filename (less any defaulted Entry scope) the tool wishes the file to have. If not specified, entry-name defaults to the name part of the full NSW-filename contained in the LND entry for the filespec (this is usually the same name as the one returned from the GET operation). If the LND has no NSW-filename and entry-name is not specified, then the Foreman returns failure to the tool. The WM requires that the user have enter access in order to deliver new files into the

global space. The output-attribute-code is a tool dependent code denoting an attribute which should be associated with the file. The WM will convert these codes to textual attributes which become part of the full NSW-filename. Qhelp guides the WM in seeking help with filename conflicts, and qreplace indicates whether the tool desires that the current file replace any file which may already exist with the same name. The full NSW-filename of the file as it is put into the global catalog is returned to the tool. A copy of the file also remains in the workspace, and can be referenced again using any workspace file manipulation primitive.

5.8.3 Primitives for Workspace File Manipulation

6. OPEN (filespec, version #, new-file-flag, old-file-only-flag, type-of-access) -> file-handle

The tool uses the OPEN primitive when it wants to actively access file data in a workspace NSW file, or to create a new workspace NSW file. A successful OPEN returns a handle for the referenced file. The handle is intended to be used when subsequent manipulations of the file data are requested. The nature of the handle, as well as the primitive operations available for the actual data manipulation are host dependent, based on the existing host file system, and are beyond the scope of this document. The handle is necessary since tools use NSW syntax for dealing with NSW domain files, and for the most part remain ignorant of the intermediate representation of the file in the local host file syntax. However, a Foreman implementation is not forbidden from using the local host syntax for the file as the handle returned from the OPEN, although this is not recommended. The file handle is also used to query the Foreman regarding any NSW information the Foreman maintains about the file, including the full NSW filename, known attributes, etc., should such a primitive be implemented.

If the new-file flag is set, a new local workspace file is created using filespec and version number (default is next higher version). The only failure for creating new files, other than failures due to the nature of the specific workspace implementation, is when specifying a version number of a filespec which already has such a version.

If the old-file-only flag is set, then success can be returned only if an existing file adhering to the filespec,version specification is found. If neither the new file flag nor the old file flag is set, then a failure to find an existing workspace file results instead in creating a new local file referenced by filespec.

The type-of-access parameter is optionally specified by the tool to indicate more precisely the type of file access it requires (e.g. read, write, read&write). The default for type-of-access is read & write. A Foreman may find the type-of-access information useful in determining whether a file is being modified (and may need to be delivered back into the global space), and in utilizing the structure of the underlying file system.

The search for an existing file matching filespec is within the local workspace only. Version number defaults to the highest existing version (except for new file as outlined above).

7. CLOSE (handle, output-attribute code, qdisp) ->

The tool uses this primitive to indicate that it has completed accessing the file denoted by handle and that the system should now assume responsibility for it. In addition, using qdisp the tool can guide the Foreman as to the ultimate disposition of the file i.e. whether it needs to be placed as a global NSW file (qdisp=true), thereby becoming referenceable by other NSW tools; or whether it can remain a workspace file until the end of the session or until such time as the tool instructs the Foreman to do otherwise (qdisp=false). The default value of qdisp is true, i.e. deliver the file at the end of the session. Completion of the CLOSE invalidates the handle for the file, and further access to the file must be preceeded by another OPEN.

The output attribute code is a tool dependent code denoting an attribute(s) which should be associated with the file. When the file is delivered to the global NSW space, the WM will convert these codes to textual attributes which become part of the full NSW filename.

8. DELETELOCAL (filespec, version #) -> version of deleted file

This is the primitive a tool uses to delete an existing file from its workspace. For DELETELOCAL only the workspace is searched for the matching filespec. The default version number is the lowest numbered version. Once a file has been deleted, it will no longer be accessible with OPEN, PUT, etc. The version number of the file actually deleted is returned to the tool on a successful deletion.

9. RENAMELOCAL (from-filespec, from-version #, to-filespec, to-version #) -> from-version #, to-version #

A tool uses this primitive to change the name of an existing workspace file. It renames one workspace file to be another workspace file. The new file acquires any tool supplied attributes of the old file. For RENAMELOCAL, from-version # defaults to the highest existing version and to-version # defaults to a new highest version for the file. The version number of the files actually operated on are returned to the tool.

10. COPYLOCAL (from-filespec, from version #, to-filespec, to-version #) -> from-version #, to-version #

A tool uses this primitive to create a new workspace file which is a copy of an existing workspace file. Both the "from" and "to" files exist in the workspace on successful completion. Note carefully that COPYLOCAL merely makes a copy of the file. It does not provide access to the file data. In general, since the actual local host name of the file remains unknown to the tool and no handle for it is provided through COPY, accessing the file requires an OPEN primitive. The from-filespec can not normally be defaulted, but the to-filespec defaults to that selected for the from-filespec. Default versions are highest version and next higher version for from-version and to-version respectively. The version numbers of both the "from" and "to" files are returned to the tool.

5.9 Other File Related Primitives

There are a few other file related primitives which are thought to be needed but not necessarily for the current set of tools in the initial configurations.

5.9.1 Global Space Primitives

11. WARRANT (filespec, attribute code) -> new NSW-filename

This primitive is used by a tool to assign attributes to a global space NSW file. (Recall that attributes can also be assigned to an open file at the time it is CLOSEd, and also when it is PUT into the global catalog. These are probably the most prevalent means for assigning attributes to files). The attribute code is a tool specific indicator for textual attributes which become part of the file name. The new full NSW-filename is returned to the tool, since the result of a WARRANT may actually change the filename. Filespec must uniquely identify an NSW file. Help is not provided, since only tools (i.e. not users) can assign attributes to a file.

The warrant capability is not yet supported by the WM and therefore need not now be supported by the Foreman.

[For completeness, we refer the reader to the previously mentioned semaphore related operations, which are also global space primitives.]

5.9.2 Local Space Primitives

12. GETFILEDESCRIPTOR (local filespec or handle, version #, data fields) -> data structure with specified items

(A primitive of this type is an optional implementation item). This primitive is used to view the information associated with a workspace file through its LND. Typical data fields will include: full NSW-filename, file attributes, existing versions, etc.

13. CHANGEFILEDESCRIPTOR (local filespec or handle, version #, data structure with changed items) -> change outcome indicators

(A primitive of this type is an optional implementation item). This primitive is used to change the LND information associated with local workspace files. Some LND information may not be subject to change. The exact nature of the information kept in the LND will be implementation dependent.

5.9.3 File Movement Into and Out Of the NSW System

The following four primitives are used essentially to move files into and out of NSW controlled spaces, either the global NSW filespace or the tool workspace. The primitives serve as interfaces to Works Manager facilities of the same name. READDEVICE and WRITEDEVICE are used to move copies of non-NSW files into a tool workspace, and to move copies of workspace files into non-NSW controlled space. IMPORT and EXPORT perform the same functions using the global NSW filespace as its base of operation. By non-NSW file space we mean not only space on file oriented devices, but also physical devices such as card readers, line printers, magnetic tape, etc. A user profile guides the default locations for the various physical devices. That is, a tool might request that a particular file be written (WRITEDEVICE) to the LPT (lineprinter). The user profile would indicate which lineprinter was local to the user, and perform the transfer. The details of using the user profile are currently being worked out.

14. EXPORT (filespec, external-name, password, qhelp) -> NSW-filename

EXPORT copies a global space NSW file to a non-NSW destination. EXPORT verifies COPY access and sends a copy of the source file to the location designated by external-name. An external-name is either an ARPANET pathname or a device pathname. Password is a string which is used for gaining access to the external directory, device, etc. The full NSW-filename of the file actually EXPORTED out of the NSW file system is returned for the information of the tool.

15. IMPORT (external-name, password, entry-name, qhelp) -> NSW-filename

IMPORT is the inverse of EXPORT, i.e. bringing a non-NSW file into the global filespace.

16. READDEVICE (external, password, filespec, version #) -> version #

READDEVICE is used by tools to input from sources outside the NSW without making a global space file. The file is placed directly in the tool workspace. Version # defaults to a new highest version. The actual version number of the created file is returned to the tool. When (if) the file is placed in the global file space, it must be given a full NSW-filename.

17. WRITEDEVICE (filespec, version #, external-name, password) -> version #

WRITEDEVICE is the inverse of READDEVICE i.e. copying a workspace file directly to a source outside the NSW domain. Version # defaults to the highest existing version. The version number of the file actually transferred is returned to the tool.

[Only IMPORT and EXPORT are available for tool invocation in the current version of the WM.]

5.10 Implementation of the File Primitives

The WM has procedures that can be invoked by the Foreman to implement the global space file manipulation operations. There are procedures for deleting, renaming, and copying global space files. These procedures and their call/return sequences are described in the Works Manager Procedures document. Each procedure is invoked by sending a generically addressed message

to a Works Manager process. Every procedure call generates a reply which can be obtained using the ReceiveSpecific MSG primitive. Replies to multiple outstanding procedure call messages can be distinguished through the conventional use of transaction IDs. These IDs are generated by the invoking process (Foreman) and are included in the message specifying the procedure call. The recipient of the message (a WM process) includes the transaction ID of the call in any reply that it generates. The NSW message transmission conventions (see Postel's note of 10 March 1976) also include indicators of whether a message is a new request or a reply to a previous request. This enables the Foreman to distinguish replies for its WM requests (e.g. W$DELETE) from WM commands regarding the tool (e.g. F$STOPTOOL), since both types of messages are received using the same ReceiveSpecific MSG primitive.

For GETting a local workspace copy of a global space file, the Works Manager's W$OPEN procedure is invoked. The local host syntax file name (of the new workspace file) which the WM returns is used as part of the basis of a new LND entry reflecting the NSW name given to the copy. For PUTting a file into the NSW global space, the Foreman merely invokes the Works Manager's W$DELIVER procedure regarding the local host file indicated by the LND entry associated with the workspace filespec.

For local workspace file delete, rename and copy the obvious LND manipulations are performed. The local host operating system will usually provide help in actually deleting the files, should this be desirable. If not, and also in the case of RENAMELOCAL, merely changing the contents of the appropriate LND entry is sufficient, since the tool does not deal with host syntax file names anyway. The OPEN and CLOSE primitives are implemented entirely within the Foreman to perform the function of relating the NSW file syntax and conventions to the underlying host file system.

The Foreman is not expected to implement controls over the type of access a tool has to workspace file copies since there is no NSW concept of file write access, append access, etc. The NSW is based on getting xerox copies of files, performing arbitrary operations on the copies, and then trying to deposit the altered copies back in the global space. It is the act of obtaining a copy (copy access) and the act of placing a new file (enter and possibly delete access) in the system that require access control. (However, since the host file system may require more specific access type information, the implementation of a Foreman for a particular host may require additional parameters indicating the type of access a tool needs to the particular file (i.e. the type-of-access parameter of the OPEN primitive). Whether or not this is included, at file CLOSE time, it is the responsibility of the Foreman to attempt to determine whether or not a file has been modified, and may therefore be a candidate

for re-delivery into the global catalog. Modified files should be so marked within the LND as an aid in post-tool delivery decisions.

5.11 Extension of the File Name Syntax

In the implementation of primitives which refer to a tool user's files, it is often useful to have the system itself (in this case the Foreman) gather from the user the strings for identifying files. An example of such a facility is the GTJFN (Get JFN) system call in the TENEX operating system, where as an option, the program can defer the actual accumulation of the filename string to the operating system. The alternative is to have each tool gather its own filenames by using available communication facilities. For tools that utilize direct channel communication with the user, having the option of specifying the connection to the user instead of a filespec, and letting the Foreman gather the filename string can lead to a much simplified tool implementation. It is recommended that Foreman implementers consider such an interface to their file system primitives.

However, whether the Foreman or the tool gathers the filenames, the user is often the ultimate source of the parameters supplied with the file system operations and as such, the NSW user must be provided with a way to syntacticly specify the exact file on which to operate. That is, the user level NSW file syntax must at least include an option for specifying a particular version from a set of workspace files. If a tool does not provide separate user commands for operating on local and global files, then it may also be necessary to syntactically specify the space to which a filespec refers. We think it important to present these features uniformly to the user, independent of the tool/Foreman he is currently using. In that regard, we are now specifying a syntactic extension to the NSW filename, which can be used by NSW tool users to explicitly specify a version of a particular file. Further extensions delimiting the domain of a filespec may also become appropriate. We emphasize, however, that these extensions are usable only within tools and Foremen, and have no meaning whatsoever at the WM command language level. A user must understand the local workspace concept and when it applies to grasp the meaning of the syntactic extensions.

The syntactic extension consists simply in adding a field to the end of the NSW filename syntax. This optionally specified field is to be delimited at the beginning by a semi-colon (";") character. Following the ";" can currently only be a decimal integer between 1 and 32. This indicates a particular version of a file. By its very nature, a file specified with a version number must be a workspace file, since version numbers are not supported at the global space level. Other extensions will be defined as necessary. Filenames which do not include any

extensions (currently a version number is the only possible extension) will take the normal default for the particular operation.

example:

WALDO.GEORGE.TXT;23

the local workspace for use or creation depending on the context in which it is used.

The determination of the version can be derived from either the parameters associated with a call (i.e. version #) or explicitly from the syntax of the filespec provided. Filespec syntax takes precedence over tool parameters in the event of conflicts, since we assume the user to be responsible for most syntax related directives.

VI. Tool to Front End Communication

The NSW user accesses the NSW system through a Front End process. For those tools that require direct user involvement, the Foreman and the Front End must cooperate to provide channels for the communication. We are firmly committed to providing tools with the ability to utilize MSG for both message type communication and direct connections with the FE. The FE could interpret and package user input and transport the pertinent data to the tool in a network MSG message. The tool to FE communication could be handled in an analogous fashion. Another approach to tool/FE communication is through the use of direct network connections. This would typically take the form of an ARPANET telnet connection pair from the FE directly to the tool. The decision as to which type of communication facility a tool uses is left entirely to the tool builder. The extent and type of user interaction which the tool supports, as well as the possibility of additional burden on the FE system must be weighed in selecting a mode for tool communication. Using the techniques outlined in the MSG document addition NSW Note #11 (and included as Appendix 2 of this document) we will support tool communication with the FE using direct (but controlled) tool access to both the message and connection oriented MSG facilities. A tool will be able to selectively use messages, or sets of connections, or both, depending upon the tool circumstances. However, again letting immediate necessity drive our initial efforts, we find that the initial tools are not written using a message type FE interface. Rather, they utilize a terminal oriented interface, best served by a direct connection from the tool to the FE (and hence the user). Because of this, we temporarily defer extensive details of the tool-FE message interface. These details will be of primary concern immediately after the initial Formen implementations are complete. We do require however, that the Foreman support direct FE to tool connections as an immediate objective. This does not require any of the modifications mentioned in Note #11, and hence is in line with the short term implementation plan for all NSW components.

The Foreman initially received the MSG process name of the FE process servicing its tool (see description of F$BEGINTOOL request). Based on this information, the Foreman is required to implement a CONNECTION-TO-FE primitive operation for its tool. To establish the (Telnet) communication path between the FE and the tool, the Foreman sends an MSG message to the designated FE process. The message is a request to exchange MSG connection operations, and indicates that the connection should be of type telnet. After sending the request, the Foreman immediately issues its MSG Openconn primitive directed toward the FE process. If the Openconn succeeds, the handle for the connection(s) is returned to the tool as the response to the CONNECTION-TO-FE primitive. If the Openconn fails after a sufficiently long timeout and retry period, then the Foreman reports failure to the

tool. The MSG message sent to the FE process requesting the connection requires no acknowledgement. The completing of the connection serves as a positive acknowledgement to the request.

In cases where the Foreman knows that the tool requires a direct FE connection (e.g. encapsulated tool), the implementation may be such that the Foreman acts to create the connection without requiring the tool to request it. However it is accomplished, the initial Foreman requirement is that each tool be provided with a means of using a direct telnet connection to its FE process. The exact nature of the FE support for tool connections is detailed in the forthcoming document describing a minimal Front End.

VII. Tool to Tool Invocation and Communication

7.1 Present State

The mechanisms for tools invoking other tools or interacting with other existing, background tools, and then for tool-to-tool communication certainly constitute a part of the abstract tool environment. However, no tool from the set of initially anticipated tools needs to use such facilities. Therefore, we are postponing the precise description of the mechanisms provided to tool builders for dynamically creating other NSW entities and communicating/synchronizing with them. At this point however, a rough sketch of the planned mechanisms and a possible implementation strategy can be given. It must be emphasized that much of the content of this section is still in the design stage, and is presented here only to give a more complete picture of a future direction. The emphasis placed on these areas is dependent on the nature of the tools which will populate the NSW, and on whether or not people are willing to customize their tools for the NSW. To even allow the possibility of extensive customization, we are presenting the concepts surrounding these other aspects of the tool environment. It is difficult to judge the impact of these extensions in the absence of tool candidates which need to make use of them. However, we will pursue the refinement of some of the tool-to-tool concepts so that as tools emerge which require such facilities (as they surely will), we will have a cohesive approach for handling them. Implementation may await an expected use. To this end, we invite comments and suggestions on these more complex uses of the NSW.

7.2 Emerging Tool-Tool Concepts

As in the file system operations, the WM and the Foreman in combination are responsible for the dynamic creation and communication aspects of the tool virtual machine, with a large assist from MSG. A general overview of the supported facilities is that there is a variant of the Works Manager's W$RUNTOOL procedure which can be invoked via the Foreman by a tool to create a new instance of another tool. Each tool will be able to use MSG facilities for sending and receiving specifically addressed messages, sending and receiving alarms, and setting up and taking down direct connections, all to a selected list of conversants which includes any tools it has started. Some of the ancillary features of MSG are expected to be included in the tool domain (e.g. Rescinding MSG primitives, Resynchronization with a correspondent). The direct use of MSG by tools is based on the concepts outlined in Appendix 2. The tool will also be provided with primitives for locating service facilities which are implemented as tools, but which do not dynamically become part of the initiating tool's job, as is the case with tool-to-tool creation. With proper verification, the tool can then engage in

message and/or connection oriented exchanges with the service tools.

For dynamic tool creation as well as for trying to locate and utilize a background service tool, we provide tool primitives which are fielded by the Foreman. The WM implements procedures which perform the access checking as well as establishing new components where needed using MSG facilities, and returns the pertinent information to the initiating Foreman. The information returned includes the MSG process address of the new tool. The initiating Foreman then manipulates its tool's environment using MSG primitives to allow message and/or connection type of communication between the tools. The initiating tool can specify the MSG process name of a process in its family tree which is to serve as the FE process to the new tool. The Foreman of the newly created tool receives the address of the creating tool as well as the process which is to serve as the tool FE (if any) and adjusts its tool's environment to facilitate communication with these processes. In addition, we envision providing primitives with which the creating tool can control the progress of the new tool and terminate it, in much the same fashion as the user can control a tool through the WM command language.

Once the tools are in communication, they can pass around the names of other tool instances that they know about. We perceive a Foreman call by which a tool can ask that "communication to process xxx" be allowed. The Foreman would try to get the WM to consent to the pair as legitimate conversational partners. If they are, the appropriate Foremen are notified to adjust the MSG environment for their tools, and communication can proceed without further Foreman intervention. How the WM decides if two tools should be allowed to communicate is an aspect of the tool-tool model still undergoing investigation. Another very important aspect of the tool-tool problem, which also has no immediate solution, is handling the situations in which system failures cause breaks in the process trees.

VIII. Tool Encapsulation

The initial TENEX approach to integrating tools into the NSW was through an encapsulation technique. This approach has proven very successful, and we, therefore, feel that each Foreman implementation should consider a similar facility.

In general terms, NSW encapsulation implies the automatic trapping and translation of local host operating system calls into calls meaningful in the NSW system. Any trapping and translation is done within the Foreman process. Using an encapsulation technique, we take programs which are written exclusively for the local host operating system execution environment, and with little or no modification execute them as NSW tools. This is possible only because of the similarity, in many aspects, of the NSW system to a conventional single host operating system. As an example, when an encapsulated tool issues a local system primitive to gain access to a file, the Foreman could get control and translate the request into one which provides access to an NSW file. This assumes that the "old style tool" is somehow capable of handling the NSW filename syntax within the local host file manipulation primitives. In TENEX, for example, this is often be very easy since the tool will frequently allow the "system" to gather the filename from the user. In TENEX encapsulation, the Foreman is interposed between the tool and the operating system only for selected system calls. With its intimate knowledge of both the local system primitives and the NSW system structure, the Foreman gathers the NSW filename and ensures that the tool utilizes NSW files. Using both local host facilities and facilities supported by other NSW components (e.g. WM), the Foreman "implements" the local host file primitive in a new context.

Encapsulation cannot be discussed in terms of its algorithms. It requires an extensive knowledge of the local host operating system primitive operations, and a determination of how they can be made to relate automatically to the NSW environment. Thus each TBH approach to encapsulation will probably be different. As far as the other NSW components are concerned, running an encapsulated tool is no different from running a tool which was written to function in the NSW. The behavior of the FE and the WM is identical in the cases of the integrated and non-integrated tool, with the exception that the WM notifies the Foreman in the F$BEGINTOOL message that it will be running a tool which requires encapsulation. We can, however, speak in general terms about certain aspects of encapsulation.

Encapsulation requires some mechanism with which the Foreman can gain control after the tool executes certain operating system functions, but before the operating system proceeds with the local implementation of the operation. The TENEX JSYS trap facility is an example of such a mechanism. It is entirely left

to the encapsulation implementation to determine which system calls need trapping and how to integrate these calls with NSW facilities. For the most part, the tool initialization and termination conditions, interactions with the file system, and the communication with the tool user will all require careful attention within the encapsulation component of the Foreman. In some cases, mapping the local system operation into a comparable NSW facility will be straightforward. An example is the terminal interface which drives many tools. The Foreman can simply request the creation of a direct FE connection of type Telnet, and provide this "NSW connection" to the encapsulated tool. In other areas, the Foreman has a wide range of possible implementation strategies. An example of this type is the handling of file delivery into the NSW file system. Since encapsulated tools are not aware of the NSW file system, they cannot guide the Foreman as to the disposition of the files. The Foreman must choose an implementation strategy for delivering new and changed files to the global NSW file space. This can be done as the files become available (i.e., closed in most operating systems), or only at the end of the tool session, or even anywhere in between. Each encapsulation implementation selects the strategy most appropriate for the anticipated needs of the tools for that host.

TENEX NSW encapsulation already exists for selected tools. In general, the simpler a tool is, the more easily it can be encapsulated. By simple, we mean the straightforward use of common operating system facilities. Such facilities are apt to have analogous mechanisms in the NSW, since the NSW caters to many of the same aspects of the tool environment but with wider domain. Depending upon the effort placed into translating system calls, a Foreman will be capable of encapsulating an expanding set of "old tools." However, let us emphasize that encapsulation has limitations. There will always be local host programs which cannot be NSW encapsulated. This is because the NSW system IS different from the local host system, and substituted components can be made to appear similar only to a certain degree. Tools which utilize obscure features, or features peculiar to a particular operating system are sure to be difficult or impossible to encapsulate correctly. Very often this will mean that certain features of a tool are not available when using the tool encapsulated. If this is not satisfactory, or if other problems prevent the tool from being encapsulated (e.g., the local host does not have system facilities for building an encapsulator) then the tool program must be modified to directly call Foreman NSW primitives if it is to function as an NSW tool. Let us also emphasize that for a tool to be most effective in the NSW domain it should be coded using the NSW facilities directly.

We feel that for some TBHs encapsulation can have a high payoff in establishing a large class of programs as NSW tools, and should be actively explored. It is often undesirable to recode existing programs, and it is in this area that

encapsulation has its maximum effect. Designing an encapsulator is in many ways similar to designing a tool which directly utilizes the NSW facilities. As such, much of the discussion in the preceding sections of this document will be helpful. As a note of interest, the form of the TENEX encapsulator for the initial test NSW system has influenced the design of the Foreman component, since in a way, the encapsulator was an integrated tool. Some of the concepts embodied in the TENEX encapsulator are discussed as part of Appendix 1, to serve as a model to other encapsulator builders.

IX. Batch Tools

Until the present section, this document has primarily focused on interactive tools. Interactive tools are tools whose users are on-line while the tool is running. There are also batch tools - tools whose users may not be on-line. The primary difference between batch and interactive tools, therefore, is the time at which information about input/output files, parameters, etc. is obtained. A user can supply information to an interactive tool at any time while it is running. A user must supply information to a batch tool before it is run. Thus, an interactive tool needs controlled access to NSW resources while it is running, and an active Foreman supplies the controlled access. Access control for batch tools can be done before the tool is run, so a much less complex Foreman is needed. (Note that nothing in this discussion precludes building a Foreman for batch tools with all of the functions described in this specification. We are merely pointing out that such a complex Foreman is not required.)

We shall sketch the features that are absolutely necessary in a batch Foreman. Since batch tools in NSW will be handled by the IP protocol for the immediate future, we defer details until a later version of this specification.

In the current NSW model, execution of a batch job is handled by the Works Manager Operator (WMO) process. The WMO is given (by the WM) tables which contain skeleton job control language (JCL) and a mapping between dummy parameters in the skeleton JCL and NSW files, real values, etc. The WMO prestages the job on a selected batch host by asking the WM to move (via the File Package) all input files to that host. The skeleton JCL is then edited to insert local file names (obtained as a result of the file movement) and parameter values. The IP server on the batch host is then given the JCL and told to run the job. When the job has run to completion, the IP server informs the WMO, which then moves (again via the FP) the result files into NSW file space. The minimum batch Foreman must support this model. That is, it must have a WMO-invokable F$SUBMITJOB procedure and it must invoke a WMO procedure W$JOBHALT. (In addition, it must support status probes.)

A slightly more complex model requires that the batch Foreman receive the tables now given to the WMO. The batch Foreman would then be responsible for moving input files (either prestaging or during execution) to the batch host, editing the JCL, running the job, moving result files to NSW file space, and informing the WM that the job was complete. We expect that many batch hosts will prefer to control job execution more completely in this later fashion.

Finally, some hosts may choose to implement a complete Foreman. F$SUBMITJOB would then be F$BEGINTOOL and file motion would be handled dynamically. This last case is the least explored of the possibilities although we expect that batch jobs on interactive hosts (TENEX, MULTICS) will be handled by this mechanism.

The WM and WMO will support these several different kinds of batch Foreman so that batch tools may be run on hosts as diverse as B4700, 360/91, and TENEX.

Appendix 1. Functional Summary

This appendix summarizes the externally invocable functions which must be implemented by each Foreman, and proposes parameter value conventions for the functions. Transmissions currently follow the SRI conventions except where noted. That is, transmissions are modified PCPB8 data structures of the form:

LIST (type, length, tid, parameter, args).

This appendix will be modified from time to time, as needed.

Functional Description and Transmission Formats

A. Functions implemented within each Foreman

(Note: Because of a phased implementation plan, and because all functions may not be applicable to all host systems, implementation may consist simply of replying with a rejection message. In that sense, all functions must be implemented (recognized) by all Foreman from the outset. The error code reply value of 177777 (16 bit value) will be taken to mean unimplemented function.)

All functions are invoked with a reply requested (i.e., using TID) except where explicitly stated. Recall that the F$ prefix is used as an expository aid in indicating a function implemented in a Foreman. Where the string "n-" prefixes an element, it should be read as the element repeated n times.

A.1 F$BEGINTOOL (program-name, tool-type, entvec, FE-addr, cr-addr, filename-list)->result, qstart, workspace-descriptor, tool-addr

Program-name:
charstr: local host syntax completely specifying the program to be run as NSW tool

tool-type:
index =1 -> encapsulated tool
=2 -> tool uses NSW calls, does not use MSG
=3 -> tool uses NSW calls & uses MSG

entvec:
index =0->do not start tool (illegal except in F$BEGINTOOL)
=1->continue from point stopped (illegal in F$BEGINTOOL)
=2->cold start entry point
=3->warm start entry point
=4->termination routine entry point
=7->tool specific entry point
.
.
.
(Please note that the index value assignments for entvec are changed from those indicated in the text of the Foreman document.)

FE addr:
procaddr (new data type corresponding to MSG process address)

cr-addr:
procaddr

```
filename-list:
   list(n-filenames)
         filename:
             charstr:  full local host syntax
result:
    empty -> success
    index -> error code          (error codes to be defined)

qstart:
    boolean = true -> program started
            = false -> program not started

workspace-descriptor:
    list (name, access-info)
              name: charstr
              access-info:  charstr -> info used by File Package
                                       to access workspace via name
                            empty -> access info not needed by File
                                       Package

tool-addr:
     procaddr
```

A.2 F$STARTTOOL (entvec) -> result

```
      entvec:  index (see above)
      result:  index (see above)
```

A.3 F$STOPTOOL (entvec) -> result

```
      entvec:  index (see above)
      result:  index (see above)
```

A.4 F$ENDTOOL (reason, termtype, qmaintain)-> result,
accounting-list, qmaintained

```
     reason:
     index  =1-> user request
            =2-> WM decision
            =3-> user disconnected

     termtype
     index  =1-> no LND processing necessary
            =2-> step thru LND directly with user
            =3-> automatically deliver latest copy of changed files

     qmaintain:
          boolean = true -> maintain tool image if possible
                  = false -> don't maintain tool image (default)

     accounting-list:
         list (cost, n-list(type,amount))
```

```
                cost: an integer reflecting the cost in cents of
                      running the tool.

                type: an index indicating the type of resource
                      accounted for
                  =1 -> CPU milliseconds
                  =2 -> connect minutes
                  =3 -> I/O operations
                  =4 -> primitive calls
                  =5 -> core usage
                   .
                   .
                   .  to be defined as needed
                    (note:  each TBH will select the types of
                    resource measures it will provide)
               amount:  integer
                        reflects the session utilization for the
                        corresponding resource type (either in
                        resource units or in cents).

     qmaintained:
         boolean = true -> tool image has been maintained
                 = false -> tool image has not been maintained
                                                  (default)


B.  Alarms to be recognized by each Foreman

B.1  alarm code = 1 -> forthcoming tool termination request
                response:  immediately begin processing input
                   messages looking for F$ENDTOOL request;
                   all other messages are discarded

B.2  alarm code = 10 -> status probe (name = STATUS)
                response:  return list (4-statevariables) to
                           invoking process
                   statevariable1: NSW external state
                        index = 0 -> running
                              = 1 -> stopped, never started
                              = 2 -> stopped
                              = 3 -> running at termination code
                              = 4 -> terminated (tool did HALTME)
                   statevariable2: NSW internal state
                        index = 0 -> running
                              = N -> awaiting  completion of NSW
                                     primitive # N
                                     (note:  the primitive name used
                                     by tools need not be uniform
                                     across all TBHs.  However, for
                                     status reports, we standardize
```

all tool functions by
equating each one with a code
indicating particular function
classes. It is this code
(non-zero) which indicates the
type of NSW function the tool
is executing).

statevariable3: current local operating system
state

index = 0 -> running
= 1 -> I/O wait
= 2 -> dismissed

.
.
.

statevariable4: current program counter
integer

NSW primitive functional classes:
1-> global file space manipulation
2-> local file space manipulation
3-> MSG communication
4-> tool invocation

B.3 alarm code = 11 -> accounting probe (name = ACCOUNT)
response: return accounting-list (defined earlier)
to invoking process

other alarms will be defined as needed

A note on responses to alarm codes -

There is currently no convenient way to signify a response to an alarm. Accordingly we are proposing the following addition to the transmission conventions as outlined by Postel.

the standard message transmission format is:
LIST (type, length, tid, parameter, args)

for a response to an alarm we specify that

type = 3 (definition of a new type)
length (same as before)
tid = alarm code (the id field contains the 16 bit alarm code for type 3 messages)
parameter (same as for type 2 = acknowledgement)
args (same as for type 2)

In accordance with this format, a defined alarm code which has no Foreman implementation should return an error reply value of 177777 (unimplemented function).
Receipt of an undefined alarm code can simply be ignored.

C. TENEX Encapsulation

This section, which sketches selected aspects of the TENEX NSW encapsulator, is included as a model for potential encapsulator builders. Encapsulation provides the implementer with large margins of flexibility, and each such implementer must decide upon the nature of an encapsulator best suited for the existing local host programs.

An NSW encapsulated TENEX tool is automatically set up with a network virtual terminal (NVT) to the FE process as its primary input and output device. The structure of the TENEX operating system has allowed the encapsulator to be programmed as an ordinary user process. With respect to the tool it is running, the encapsulator can gain control when the tool executes selected system calls, and in addition can read from and write to the same NVT which was given to the tool.

When a TENEX encapsulated tool requests access to a file (using GTJFN operation), the call is such that in general the program either provides a filename string or indicates a device or file which can be read to obtain the filename. To simplify matters, in this discussion we will consider only the cases where the program has specified that the filename is to be obtained from the NVT (i.e., from the user) or is provided as a text string parameter of the system call. If the call indicates the user as the source of the file name, the encapsulator reads the file name using the NVT. Since the file name has been supplied by the NSW user, we can be certain that it is an NSW syntax Filename referring to an NSW file. As such, we can simply check the LND for a copy of the file, and failing to find one, we can try to obtain a copy from the global workspace. In either of these cases, the tool is provided a direct handle for the TENEX file representing the NSW file copy. In the case of a parameter supplied filename string, the complexity increases. The filename may refer to an NSW file (in NSW syntax) whose name was previously obtained from the user. Alternatively, the filename may refer to a TENEX file (in TENEX syntax) whose use is outside the NSW (e.g., a documentation file). The WM provides a list of such locally accessible files to the encapsulator at tool startup time. Using this list, and based on its knowledge of both the NSW and TENEX file name syntax, the encapsulator determines if the file is a legally referenceable TENEX file or an NSW file. Access to a TENEX file can be granted outright. Once we determine a file name to be an NSW name then we search the LND and consult with the WM as necessary to provide file access.

For those cases where the program requests a "new" file (as indicated by the TENEX system call parameters) it must de facto be an NSW file, since encapsulated tools are not aware of the two different file systems. In this case, an LND entry is created and the encapsulated tool is given a handle on a TENEX file

representing the NSW file. TENEX also has the facility to create temporary files, i.e., files which disappear on logout. We have taken the position that tools can utilize temporary files unimpaired, since by their very nature they would not be candidates for being maintained in the central catalog.

Since the encapsulated tools are not reprogrammed for the NSW environment, they do not indicate which files need to be delivered to the WM and when. Thus when an encapsulated tool indicates (in TENEX terms) that it has finished accessing an NSW file, the encapsulator determines whether or not the file has been modified. If it has, then the file is marked in the LND as possibly requiring delivery to the global space at a subsequent time. We have taken the approach that file references are always interpreted locally where possible. That is, in general, file operations will only create new local workspace copies of NSW files and access these copies when they exist. No files are delivered to the global space until the tool terminates, and then only selected files (e.g., latest versions of changed original copies). Version numbering and defaulting are supported as in TENEX, with some assist from the LND. The TENEX encapsulator supports limited TENEX style command editing while accumulating a file name from the user.

Appendix 2. Foreman Induced MSG Additions

The following pages reproduce an appendix to the original MSG design document. The document was originally introduced under the name NSW Note #11. Because of the relevancy to the subject of this document, and because it was not included in the originally distributed MSG report, we are including it as part of the Foreman specification document.

NSW Working Note #11
January 27, 1976

The Impact of the Foreman Concept on MSG

The needs of the Foreman component of the NSW have motivated some proposed additions to the MSG facility. In essence, a Foreman is a local-to-the-tool component of the NSW. The Foreman provides an interface to the tool for the facilities provided by the Works Manager, and in addition helps to provide the NSW environment in which the tool is run. This note discusses one aspect of that environment, the communication facilities made available to the tool.

To a first approximation, the message oriented communication modes provided by MSG to the components that create the NSW environment are also appropriate for tools to communicate with other tools and with the user through a front end process. However tools, especially those in the debugging stage, cannot be allowed to function directly in the uncontrolled MSG environment.

The interprocess communication (IPC) needs of a tool, along with the IPC needs of the Foreman component imply the existence of two logical communication streams. One set of messages is destined for the Foreman, while the other stream is destined for the tool itself. If the IPC needs of a tool can be satisfied using direct connections only, then message traffic can be dedicated to the Foreman implementation. If, on the other hand, the tool must be provided with a message oriented IPC facility which is supported by or derived from the MSG message passing capability, then a multiplexing problem exists. In the following we assume that it is indeed desirable to provide tools with a message oriented communication facility for many of the same reasons that such a facility was desirable for building the NSW itself. We also assume, for obvious reasons, that such a facility will indeed make use of similar MSG functions. Therefore, we must address any problems this situation causes.

For the tool/Foreman complex in the current MSG context there seem only two possibilities: either the Foreman and the tool occupy a single MSG process or they do not. [Note that in any event, the Foreman must maintain a special relationship to the tool. That is, it is the Foreman that provides much of the NSW virtual environment for the tool.] In the case where both the Foreman and the tool occupy the same MSG process, the Foreman is required to receive all incoming messages in order to filter out the ones intended for the Foreman. This filtering would have to

be based on NSW addressing conventions transmitted as part of the message data. Messages intended for the tool would be passed to the tool by the Foreman using local operating system facilities outside the scope of MSG. The major advantage of this approach is that it is very convenient to apply the needed access controls on the tool's use of the message facility. The Foreman implements for the tool a new abstract IPC facility which is built upon the Foreman's use of MSG. The "new" IPC facility can be customized for tool use if this is desirable. The major disadvantage for the NSW stems from the fact that the IPC facility we want to provide to the tools is indeed very similar to that offered by MSG. We would like a somewhat restricted version of MSG. Yet to achieve this, we must incur an extra transfer of control between the MSG facility and the Foreman for each incoming and outgoing message of the tool. In addition, this may involve additional handling/copying of the messages and duplicating some of the functions already performed by MSG (e.g. setting up and queuing alarms, handling multiple operations). Furthermore, the Foreman's use of the MSG facility may conflict (interfere) with that of the tool (e.g. MSG queues only a single alarm; also, message sequencing is on an entire MSG process basis). Such conflicts may force an otherwise unnecessary change in the nature of the IPC facility available to tools.

An alternative NSW design is one in which the Foreman and its tool are separate MSG processes. Their distinct MSG addresses would mean that MSG itself could mediate the separation of Foreman messages from those of the tool. The main disadvantage of this approach is that MSG, as currently defined, does not provide any way for the Foreman to limit the tool's use of the message passing facility. The type of control needed for the NSW application is fairly well understood. That is, we would like to be able to limit the conversational partners with which the tool can communicate, and have the Works Manager/Foreman combination responsible for changes in the set of legal conversants. In addition, it may be necessary to forbid a tool from executing certain MSG primitives which are not directly related to the sending and receiving of messages.

In the following sections we outline several additions to MSG, which, if implemented, would allow an NSW tool to directly utilize the MSG IPC facility while allowing the Foreman to specify limitations on how the facility can be used. The MSG additions are in two basic areas. First, we would like to establish the ability of an MSG process to "introduce" to MSG a new process which MSG will then support. Such a new process has presumably been created using whatever facilities are provided by the local host operating system. The result of a process introduction is that the new process is given an MSG process name and can issue MSG primitives. Second, we would provide additional MSG primitives which allow an introducing MSG process to selectively manipulate an access control matrix which would be

associated with every introduced process. Such an access control matrix would indicate for each process the allowable objects of each MSG primitive. [In general, the object of an MSG primitive is an MSG process name.]

We view these additions to MSG as the cleanest, most effective way to bring tools into the NSW environment while providing them with a flexible message passing communication facility.

Introducing New MSG Processes

Many modern day operating systems provide mechanisms for dynamic process creation. There has been no limitation placed on the nature of an MSG process, so that multiple processes (in the local host operating system sense) can serve as a single MSG process. In this way MSG programs can continue to utilize any dynamic process creation primitives available on the local host operating system. However, as currently constituted, these MSG processes must share a single MSG address and hence a single MSG message stream. In some cases this is exactly what is desired (e.g. a structure which has one process (in the local system sense) sending all messages, while another does the receiving). However, in other instances (e.g. tool/Foreman) the concurrent, asynchronous operation of multiple processes would be impeded by the single message stream, and force each such MSG process to implement its own local dispatching.

As suggested above, one way to allow MSG itself to mediate the message stream between concurrent but cooperating processes is to assign each a separate MSG address. It may at first seem attractive to add to MSG the notion of a general purpose "create new process". Such a general facility is not necessary for building the NSW. To be sure, such a general inter-host process creation and manipulation mechanism is a goal we have in creating the support environment for the tools themselves, but it need not be implemented by MSG alone. The current discussion is concerned with being able to more flexibly use within the MSG context whatever local host operating system process creation facilities are available. With that goal, there are a number of reasons for refraining from defining a standard MSG "create process" primitive. One is that in many cases the creating process needs to maintain a special relationship between itself and the created process to maintain a particular type of cooperation. This may take the form of being able to directly manipulate certain aspects of the created process, or perhaps results from sharing parts of an address space. In any event, it would be difficult to be able to represent all of the potential relationships from all of the constituent systems, and even more difficult to implement some of them. A second argument against a standard MSG "create process" primitive is that it would have to be accompanied by a suitable way of describing the process that was

to be created. Typically this is handled in the context of a file system, but there does not exist a unified MSG file store. [Although a unified multi-host file system is part of the NSW design, it is not realized at the MSG level.]

An alternative to a unified MSG "create process" primitive is the approach which acknowledges the local nature of the creation and specification of new processes, but allows the creating process to "introduce" to MSG the created process. Any special relationship between the processes, as well as the means of specifying how to create the process is handled on a strictly local host basis. It is presumably complete before the introduction is made.

After introducing a new process to MSG, we will have established two separate MSG addresses. The Foreman and tool can have separate MSG message streams, with MSG mediating between them. However, process introduction by itself does not solve all of the problems raised by the NSW Foreman application.

Limiting the Use of MSG Facilities

Up until this point, the MSG documentation has been careful to avoid specifying any hierarchy of control among the MSG processes. The MSG processes are largely independent of one another from a control standpoint. Communication between processes via MSG messages is unrestricted, with each process determining for itself the validity of any message it receives. The concept of "introducing" another process into MSG establishes a natural place for integrating a form of control by selected MSG processes over other MSG processes. The introduction of any such control is again motivated by the Foreman application. However, we attempt to define the use of the control facilities in a way which does not preclude its application in other contexts.

As a result of process introduction, the introducing process is established as the superior of the introduced process. The superior/inferior relationship represents a tight coupling among MSG processes, as contrasted to the loose coupling provided through message communication. Termination of a superior MSG process also causes the termination of its inferiors. Superior processes are permitted to regulate their inferior's use of the MSG facilities. This is accomplished through the addition of a control mechanism associated with the execution of MSG primitives. In general terms, the control consists of associating an access control matrix with each new MSG process, and allowing superior processes to manipulate (through MSG primitives) the contents of the matrix for their inferiors. Entries in the matrix represent permission to execute a particular MSG primitive on a particular object. MSG will reject a primitive call from one of its processes if the access control

matrix does not indicate that this (call, object) pair is allowable. An object is usually an MSG process name. However, some MSG primitives (e.g. Stopme) do not take objects as arguments. In such cases, a single entry regulates the ability to execute such a primitive.

It may be helpful to view an MSG process which has no MSG superior (i.e. has not been "introduced" by another MSG process) as having an unrestricted access control matrix. An MSG process can only supply its inferiors with rights that it currently has. Removal of a right from an immediate inferior causes removal of that right from any MSG process further down the hierarchy. Applying access control to the sending of messages (data) has the beneficial side effect of reducing bandwidth consumption by unauthorized messages. It also increases the confidence in the validity of messages which are received.

MSG Primitives

The following is a set of primitives which, if added to MSG, would allow the Foreman to function in the previously discussed mode. (This is just a rough sketch of the primitives, along with some possible implementation details).

1. Introduce New Process (pointer to process descriptor, pointer to initial access control information, location of returned MSG name, disposition)

This is the primitive which is used to introduce a new process into MSG. In response to this primitive, MSG establishes an MSG address for the introduced process. The generic component of the generated name is always null, implying MSG has no knowledge of the function performed by the process.
It also establishes the issuing MSG process as the superior of the process, and initializes the access control matrix based on the data passed in the parameter list. The new MSG name is returned to the calling process.

process descriptor: local host operating system dependent parameter for conveying to MSG the identity of the new process. The exact nature of this parameter is dependent on the entity which is discernable as a process on the local operating system.

access control information: list of pairs, where each pair is of type (primitive, list of objects). Primitive denotes a particular MSG primitive, and list of objects is a list of MSG process names. Special designations exist for the classes of objects "all" and "none". Individual process names include all defined MSG process name fields, with the addition that each field may optionally have the "all" designator.

2. Renounce Process (MSG process address, disposition)

This primitive is the inverse of process introduction. MSG checks to see if the issuing process is the superior of the object process, and if so MSG causes the removal of the object process (including removing any knowledge of its existence from MSG tables). This also forces immediate rejection of all outstanding MSG operations on behalf of the object process. Any MSG processes introduced by the object process are similarly renounced. Note that this does not have to necessarily result in the destruction (in the local operating system sense) of whatever constituted the MSG process. It only makes MSG itself unaware of its existence.

3. Update Control Table (MSG process name, add/delete indicator, pointer to access control information, disposition)

This primitive is used to manipulate the access control information associated with an inferior MSG process. MSG checks to see if issuing MSG process is the superior of the object process, and if so updates the access control matrix of the object process according to the supplied parameters. In the case of additions, the (primitive, object) pair specified must be currently accessible to the issuing process in order for this update to succeed. A deletion causes the same pair to be removed from any inferiors of the object MSG process.

access control information: same as defined in the introduce primitive.

add/delete: boolean which distinguishes adding entries from removing them.

Chapter 6: Hardening, Scaling, and Optimizing of the Works Manager

I Introduction

This document is a plan for hardening, scaling, and optimizing the Works Manager (WM) component of the National Software Works (NSW). As such it does not directly treat other components of NSW - e.g., Front End, Foreman, etc. Nevertheless, the (not surprising) conclusion we have reached is that large scale and reliability of NSW as a whole can be obtained by distribution of the WM over many computers.

The other components of NSW - namely, Front Ends, Foremen, tools, MSG (protocol) servers, and File Packages - are already distributed. Further, failure of a single one of these components only reduces the available resources of NSW; it does not halt the system. Conversely, the resources of NSW can be increased by adding additional instances of these components. Thus, if we are able to make the WM (the only centralized component) both reliable and large scale, then we will also make NSW as a whole both reliable and large scale.

The remainder of this document is organized into three parts. The first part consists of a statement of the problem - what do we mean by hardening, scaling, and optimization. Part two is a brief discussion of various alternate solutions to our problem. Part three is a more detailed description of the solution we chose - the Pluribus multiprocessor. Part two is considerably shorter than part three since we chose not to describe the inferior choices in great detail.

II Hardening, Scaling, and Optimization

The external statement of the problem to be solved was to design a reliable, efficient system capable of supporting 1000 concurrent users.

In this section we present our interpretation of this problem. The first comment to make is that the three goals are, to some extent, mutually exclusive. To make a system large scale generally requires either many components or complicated components. More components mean greater probability of single component failure, and presumably, less reliability. Similarly, complex components will fail more often than simple ones. Reliability usually involves much checking of correctness and therefore degraded performance. (A component performing a reliability check is not serving a user directly, so with similar sets of components the more reliable system will usually provide less service.)

Thus, part of our task is to reconcile the conflicting demands of the three goals. We have made the following explicit choices:

- Large scale is an external requirement. If we cannot propose a system capable of handling the requisite number of concurrent users (1000) at acceptable cost, then we have set our sights too high, and the external requirement must be restated.

- Reliability means a fail soft rather than a fail safe capability. That is, the proposed system should try to always provide some level of service, but this level may degrade in the event of multiple hardware failures. Further, the system should be able to recover on its own from almost all hardware failures. In addition, even in the event of catastrophic failure, the state of the system should be saved so that users do not lose more than insignificant amounts of work.

- Optimization is less important than either scale or reliability. Since the WM provides only monitor, not computing, services, users can tolerate some sacrifice in performance if such sacrifice is necessary to achieve either the desired scale or reliability.

The next section discusses some of the possible alternates which would implement a system with these characteristics.

III Alternate Solutions

This section discusses various approaches to the problem of hardening, scaling, and optimization and briefly sketches reasons for rejecting all but one approach. This discussion is presented in a logical order which should not be construed to be an exact reflection of actual chronological order. In fact, all approaches were considered more or less concurrently. Whenever some approach was determined to have a substantial defect, we shifted attention away from it and concentrated on other possibilities. We do not intend, in this section, to present a detailed view of every possibility we considered. Instead, we will only summarize those possibilities and briefly describe the deficiencies of each.

We first considered the possibility of optimizing the current TENEX implementation to handle 1000 users. We can currently handle approximately 25-30 concurrent users. Optimization of existing code might conceivably increase this capacity by a factor of 2 to 3.

The TENEX software milieu levies a high cost for this application. Further improvement could be obtained by rewriting the timesharing

software, especially the file-handling and interprocess communications systems. Total improvement from optimization and software rewrite might be as much as a factor of 10. This falls short of a planned system capacity of 1000 concurrent users. It would also be very vulnerable (no reliability improvements), expensive (a dedicated PDP-10 with a lot of new software), and incapable of further expansion.

Other single system milieus seemed equally unattractive for the same reasons. We were thus led to consider multiprocessor systems. Such systems can be composed of hardware distributed over the Arpanet (RSEXEC) or locally distributed (Illiac, DCS, Pluribus).

We first considered multiprocessors distributed over the Arpanet. With any multiprocessor system there exists a synchronization problem with respect to shared data (the Dijkstra problem, etc.). For the WM this problem is particularly severe since the shared data base is extremely large and references to it are frequent. We devised techniques for cross-net synchronization [1], but it is an unfortunate fact of life that such synchronization is a multiple of network message times. Such times are measured in seconds instead of the microseconds of internal computer times.

We were thus led to the choice of either adding many seconds (for synchronization) to the time required to service a user request or else not synchronizing before servicing the request and occasionally being forced to rescind a previously granted request (e.g., a user puts away a file under some name and is told subsequently that the name had already been used). The first choice is clearly unacceptable (actual estimates of delays are about 30 seconds if we include the time needed to detect processor failure). The second choice is unpalatable, but marginally acceptable if there are no alternatives.

Notice also that multiprocessors distributed over the Arpanet do not, by their mere existence, solve the reliability problem. Additional software is required to provide system reliability. Further, distribution over the net only makes increased capacity possible; it does not make it cheaper.

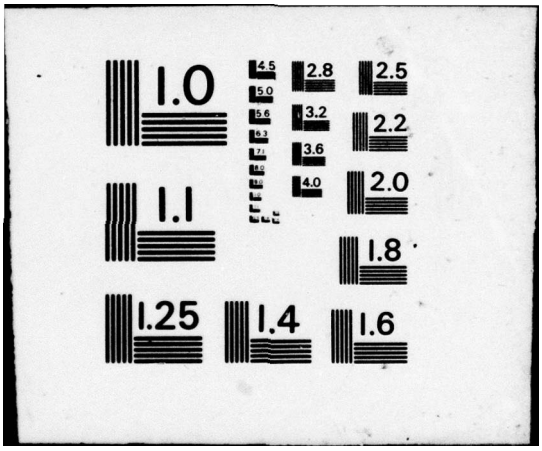
Thus, we next considered locally distributed multiprocessors. Illiac was rejected out of hand. The Distributed Computer System [2, 3, 4, 5, 6] was investigated. It suffers from at least two deficiencies. One, it was not designed for and has not been tested on the Arpanet. Two, it has been developed and supported by a university rather than industry. Hence, its continued support and development is problematic.

The fourth multiprocessor we considered was Pluribus [7, 8, 9, 10, 11]. We found that Pluribus seemed able to provide the capacity we needed, fail-soft capability had been considered part of the Pluribus problem from the beginning, the hardware was operational and installed on the Arpanet, and it had been developed by industry under an Arpa contract.

AD-A034 133 MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD F/G 9/2
NATIONAL SOFTWARE WORKS.(U)
SEP 76 R MILLSTEIN F30602-76-C-0094
UNCLASSIFIED CADD-7603-0411 RADC-TR-76-276-VOL-1 NL

3 OF 3
AD A034133

END
DATE FILMED
2-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

A Pluribus with approximately 10 Sue processors, the required memory, interfaces etc., would have the necessary processing capability combined elegantly with a significant improvement in reliability, at considerably less cost than a PDP 10. In the remainder of this report we discuss: the main featues of Pluribus and how they relate to the Works Manager task; the capacity requirements of the Works Manager; a Pluribus configuration that would satisfy these requirements; some potential difficulties; our conclusions and suggestions for future tasks to undertake; supporting notes and rough calculations for the estimates made.

IV Pluribus

Major Attributes of Pluribus

(1) Appropriateness

Pluribus is a multiprocessor develcped for communications networks. It employs a number of independent processors with independent control streams in a cooperative and equal fashion. Its effectiveness in a given problem depends upon the ease with which the solution can utilize concurrent processing capability. A Pluribus packet-switching store and forward communications processor (the IMP) is very cost effective because packets of data can be treated independently of one another. The NSW Works Manager problem has similar potential, because requests to the Works Manager, generated by a large number of concurrent users (approximately 1000) on geographically distributed hosts, lend themselves to the same treatment.

(2) Reliability

A main goal of Pluribus is to provide hardware which can be exploited by the program to survive the failure of any individual component. To this end all processors are treated as equal units and all jobs are treated uniformly. A processor is viewed as a resource to advance the algorithm and individual processor identification is irrelevant. A Pluribus configuration includes at least two of every vital hardware resource. Both hardware and software organization reflect a deep concern with reliability. Note that Pluribus is not a 'non-failing' device. Rather, it recuperates automatically within seconds or at most minutes, following a failure. The system survives not only transient failures but also solid failures of any single component. This approach works best in a situation where the system is embedded in a larger system that has the ability to resend lost or destroyed messages and resupply status information. This is the case with a Pluribus IMP on the Arpanet; it would also be the case with a Pluribus Works Manager in NSW.

Such a Works Manager would be extremely reliable, compared with a TENEX implementation. Efforts at providing this degree of reliability on conventional equipment by the employment of several Works Managers on dedicated computers at different sites would prove both costly and difficult. The difficulty arises in connection with the problem of keeping two separate Works Managers in synchronization. System degradation due to transmission delays of synchronization messages would be significant.

If protection from a local disaster (e.g., regional power failure) is regarded as an NSW imperative, it would be necessary to have two geographically separate Works Managers. Pluribus would still be a cost effective hardware basis for this situation, but would certainly suffer from degradation due to the need to transmit synchronization messages between the two Pluribus sites.

(3) Capacity

The Pluribus approach relates capacity and reliability. Adding additional copies of scarce resources (such as processors) not only makes the system more reliable, but increases its processing capacity in a natural way. Thus, the Pluribus approach differs from 'reliability by replication' which exploits the extra hardware only for error checking and substitution in the case of failure, but not for increased processing bandwidth when all is going well.

(4) Expandability

The approach (to capacity and reliability) also permits graceful expansion of the system. The addition of more processors, memory and input/output can be done without significant alteration of software, and without bringing the whole system down for a long time. Thus the Works Manager implementation can be realized initially on a smaller hardware configuration and be expanded as the NSW user population grows.

(5) Resource Locking

Pluribus has taken a basic approach to the locking of critical resources. This is the mechanism by which the algorithm enforces sequentiality when it is needed. An example of a lockable resource is the queue of free buffers. The system uses an uninterruptable load and clear operation as its primitive locking facility.

The Works Manager requires such a locking mechanism in a variety of circumstances. The Works Manager information retrieval subsystem makes heavy use of interlocks to properly interlace read and update activities in its data-base. Conventional hardware/software milieus make these interlocks slow and costly. Pluribus provides an interlock capability that is extremely fast and cheap.

(6) File Handling

Pluribus provides no software associated with file handling or secondary storage management. Since the Works Manager must do a considerable amount of just this, it would appear at first glance that a conventional hardware/software milieu, such as TENEX, has an advantage in this area. On the contrary, the capacity of the Works Manager to handle a large number of concurrent users will depend critically on efficient use of the disk. Conventional file directory structures and disk management software make it more difficult to attain this objective.

Pluribus Configuration for NSW Works Manager

The task of configuring a Pluribus for the NSW Works Manager is now under investigation. A preliminary study of what will be needed has been made to obtain upper bounds on the cost of this approach.

(1) Processors

On the basis of current understanding of the Works Manager task, rough calculations (see notes) suggest that 5 Sue CPUs would provide adequate processing capacity. Given that a processor is cheap (the processor itself is appr. $600; with 4k local memory + power, rack etc. it is around $3000), it seems reasonable to plan on eventually having a 10 CPU configuration.

(2) Shared Memory

High speed memory requirements for the Works Manager depend in part on intelligent use of secondary storage. For the moment we assume that the bulk of status information pertaining to each active user must be stored in fast memory. For 1000 users this requires a lot of memory; we estimate 1/2 million words of shared memory for the Pluribus (a word is 16 bits). This memory plus necessary busses and bus couplers account for a significant part of the whole configuration. Further investigation may show that disk storage can reduce this cost. In any case we should add memory gradually as the actual number of NSW users grows.

(3) Secondary Storage

Secondary storage for the system would be provided by disk, most likely of the IBM 3330 type. Two controllers on different i/o busses, each with 2 to 3 drives, would provide the capacity and reliability required. In the initial configuration each controller should have only one drive.

(4) Cost Estimate

The cost for a Pluribus with 20 CPUs, 1/2 million words of shared memory and 2 i/o busses is estimated at approximately $450,000. This figure is very conservative in several respects:

(a) 20 CPUs is twice our already conservative estimate of 10 CPUs for the final system.

(b) The cost of 1/2 million words of shared memory is based on the memory modules currently used in Pluribus configurations. These come in 8k units that require 3 cards on the memory bus. Considerably cheaper approaches are available but have not yet been explored.

(c) A reduction in the number of CPU's and the number of memory bus cards (and hence memory busses) would reduce the costs associated with busses and bus couplers.

In addition to the above hardware, additional costs would be incurred for the required disk interfaces (2) and for the two disk systems.

Potential Difficulties with a Pluribus Implementation

(1) New Hardware/Software

Pluribus is a new system. The first machine has been running approximately two years. The reliability concepts have not yet been thoroughly verified by experience. Some of the reliability software has not been tested.

(2) Disk Storage

No existing Pluribus configuration has disk storage. In principle disks should not create any new problems. However, the transmission rate of a suitable disk (4 megabaud) is higher than the transmission rate of any i/o device that has been utilized in Pluribus so far (1.5 megabaud).

An interface with adequate buffering would have to be designed and built. A rough time estimate for this is approximately three man months at a cost of $20,000.

(3) Software Aids

There are only minimal software aids available for program construction. All Pluribus programming is now done in assembly language.

(4) Applications Programming

The Pluribus approach to reliability and robust software requires that algorithms be organized into 'ribbons' and 'strips'. A ribbon can be thought of as a serial process. It is divided into strips with the requirement that no strip ever take longer than a certain time to execute. This maximal strip time is determined by the application and is related to the maximum time that can elapse before the most demanding i/o interface is serviced.

This approach does away with many of the problems associated with interrupt handling of input/output. It also encourages program design which is understandable and programs which are predictable, with firm commitments for the maximum time processes may take.

However, it must also be recognized that algorithms are not usually organized in this way and programmers are not accustomed to this idea. Some additional programming time may be required as a result.

There is some software already developed as part of the Pluribus concept, namely the Reliability Software, which could be used with the WM application software.

Conclusions

Despite the potential difficulties noted in the preceding section, we believe that Pluribus provides the best environment for a Works Manager implementation offering large scale, expandability, and reliability. Such an environment will be of relatively low cost and acceptable risk.

We believe that the following tasks should be undertaken in order to continue the investigation. These tasks are listed in the approximate chronological order in which they should be begun. We suggest that Task 6 should begin no later than 1 October 1976 in order to ensure continuity of NSW effort.

1. Definition of plausible user scenarios and associated WM actions in order to refine Pluribus sizing specifications.
2. Investigation of possible Pluribus disks.
3. Investigation of possible Pluribus memories.
4. Redesign of WM code with respect to recoding in ribbons and strips.
5. Study of IMP, WM coexistence on a single Pluribus.
6. Procurement of a suitable Pluribus.
7. Recoding of WM.
8. Debugging of Pluribus-based NSW.

Notes

In what follows we use some abbreviations:

FE = Front End
WM = Works Manager
FP = File Package
IR = Works Manager information retrieval system

For the purpose of making a rough estimate of the work load placed on the WM by 1000 users, the following scenario will be assumed to describe an average user.

(1) The user logs in (< 1 minute)

The FE, in order to process login, calls the WM. A message is sent from the FE to the WM including such information as project identification, node name and password. The WM must verify the login request, using the IR to identify user, establish user rights and so on. The WM generates a user id and stores information in the WM user status table. The WM sends a message to the FE. The message includes the assigned user id. The WM must also update the history file.

(2) The user edits a program file (5 minutes)

(a) The FE calls the WM in order to run a tool (an editor). The WM must check the rights of the user with respect to this tool. This may involve the use of the IR. The WM gets the tool descriptor, using the IR. The WM locates a suitable host to run the tool and sends a message to that host. The response provides the name of the tool instance which is then sent to the user FE. The WM must also update the history file.

(b) The tool (editor) calls the WM to obtain a file. The WM must check the rights of the user with respect to this file (this may use the IR). The WM uses the IR to determine an unambiguous file name. (Bad names require more work; we assume that they are the exception, rather than the rule). The WM sends a message to the FP which produces a copy of the file on the same host as the tool. (This requires transmission of the file over the network if the file is not already at the tool host site). The FP sends to the WM the new local name for the file. The WM then updates the history file (possibly using the IR) and then sends a message giving the local file name to the tool.

(c) The tool (editor) calls the WM to put the result file away. We assume that this is considerably less costly than step 2b. (No file transmission is necessary at this time.)

(3) The user compiles/assembles the edited program (5 minutes)

(a) Similar to step 2a where the tool now is a compiler

(b) Similar to step 2b where the tool now is a compiler

(c) Similar to step 2c where the tool now is a compiler

(4) The user link-edits in preparation for debugging session (5 minutes)

(a) Similar to step 2a where the tool now is a link editor.

(b) Obtain N files, where each file requires the activities described in 2b above. Each file is a relocatable file, which we assume consists, on the average, of 1000 lines x 5 bytes/line = 5000 bytes, plus a symbol table of 100 entries x 10 bytes/entry = 1000 bytes. This is a total of 6000 bytes per file. Any file not local to the link-editor will have to be transmitted over the network.

(c) Put the result file away, as in 2c above. This is the load module for a debugging run.

(5) The user debugs the program (60 minutes)

(a) As in 2a above, with DDT being the tool.

(b) Obtain file, as in 2b above. This file is the load module.

(c) Debugging session. This may use an output file for DDT messages. The program being debugged may also use some files. We assume here that these files are not NSW files.

After step 5 the user returns to step 2 (the user edits a file) to correct errors uncovered during the DDT session and to try again.

(6) The user logs out (< 1 minute)

We assume this is the same in terms of requirements as login (step 1 above).

The dominant component in terms of network load and Works Manager load is determined by the edit-compile-linkedit-debug cycle. Within this, the following interchanges with the WM are required:

edit	6
compile	6
link-edit	4 + 2 x N
debug	6

We assume that N, the number of relocatable files to be linked, is, on the average, 15. Thus the link-editor accounts for 34 interchanges with the WM (out of a total of 52).

The interchange rate is 52/75 per minute, or approximately 2 every 3 minutes of .011 per second. For 1000 users, this results in 11 interchanges per second with the works manager.

Since this also exercises the WM information retrieval subsystem, it seems reasonable to estimate capacity requirements in terms of the rate of file requests. Therefore, assume all interchanges are about files. As described in 2b above, there are 2 interchanges per file. Thus there are 5.5 file requests per second. If each file is 6000 bytes (as in 4b) and a file is never on the right host, transmission requirements on the net (as a whole) would be 264 Kilobaud. The pattern described would tend to guarantee that most of the relocatable files input to the link-editor did not change during an iteration of the edit-debug cycle. Thus, if we imagine that at the beginning of a session (day) all files have to be moved, but thereafter only a few, we see that the 264 KB rate must be high by a factor of 4 or 5.

In accordance with the description in 2b above (The user edits a file), we can estimate the number of disk accesses made by the WM as a result of using the IR. The rate of file requests has been extimated at 5.5/sec. This requires use of the IR at least once to check the file name for ambiguity), possibly another time (to check the user's rights with respect to the file) and possibly a third time (to update the history file). The number of accesses to the disk made by the IR is roughly a function of the number of descriptors (plus one). If we assume:

name check:	3 descriptors
user rights:	2 descriptors
history file:	2 descriptors

we see that 10 accesses are required per file request. On this basis the load on the disk storage system is approximately 55 accesses/second.

A disk controller with 2 to 3 drives (capable of independent seeks) and modestly intelligent disk software could provide the requisite capacity.

If we assume that the disk system is providing data at close to transmission speed (4 megabaud) degraded only by rotational latency (12.5 milliseconds), with 8 pages per track, actual transmission would be 625 K bytes x .2 = 125 K bytes/second. If we assume that, typically, 2 bytes are handled together in an instruction loop comprised of 10 instructions, the system must execute .5 x 10 x 125 k = 625 K instructions per second.

This estimate does not include:

(1) Whatever is required for the operation of a second disk controller. This system would be intended as backup. It could function in several ways:

   (a) all write operations to the primary disk system would be replicated on the second system. If the primary fails, then the secondary is switched in; when the primary comes back up it is brought into agreement with the secondary and their roles are switched.

   (b) all operations are replicated for both disk systems. Consistency checks are performed at various points.

(2) Other reliability and maintenance activities.

(3) Extra capacity for burst load situations.

(4) Extra capacity for disk optimization with respect to sector positioning (rotational latency). This would in effect increase realized disk transmission speed, thus requiring more processing capacity and would also make more critical fast processor response to disk i/o.

We assume these four taken together would require another factor of 2 in instruction execution rate:

instruction rate = 2 x 625 K = 1.25 M instructions/sec.

With a typical Sue instruction taking 4 microseconds, 5 processors would be required to sustain this execution rate.

References

1) Lamport, L. Distributed multiprocess systems without central control. In preparation.
2) Farber, D. J. The design of the distributed computer system.
3) Farber, D. J. The status of the distributed computer system.
4) Farber, D. J. and Larson, K. The system architecture of the distributed computer system, presented at Polytechnic Institute of Brooklyn Symposium on Computer Networks.
5) ---, Progress report on the distributed computer system. April 1972.
6) Farber, D. J. and Larson, K. The structure of a distributed computer system, presented at Polytechnic Institute of Brooklyn Symposium on Computer Networks.
7) Heart, F. E., Ornstein, S. M., Crowther, W. R., and Barker, W. B. A new minicomputer/multiprocessor for the ARPA network, Proceedings of the National Computer Conference, 1973, pp. 529-537.
8) Ornstein, S. M., Barker, W. B., Bressler, R. D., Crowther, W. R., Heart, F. E., Kraley, M. F., Michel, A., and Thrope, M. J. The BBN multiprocessor, Computer Nets Supplement to the Seventh Hawaii International Conference on System Sciences, January 1974.
9) Ornstein, S. M., Crowther, W. R., Kraley, M. F., Bressler, R. D., Michel, A., and Heart, F. E. Pluribus -- a reliable multiprocessor, Proceedings of the National Computer Conference, 1975, pp. 551-559.
10) Bressler, R. D., Kraley, M. F., and Michel, A. Pluribus: a multiprocessor for communications networks, Computing in the Mid-70's: An Assessment, June 19, 1975.
11) Private communication with Kraley, M. F., and Crowther, W. R.

Chapter 7: Management Tools

1. Overview

It is our object to create a set of tools for the support of management in its control and tracking of the activities of groups of NSW users. Specifically, we intend to support the following management functions:

Authorization

Access to the resources of the NSW is effected by "logging in" (supplying user's name and password). The system attempts to match this log-in information against a set of authorized user descriptions. If a match is found, the log-in is accepted, and that user's activities during the session are precisely restricted to the authorizations found in his description. The tools required here are those which support the manager in the creation and deletion of authorized user descriptions.

Scheduling

Managers frequently find it useful to create and manipulate documents which record expectations. Such documents include predictions of the progress of work and budgetary plans. No matter what is being predicted -- program creation or dollar evaporation --, we use the word "schedule" to designate any document which maps anticipated change in some variable against time. The tools here are those which support the manager in the creation of schedules.

Projection

A principal use of schedules is projection: the prediction of values for the variables mentioned in the schedules at some given time in the future. The tool to be supplied is a projector which, given a schedule and a date, automatically produces an estimate of variable values on that date, as predicted in the schedule.

Tracking of machine-observable activity

Managers may wish to obtain reports of what has actually happened. A primary source of this information is the history file, a journal whose entries fully describe every access to system resources permitted by the Works Manager (the NSW monitor) and include all cost-incurred information. The history file is to be archived and a given entry will remain available on-line for a considerable time after it has been created. The tools required here are those to support the manager in creating useful status reports from the history file.

Tracking of human-reportable activity

Machine-observable activity (resource access) is not the only source of management information as to progress. The other major source is assertions by human subordinates: which phase of a lengthy process is presently being worked on, what is the current percentage of completion of some job, what is the current best estimate as to when something will be finished, and so on. The tools required here are those which support the manager in defining the form of reports he wishes to receive and those which support the subordinate in satisfying his manager's reporting requirements.

Comparison of projection with actuality

Managers may wish to compare status reports (derived from either the history file or assertions by subordinates) with projections. The tools required here are those which assist him in bringing these into relation, either displaying them "side-by-side" or isolating potentially dangerous disagreements.

Automation of routine report creation

Managers may wish to create certain reports -- specifically, status reports derived from the history file, projections, and comparisons (exception reports) -- on a regular basis, without having to bother to log in and use the appropriate tools in a perfectly routine way. The new tools required here are those which support him in defining a regular schedule of such routine activities and cause the NSW to carry out his instructions automatically.

Automation of routine management strategies

A primary function of a manager is to close the loop between tracking and authorization. That is to say, the manager observes what has happened (status reports), and responds to what he has learned by changing authorizations. Thus, if an item in the budget has been over-run, he may restrict some or all of the activities of certain users. If a certain report from a subordinate has not been received by a certain time, he may restrict that subordinate's activity until the report is submitted.

Similarly, a manager may act to close the loop between tracking and scheduling: an observed violation of some schedule may lead him to produce an altered schedule.

Certain of these loop-closing activities may be routine in nature. The manager can decide in advance when to check for some situation, and what action to take if it arises. In principle, tools can be built which support him in the specification of those routine acts of management and cause the NSW to perform them automatically.

It is not our intention to give general support to management in the automation of routine strategies, not because tools are unbuildable, but because they are likely to be unusable: the class of all possible strategies is hard to represent in anything less than a full programming language.

Within the scope of this proposal, we intend to give only limited support for routine strategy automation: specifically, we will supply tools to support him in creating automatically implemented rules for at least:

1. Budgetary enforcement: regular comparisons of costs incurred with projections, with violations resulting in limitation of already-given authorizations.

2. Reporting schedule enforcement: regular verification of the timely submission of reports from subordinates, with violations resulting in limitation of the authorization of subordinates until the missing reports have been submitted.

2. Support Mechanisms

The remainder of this document describes a set of mechanisms -- tools and files -- to support the management functions outlined in the previous section. Our sole object is to explain more fully the nature of the support we have in mind and to lend credibility to our claim that such support is possible. To keep the relation between mechanisms and intent clear, we have described the mechanisms to support each management function as independently as possible. Since the mechanisms will not in fact be programmatically independent, the result is that the sequel is frankly false as a literal description of our intended program design; this is a natural by-product of the artificial separation of an integrated system into parts for ease of exposition.

2.1 Authorization

An authorized user description is a file which defines the recognition code (user name, password) for an accredited NSW user and specifies his rights of access to NSW facilities; these rights are represented as a list of authorizations. An authorization is best thought of, for present purposes, as a vector whose components correspond to:

1. Agent
2. WM procedure
3. Filespec

An authorization means: "This user may through the tool (component 1) call on the WM procedure (component 2) to operate on the portion of NSW file space defined by (component 3)." Example:

```
  ~  ,  COPY,      NSW.COMPASS
HENRY,  DELETE,    NSW.COMPASS.MASTER
  ~  ,  RUNTOOL,   NSW.STDTOOLS
```

The user may, through any tool, call for a copy of any file in NSW.COMPASS or run any tool in NSW.STDTOOLS. The user may also delete any file in NSW.COMPASS.MASTER, but only through the tool HENRY as agent.

An authorized user description is created by use of a tool, the authorizer. Since the system guarantees that no two authorized user descriptions can refer to the same recognition code, the relation "A created B's authorized user description" induces a partial ordering (a tree structure) over the set of authorized user descriptions and thus over the set of users.

The set of users whose authorized user descriptions A has created are A's immediate subordinates; the set of users lying in the subtree rooted at A (save A himself) are A's subordinates.

When A creates an immediate subordinate B, he can determine whether B can develop subordinates by granting or denying B access to the authorizer.

The authorizer limits the authorizations which a manager can give to a subordinate, by insisting that a manager can give a subobdinate no more authority than he himself possesses.

Change of an authorized user description can be performed only through use of the authorizer. A manager may so use the authorizer on the authorized user descriptions of his immediate subordinates only. However, the authorizer will automatically restrict authorizations of the immediate subordinate's subordinates as needed to preserve consistency with the rule that no user has more authority than his manager.

2.2 Scheduling

By a schedule we mean a two-dimensional structure whose columns are labeled with dates and whose rows correspond to variables whose predicted values as a function of time are recorded in the entries of the schedule.

The variables may be of several types. A type determines the units of measure (integers, dollars, etc.), constraints on the sequence of values in its row (e.g., monotonically increasing), and whether the entries are intended as lower or upper limits for the variables. For example:

a. Expenditures -- dollar-valued, monotonically increasing, upper limit

b. Funds Remaining -- dollar-valued, monotonically decreasing, lower limit

c. Percent Completion -- integer-valued, monotonically increasing, lower limit

Here is a sample schedule:

	Type	1 July 78	1 Oct 78	1 Jan 79
TECO-USE	EX	$5000	$10000	$10000
NLS-USE	EX	$10000	$20000	$30000
SMITH	EX	$1000	$2000	$3000
JONES	EX	$0	$8000	$16000
PARSER	PC	20%	50%	100%
USER-GUIDE	PC	0%	20%	50%
TOTAL-PROJECT-REM	FR	$150000	$100000	$50000

By a scheduling tool we mean simply a specialized editor which supports the manager in the creation of a schedule. Such a tool asks the manager to supply critical dates of interest (column headings) and the names and types of the variables he wishes to include as rows, and supports him in filling in the entries consistently.

[It may be better packaging to have different kinds of schedules, discriminated by the type of variable they contain, a percent completion schedule, an expenditures schedule, etc. In this event, the scheduling tool would, in effect, become several tools, each specialized to handle one type of variable.]

2.3 Projection

By a projection we mean the one-dimensional structure which results from applying a date to a schedule. Each entry in the projection corresponds to a variable in the schedule, and is composed of a value for that variable preceded by either of two signs: <= or >=.

A projection is created from a schedule and a date by a tool: the projector. The rule for generating an entry depends on the variable type -- specifically on whether it is an upper or a lower limit.

1. If it is a lower limit, the >= is generated. If it is an upper limit, the <= is generated.

2. If the date coincides exactly with a column heading of the schedule, the value in that column is used for projection. If the date falls between two columns of the schedule, the two "bracketing values" are examined. The larger is used if the variable is an upper limit; the smaller, if it is lower limit.

   lower limit: smaller value, >=
   upper limit: larger value, <=

   Here is the projection derived from the schedule given above for the date 1 Nov. 78:

TECO-USE	<=	$10000
NLS-USE	<=	$30000
SMITH	<=	$3000
JONES	<=	$16000
PARSER	>=	50%
USER-GUIDE	>=	20%
TOTAL-PROJECT-REM	>=	$50000

2.4 Tracking of machine-observable activity

By an accounting report we mean a one-dimensional structure, exhibiting the values of variables, prepared for a manager from data recorded in the history file.

An accounting report is created by a tool called the accounting reporter, which requires three variables of call: a beginning date, an ending date, and the name of an accounting report definition. The first two variables isolate the portion of the history file to be inspected by the accounting reporter. The third determines the behavior of the accounting reporter (which is a "table-driven" program).

An accounting report definition describes a procedure for constructing an accounting report from a segment of the history file. It contains two kinds of information:

- Declarations of the (names and types of) variables to be included in the accounting report,
- Rules for mapping from the history file segment into values for the declared variables.

The different types of variables which may be declared will probably not be very numerous. The most interesting type is dollars.

The rules look much like the sorts of rule found in RPG definitions. They include:

- recognizers, which select a subset of the entries in the history file segment (e.g., all bills for TECO use);
- primitive counts, which collect numerical information from a selected subset into temporaries (e.g., sum the costs of all such bills);
- mapping program, which defines the map from the temporaries to the declared variables.

If the rules are permitted to be very general in form (e.g., arbitrary recognizers, arbitrary mapping programs), it becomes very difficult for a manager to define his rules correctly. For present purposes, and to show the power of a very restricted form, we assume:

- A recognizer is a vector of values (to be matched) and "don't cares" whose components match one-one with the fields of a history file entry. Thus, if tool cost is delivered as part of the ENDTOOL entry, a recognizer for all TECO bills would look like:

  ENDTOOL ~ ~ ~ TECO ~ ~ ~

  where ~ means "don't care".

- A primitive count is either a simple count of the number of entries in the subset, or a sum, over the subset, of the value in one field of the entry.

  Thus, if COST is a field name, the sum of all TECO bills is represented by the following recognizer, primitive count

  ENDTOOL ~ ~ ~ TECO ~ ~ ~ , COST

  The number of TECO uses is given by

  RUNTOOL ~ ~ ~ TECO ~ ~ ~ , #

- A mapping program is a sequence of assignment statements in constants, variables, primitive counts and local temporaries.

  Example:

  V1. TECO-COST-IN-STERLING

  V2. NET-SOS-PREFERENCE

  T1. RUNTOOL ~ ~ ~ SOS ~ ~ ~ , #
  T2. RUNTOOL ~ ~ ~ TECO ~ ~ ~ , #
  T3. ENDTOOL ~ ~ ~ TECO ~ ~ ~ , COST

  V1 <- T3/2.4

  V2 <- T1-T2

  Note: The search of the history file is always restricted to those entries associated with users on the subtree below the manager (including entries for the manager himself). A reserved string SUBS will match entries for subordinates only, excluding the manager himself. If the manager uses the name of a particular subordinate in the recognizer, that name matches not only that subordinate but his subordinates as well.

2.5 Tracking of human-reportable activity

By a staff report we mean a file prepared by a subordinate for delivery to his manager. A report consists primarily of a one-dimensional structure each element of which designates a value for some named variable. [In addition, every staff report includes, conventionally, the name of its author and the date of preparation.]

The varibles whose values may be given in a status report are of several types; for example:

a. status variables
b. percents
c. dates

When a subordinate wishes to prepare a staff report, he uses a tool called the staff reporter. This interactive tool collects variable values from the subordinate and assembles them into a staff report of a form previously defined by the manager. That is to say, the staff reporter is a "table-driven" tool, where the driving table (or file) includes a description of a particular form of staff report defined by the manager.

A staff report definition is a file used to drive the staff reporter. A staff report definition is created by a manager, and defines the form of some report he wishes from his subordinates.

The form of a staff report definition is as follows: it consists of a list of variable descriptions; each such description includes a variable name, a variable type, and an arbitrary character string. For variables of type status variable, a list of possible values is included in the staff report definition.

A staff report definition is prepared by the manager with a tool: the staff report definer. Here is an example of a staff report definition prepared with the aid of this tool:

Name	Type	String
ESTIMATED-COMPLETION-DATE	DATE	WHEN WILL THIS PROJECT BE FINISHED?
CUR-PHASE-PARSER	STATUS	IN WHAT PHASE IS THE PARSER?
LEX-DEBUGGING	PC	HOW DEBUGGED IS THE LEXICAL ANALYZER?

CUR-PHASE-PARSER: DESIGN, CODING, DEBUGGING, DOCUMENTATION

Once this staff report definition has been created, the subordinate who wishes to make his report calls the staff reporter, which asks him which definition to use. If the above sample definition is chosen, the staff reporter proceeds to use the strings in the definition to conduct the conversation, and the subordinate sees the questions:

```
WHEN WILL THIS PROJCT BE FINISHED?  (DATE):
IN WHAT PHASE IS THE PARSER?  (DESIGN, CODING,
    DEBUGGING, DOCUMENTATION):
HOW DEBUGGED IS THE LEXICAL ANALYZER?  (PC):
```

The staff reporter verifies that the answers are consistent with the variable types (i.e., with prompting information in parentheses) and assumes that the created staff report is well-formed.

2.6 Comparison of projection with actuality

By a comparison tool we mean a program which accepts two inputs:

- the name of a projection,
- a list of names of either accounting reports or staff reports.

A comparison tool uses the variable names found in the projection to select variable values from the accounting and staff reports. It then takes the relations found in the projection and the selected variable values and creates a new file.

There may be several comparison tools: one which simply produces a 2 x n structure, displaying the projected and actual values side by side; one which displays only violations of the projections; and perhaps others. For present purposes, we will give a name to but one comparison tool, the exception reporter, which is called an exception report, produces a file of which the following is an example:

Date: 1 Nov. 78

Variable Name	Actual	Projection
TECO-USE	$10500	<= $10000
NLS-USE	$3000	<= $30000
PARSER	45%	>= 50%
TOTAL-PROJECT-REM	$40000	>= $50000

2.7 Automation of routine report creation

An automated report schedule is a file (created by a manager with the support of a tool: the automated report scheduler) which specifies the manager's desires to have certain types of files prepared automatically on a routine basis. It consists of:

- a sequence of WM calls to be performed (drawn from a very limited set),
- a specification of the times when the sequence is to be performed.

Thus, by way of example:

Program:

```
RUNTOOL  ACCOUNTING-REPORTER (TOOL-EXPENDITURES, 1 JULY 76, NOW; TERPT)
RUNTOOL  PROJECTOR (TOOL-EXPENDITURE-SKED, NOW; TEPROJ)
RUNTOOL  EXCEPTION-REPORTER (TERPT, TEPROJ; TE-OVERRUNS)
DELETE   TERPROJ
```

```
Beginning Date:  1 August 76
Frequency:       Monthly
End Date:        1 July 77
```

This sample automated report schedule is intended to direct the NSW to execute the program portion on a monthly basis, from 1 August 76 through 1 July 77. Each execution will produce an accounting report and an exception report (the intermediate product -- projections -- is deleted after each execution). After each execution, there will exist in the NSW file system two new files, one with a name of the form TERPT.date and one with a name of the form TE-OVERRUNS.date.

[If we restrict the set of WM functions which can be called to the running of a few tools and file deletion, we can obviously make the form of the automated report schedule look a good deal simpler. We have chosen to give the example in the above form to suggest to the reader that we are dealing here with a special case of a general problem: the specification of a pattern of NSW use to be stored for later execution.]

2.8 Automation of routine management strategies

By an automated management schedule we mean an extension of the idea of an automated report schedule, which permits managers to schedule a general class of routine managment actions in advance. If we were to extend the form of an automated report schedule to include control statements and the possibility of specifying any WM procedures -- including those which alter authorizations in arbitrary ways --, the extended form could represent a very wide class of prespecified management strategies. Unfortunately, so general a form is both hard to write -- linguistically, it looks like a program -- and hard to verify for consistency with intent. Thus, we wish to proceed conservatively in this area. Initally we will restrict the set of authorization changes to a very simple population, as follows:

> Let us assume that the form of an authorization file is made slightly more complex, by permitting the managers to associate, with any single authorization, an arbitrary number of boolean variables. An authorization is ignored by the access control machinery of the WM if any of its associated boolean variables is FALSE. We will restrict the changes in authorization representable in an automated management schedule to changes in the values of these boolean variables.

Thus we imagine an automated management schedule to look very much like an automated report schedule, save that the set of forms possible in the program section is extended to include forms like:

IF <condition> THEN A,B,C <- TRUE: D,E <- FALSE

where A,B,C,D,E are the names of boolean variables. The forms possible for the type <condition> should at least include presence of a named variable on an exception report and time relations (e.g., NOW >= 1 June 77).

It should be noted that this mechanism is sufficient to provide budgetary enforcement in the sense that if some authorized activity (say, TECO use) is associated with a boolean variable Z and invoked/scheduled by a variable TECO-USE, then the statement:

IF TECO-USE IN TE-OVERRUNS THEN Z <- FALSE

blocks further budgetary overrun.

Moreover, since successful submission of a staff report is machine-detectable (entry by the staff reporter of a file of some recognizable name into NSW file space), presence or absence of anticipated staff reports can be defined as part of a regularly produced accounting report. If desired appearance of such reports is included in a schedule, dereliction will turn up in exception reports. Thus the same boolean variable mechanism can be used to turn off all authorizations (except the right to report) whenever a report is late; and to turn authorizations on again when the report appears.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.