AD-A035 087 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS F/G 9/2
MSG: THE INTERPROCESS COMMUNICATION FACILITY FOR THE NATIONAL S--ETC(U)
DEC 76 R H THOMAS, S C SCHAFFNER N00014-75-C-0773
UNCLASSIFIED BBN-3483-REV-1 NL

1 OF 2
AD
A035087

BOLT BERANEK AND NEWMAN INC
CONSULTING · DEVELOPMENT · RESEARCH

ADA035087

BBN Report No. 3483 December 1976

MSG: The Interprocess Communication Facility for the National Software Works

Robert H. Thomas
Stuart C. Schaffner

NSW Protocol Committee

Revised
December 23, 1976

DDC
RECEIVED
FEB 1 1977
A

This work was supported by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by Rome Air Development Center under contract number F30602-76-C-0094 and by the Office of Naval Research under contract number N00014-75-C-0773.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

MSG: The Interprocess Communication
Facility for the National Software Works

Revised

December 23, 1976

NSW Protocol Committee

Massachusetts Computer Associates Inc. Document No. CADD-7612-2411
Bolt Beranek and Newman Inc. Report No. 3483

This work was supported by the Advanced Research Projects Agency of the Department of Defense and monitored by Rome Air Development Center under contract number F30602-76-C-0094 and by the Office of Naval Research under contract number N00014-75-C-0773.

BOSTON WASHINGTON CHICAGO HOUSTON LOS ANGELES OXNARD SAN FRANCISCO

Document Revision History

Date	Comments
January 1976	Preliminary Specification of MSG.
December 1976	Resolves ambiguities in January 1976 version of the document; more completely specifies NSW conventions for use of MSG; includes implementation suggestions based on experience with implementations for ARPANET TENEX, Multics, and OS/360 hosts.

White Section
Buff Section
BY
DISTRIBUTION/AVAILABILITY CODES
AVAIL. and/or SPECIAL
A

The NSW Protocol Committee

The NSW Protocol Committee is an ad hoc group made up of representatives from Bolt Beranek and Newman Inc. (BBN) and Massachusetts Computer Associates Inc. (Compass). The committee members are (in alphabetical order) Paul Johnson (BBN), Robert Millstein (Compass), Stuart Schaffner (Compass), Richard Schantz (BBN), and Robert Thomas (BBN). The concepts embodied in this document are jointly the work of these five people. Special mention should go to Robert Thomas and Stuart Schaffner who wrote major portions of the document. Others contributing to the conceptualization of the MSG facility include Don Andrews (SRI), Robert Braden (UCLA), Kirk Sattley (Compass), Ken Victor (SRI), and Doug Wells (MIT).

Table of Contents

Section 1 - Introduction

1.1 Overview 1-1
1.2 NSW Components 1-2
1.3 Patterns of communication 1-3
1.4 Model of Communication 1-6
1.5 Modes of Communication 1-7
1.6 Sequencing of Messages 1-10
1.7 Host Incarnations 1-11
1.8 Organization of this document 1-12

Section 2 - MSG Process Environment

2.1 Hosts 2-2
2.2 Processes 2-3
2.3 Process names 2-4
2.4 Process addressing modes 2-5
2.5 Modes of information transfer 2-6
2.6 MSG primitive operations 2-8
2.7 Signals 2-16
2.8 Information transmittal 2-18
2.9 Sequencing of messages 2-20
2.10 Process creation and termination 2-23
2.11 Summary of terms 2-24

Section 3 - MSG-to-MSG Protocol

3.1 Transaction Identifiers 3-2
3.2 On the use of "source" and "destination" 3-3
3.3 MSG-to-MSG Protocol Items 3-4

Section 4 - MSG-to-MSG Protocol for the ARPANET

4.1 Implementation of MSG-to-MSG Paths by ARPANET Connections 4-1
4.2 Establishing the ARPANET Connections 4-2
4.3 Breaking the ARPANET Connections 4-4
4.4 Authentication of MSGs 4-5
4.5 Error Control for MSG-to-MSG Paths 4-9

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

5.1 General Format for MSG-to-MSG messages 5-2
5.2 Formats for Message Components 5-3
5.3 Identifying Transactions 5-6
5.4 MSG-to-MSG Protocol Messages 5-7
5.5 Summary of Commands 5-15

APPENDIX A - NSW MSG Scenario

APPENDIX B - MSG Implementation Notes

APPENDIX C - State Machine for MSG-to-MSG Protocol

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. As we have noted, 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.

1.3 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.

1.4 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 between some pair of processes.

A valid specific address refers to exactly one process and this address remains valid for the life of that process. Specific addressing may be used with all three communication modes. Specific addressing is used between processes which are familiar with each other. The familiarity is generally because the processes have communicated with each other before, either directly or through intermediary processes.

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.

In addition, the document includes three appendices. Appendix A is a scenario which illustrates how the NSW system uses MSG features to support user login and tool startup functions. It is included to help give the reader a better understanding of the MSG primitives. Appendix B addresses a number of implementation issues; it also specifies the NSW conventions for use of MSG, such as the generic names and failure codes defined for NSW. Finally, Appendix C presents a "state machine" description of the MSG-to-MSG protocol for message communication. This appendix provides a more detailed description of how the protocol specified in Sections 3 and 5 support message communication; it is primarily of interest to MSG implementers.

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.

2.1. 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 (1).

---

1. Guidelines for the selection of incarnation numbers by MSG are discussed in item 1 of Appendix B.

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 (2). A process name is always unambiguous; at all times it either corresponds to a single process or is invalid.

2. Guidelines for the selection of the specific designator component of MSG process names are discussed in item 2 of Appendix B.

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, but related to, 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 (3).

---

3. The selection of a destination process for a generically addressed message is discussed further in item 3 of Appendix B.

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 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 (4).

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 (5).

---

4. The NSW convention for signalling the completion of "send" primitives is presented in item 4 of Appendix B.

5. The function of the qwait parameter is discussed in more detail in item 3 of Appendix B.

3. ReceiveSpecificMessage(msgarea,srcnam,signal,disp,dt,sphndl)
   where
   msgarea points to an area of local memory in which MSG will put a message
   srcnam points to an area of local memory which MSG will set to the process name of the sender
   sphndl points to an area 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 an area 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 similar to ReceiveSpecificMessage except that here the message received was sent by a SendGenericMessage primitive rather than 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-PAIR(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 choose to return a host-dependent identifier for the connection.

   It is recommended that all hosts implement a mechanism for notifying a process in the event that a connection it has successfully opened is "spontaneously" closed (i.e., closed prior to execution of CloseConn by the process). One acceptable mechanism would be a "connection broken" signal which could be specified as an additional parameter to OpenConn.

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 or stream-marked messages to process pnam due to failure of a sequenced message transmission, then MSG will now stop doing so.

5. WhoAmI(pnam)
where
pnam points to an area of local memory

Execution of this primitive causes MSG to return the name of the executing process in the area pnam.

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 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. Software interrupt on channel n
   On TENEX, MSG generates an interrupt for the process on PSI (pseudo-interrupt) channel n when the event has occurred.

4. Flag plus software interrupt
   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-PAIR(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. If MSG is unable to deliver a stream-marked message from process A to process B, it will stop the flow of all (specifically addressed) messages from process A to process B until process A explicitly resynchronizes the message stream with process B. The Resynch primitive can be used to achieve the required resynchronization.

It is important to 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 (6):

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.

---

6. For further discussion of process creation see items 6 and 7 of Appendix B.

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 to 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.

1. 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 (stream) 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 defined as part of MSG because they have proven useful in communication system implementations and provide for experimental extensibility.

NOP

This mesage 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 29 (decimal) = 35 (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 (as soon as it is convenient to do so) closes the send connection of the pair;

2. Upon receipt of an MSG-to-MSG CLOSE message an MSG is expected to: return a CLOSE message on the send connection of the pair (as soon as it is convenient to do so); close the receive connection which carried the message; 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 presents 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 31 (decimal) = 37 (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:    Ø  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
```

In this case, the sockets S1, S2, ..., SN are MSG ICP-from sockets (which serve to uniquely specify the sockets for the MSG-to-MSG connections).

There is a cost associated with this authentication procedure which should be slightly less than that associated with an ICP exchange to establish MSG-to-MSG connections. It is important to point out that authentication is not required each time a message is sent from one host to another; rather, it is required only when an MSG establishes connections with another MSG. We expect that a pair of MSGs will establish connections

and authenticate one another when there is reason for them to communicate and will keep those connections open for as long as they have reason to communicate. As noted in Section 4.3, no policy for closing MSG-to-MSG connections has been formulated.

It is important to point out the limitations of this procedure. It provides only for the authentication of remote MSGs and not for the authentication of remote processes which may be using a remote MSG. As long as MSG is dedicated to a single application, such as NSW, this limitation is not a problem. That is, a remote entity authenticated as an MSG is also authenticated as an NSW MSG. Therefore, after an MSG is authenticated it is safe to create processes and charge their resource consumption to the NSW. If an instance of MSG (7) is to be used to concurrently support other requirements along with NSW, additional authentication may be required. For example, this authentication may be needed to determine which application a process being created should be charged against or how the access of such a process should be controlled, etc. These considerations, while important, are beyond the scope of this document.

---

7. By an instance of MSG we mean here a collection of cooperating, intercommunicating MSG modules on a set of host computers. It is possible to imagine independent collections of MSG modules (i.e., separate MSG instances), each collection of which is dedicated to a single application. This could be accomplished by reserving a separate set of ICP contact and authentication sockets for each such MSG instance. This would enable several distinct MSG modules to co-exist on the same host computer without interfering with one another; each such module would be dedicated to a particular application. The additional authentication discussed above would be required when a single instance of MSG is to be used for several different applications concurrently.

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 acknowledement 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	host incarnation #	process instance #	count	string
2	2	2	1	count

Host is a 16 bit host address. It is the standard ARPANET host address of the host in question. 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 are defined in item 9 of Appendix B.

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

```
--------------------------------------------------
* host            * process      * count * string *
* incarnation # * instance # *        *        *
--------------------------------------------------
       2               2           1     count
```

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

```
-------------------------------------------
* host            * process      * generic *
* incarnation # * instance # * code      *
-------------------------------------------
       2               2            1
```

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

2. Host Incarnation #:

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

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

```
----------
* MSG id *
----------
    2
```

16 bit (2 byte) number.
Ø = unspecified (to be used in "destination-id" field when destination-id is unknown; e.g., in the first MESS message for a message transaction and in some CONNECTION-OPEN messages).

4. Alarm code

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

16 bit (2 byte) number.

5. Failure/Rejection codes

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

16 bit (2 byte) number.
Specific codes assigned for use within the NSW are specified in item 10 of Appendix B.

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 associcated 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	MESS	src-id	dst-id	First byte	Handling
2	1	2	2	1	1

src-proc	dst-proc	message
5+j	5+k	M

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) |
  bit 4 - MESS-HOLD is acceptable response (i.e., MESS is also implicit HOLD-OK) |
  bit 5 - non QWAIT (i.e., qwait is false). |
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	MESS-OK	src-id	src-proc	dst-proc
2	1	2	5+j	5+k

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

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	XMIT	src-id	dst-id	src-proc	dst-proc
2	1	2	2	5+j	5+k

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

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

length	ALARM	src-id	acode	src-proc	dst-proc
2	1	2	2	5+j	5+k

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

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

```
-----------------------------------------------------
* length * ALARM-OK * src-id * src-proc * dst-proc *
-----------------------------------------------------
    2          1         2        5+j        5+k
```

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

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

```
----------------------------------------------------------------
* length * ALARM-REJ * src-id * reason * src-proc * dst-proc *
----------------------------------------------------------------
    2          1          2        2        5+j        5+k
```

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 * CONN-OPEN * src-id * dst-id * conn-id * type
----------------------------------------------------------
    2          1          2        2        2        2

   ---------------------------------
   * socket * src-proc * dst-proc *
   ---------------------------------
       3        5+j        5+k
```

length = 24+j+k
CONN-OPEN = 20 (24 octal)

```
type:  bit 0 + size    - binary send/receive pair + size
       bit 1 + size    - binary send + size
       bit 2 + size    - binary receive + size
       bit 3           - Server TELNET
       bit 4           - User TELNET
   where "send", "receive", "server", and "user" describe the
   end of the connection at the MSG which is sending the
   CONNECTION-OPEN message.
socket: 32 bit socket number = N
   for:
       Binary Send:    N = odd = send socket
       Binary Receive: N = even = receive socket
       Teletype:       N = even = receive socket
                       (N+1 = odd = send socket)
       Binary Send/Receive pair: (same as Teletype)
```

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

```
------------------------------------------------------------
* length * CONN-CLOSE * src-id * dst-id * conn-id * reason
------------------------------------------------------------
    2          1          2        2         2         2

   -------------------------
   * src-proc * dst-proc *
   -------------------------
       5+j        5+k
```

length = 21+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 * CONN-REJ * src-id * dst-id * conn-id * reason
-----------------------------------------------------------------
    2         1         2        2        2         2

   -------------------------
   * src-proc * dst-proc *
   -------------------------
       5+j         5+k
```

length = 21+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
- no such connection

14. NOOP

```
-----------------
* length * NOOP *
-----------------
    2        1
```

length = 3
NOOP = 0 (0 octal)

15. ECHO(data byte)

```
-----------------------------
* length * ECHO * data byte *
-----------------------------
    2        1        1
```

length = 4
ECHO = 1 (1 octal)

16. ECHO-REPLY(data byte)

```
    -------------------------------------
    * length * ECHO-REPLY * data byte *
    -------------------------------------
        2           1            1
```

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

17. EXPERIMENTAL(command, length, data)

```
    ----------------------------------
    * length * EXP * command * data *
    ----------------------------------
        2       1       1       N
```

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

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

```
    ------------------------------------------------------------------
    * length * SEND-STATUS * src-id * src-proc * dst-proc *
    ------------------------------------------------------------------
        2           1           2        5+j        5+k
```

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

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

```
    ------------------------------------------------------------
    * length * STATUS-OK * src-id * src-proc * dst-proc
    ------------------------------------------------------------
        2          1          2        5+j        5+k

       -----------------
        * status bytes *
       -----------------
                N
```

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 * STATUS-REJ * src-id * reason * src-proc * dst-proc *
--------------------------------------------------------------------------
     2          1          2        2        5+j        5+k
```

length = 17+j+k
STATUS-REJ = 6 (6 octal)
reason = To be specified, but including:
   dst-process unknown

21. CLOSE(reason)

```
-------------------------------
* length * CLOSE * reason *
-------------------------------
     2        1        2
```

length = 5
CLOSE = 7 (7 octal)
reason = To be specified.

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

```
--------------------------------------------------------------------------
* length * SYNCH * sender # * receiver # * version # *  data  *
--------------------------------------------------------------------------
     2        1        2           2            2          N
```

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 * PTCL-ERR * error code *  bad message   *
-----------------------------------------------------
    2          1           2              N
```

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

5.5 Summary of Commands

Code Dec	Code Oct	Command	Length
0	0	NOOP	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	5
8	10	MESS	19+j+k+N
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	24+j+k
21	25	CONN-CLOSE	21+j+k
22	26	CONN-REJ	21+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.

APPENDIX A - NSW MSG Scenario

This appendix describes in general terms how MSG is used by NSW system components to support the NSW user login and tool startup functions. Some familiarity with the general NSW system structure is assumed (see Sections 1.2 and 1.3).

For this scenario it is assumed that an NSW process exists in the (local to the user) Front End machine which is dedicated to handling the user's terminal. It is also assumed that Works Manager command interpretation is performed by this FE process. The scenario begins at the point at which the FE process has been assigned to the user.

The FE process starts by prompting the user for login information. After accumulating the pertinent information, the FE process uses the generic addressing facility of MSG to send the login data to a Works Manager process. The WM process selected by MSG to receive the generically addressed message notes the full name of the FE process and attempts to verify the user's login parameters. (The name of the FE process is delivered by MSG to the WM process along with the message.) The WM process communicates the success or failure of the user login to the FE process using a specifically addressed message.

Assume a successful login. When the user tries to run a tool the FE process gathers the tool name along with any other pertinent information and sends the request to start the tool as a generically addressed message to a WM process. The request is actually an invocation of the WM "RUNTOOL" procedure on behalf of the NSW user. The receiving WM checks to see if the user is authorized to run the tool, and, if so, retrieves the tool descriptor from an internal WM data base. Based on tool descriptor information the WM formulates a message containing tool startup information and sends it as a generically addressed message to a Foreman process on the host which has been selected (by the WM) to run the tool. This information includes the MSG name for the user's FE process and the nature of the tool (e.g., encapsulated, split FE, etc.). The Foreman process creates an instance of the tool. Next, using a specifically addressed message, it informs the WM process that the tool has been started. The WM then replies to the user's FE process with a specifically addressed message which includes the MSG address of the Foreman process for the tool (which the WM obtained with the reply from the Foreman).

For the case in which the tool and FE communicate via MSG messages, the scenario is complete since both the FE and

tool/Foreman have each other's specific name and can send messages directly to one another. For the case in which the tool is encapsulated, or known to use ARPANET connections to the FE, the WM-to-Foreman message requesting tool creation indicates that a direct FE-to-tool/Foreman connection is needed. Since each knows the other's MSG name, the FE and tool/Foreman can use the MSG OpenConn primitive to establish a direct communication path. As soon as the OpenConn primitives complete, data can flow between the user and the tool.

The discussion above describes in general terms how a tool is started. The specific details of tool invocation and the establishment of tool-FE communication are defined in the NSW Foreman specification document.

APPENDIX B - Implementation Notes

This appendix considers a number of implementation issues regarding MSG. The issues discussed are among those that arose during the implementation of MSG for ARPANET TENEX, Multics, and OS/360 hosts. In addition, this appendix specifies a number of conventions adopted for NSW use of MSG.

1. On the Selection of Host Incarnation Numbers.

The purpose of host incarnation numbers is to provide a means for distinguishing whether a given process name (i.e., specific process address) refers to a process that may currently exist or to one that existed during a previous period of MSG service by the host computer in question; for example, to provide a way to differentiate between names for processes that existed on a particular host prior to a host (MSG) crash from those for processes existing on that host subsequent to a host (MSG) restart.

It is important that a host MSG not reuse the same 16 bit number for identifying its incarnations "very often". By "not reuse very often" we mean that a given incarnation number should not be reused until there no longer exist any processes which may have stored and intend to use a process name which includes that number. This criteria is difficult, if not impossible, to insure with absolute certainty. However, there are several approaches which approximate satisfying the criteria.

One approach is to maintain a file which holds the current incarnation number. Whenever MSG is restarted, as part of its initialization procedure, it reads the number from the file and increments it by one. The resulting number is then stored back into the file and also used as the incarnation number for the new period of MSG service. Assuming that MSG restarts once an hour, which is much more frequently than one would expect under normal circumstances, it would take more than 7 years for a given 16 bit number to be reused. Storing the incarnation number in a file is relatively safe since the file storage medium on most systems is relatively invulnerable to all but the most catastropic host system crashes.

Another approach would be to derive the 16 bit incarnation from a real time clock. Although many host clocks maintain time with more resolution than 16 bits, it is still possible to derive a 16 bit number which approximates the "non-reuse" criteria. For example, with the knowledge that a host (or MSG on that host) can never crash twice in one minute (ten minutes), the least significant bits of the time obtained from the clock could be

discarded to give a time in minutes (tens of minutes) and enough high order bits discarded to produce a 16 bit number that would not recycle for weeks (months).

2. On the Selection of the Specific Designator Component of Process Names.

The specific designator component of a process name is generated by the host MSG that is responsible for managing the process. It is important that distinct processes have distinct names. A way to insure this is to guarantee that a given specific designator is seldom or never reused for a given generic process class during a given host incarnation of MSG.

The following is one strategy for assigning specific process designators in a way that insures they are seldom reused during a given MSG incarnation: The 16 bit specific designator is divided into 2 parts: an n bit usage count (high order bits) + an m bit index. The index identifies an entry in a process descriptor table which is used by MSG to maintain information about the process. Given a process specific designator, MSG can access process information quickly and efficiently simply by masking the high order n bits to obtain the index. The usage count is incremented each time the entry is reallocated for a new process. Therefore, the same entry must be reused 2**n times before the same specific designator is generated; that is, at least 2**n processes which use that entry must be created and destroyed. In addition, process descriptor table entries can be allocated in a circular manner such that an entry will not be reallocated until all other (free) entries have been allocated. Finally, note that the MSG names of two different processes within the same incarnation match (note that these processes can not co-exist) only if both their specific designator and their generic names are the same. We believe that this strategy insures that this will seldom occur.

3. On the Delivery of Generically Addressed Messages

The rule given in Section 2.4 states that a process must have a ReceiveGenericMessage primitive pending in order to be selected for a generically addressed message. This rule requires some elaboration. A strict interpretation of this rule would suggest that a generically addressed message can not be successfully delivered if there is no "matching" process with an outstanding generic receive. Such a strict implementation of the rule would result in undesirable behavior in which the successful delivery of generic messages would depend upon the relative timing of send and receive primitives. Each MSG implementer should insure that such race conditions do not occur.

The intent of MSG with regard to generic messages is that there is an implicit ReceiveGenericMessage primitive outstanding at each host for each generic process class implemented by that host (so long as the host has sufficient resources to allocate a process to accept a generically addressed message). Precisely how this is accomplished will vary from implementation to implementation and may vary within implementations from generic class to generic class. In some situations it may be appropriate to queue a generically addressed message while a process is created and allocated to accept it; in others, it may be appropriate to always have a free process ready to accept such a message should one come; and, in still others, it may be appropriate to queue a message for delivery to an existing, but currently busy, process until the process becomes free. In the case of inter-host messages, a host MSG is free to accept a generically addressed message (i.e., reply to the MSG-to-MSG MESS message with a MESS-OK) before allocating a specific process to receive the message.

In some cases this liberal interpretation of the rule for handling generically addressed messages may cause a message to be queued for a long period. The intent of the qwait (=false) parameter of the SendGenericMessage primitive (see Section 2.6.1) is to prevent a receiving MSG from queuing a message, possibly for a very long time, until a process becomes available to accept it. The qwait parameter allows the sending process to force an immediate process allocation decision. The intent of the QWAIT special handling field in an MSG-to-MSG MESS message (see Sections 3.1 and 5.4) is to force an MSG to reject the message (i.e., to return a MESS-REJ) if there is not currently a local process with a matching receive pending. Because this intent is also subject to some interpretation, the MSG implementer has some freedom in implementing it. For example, it would be legitimate to accept a message if a matching receive primitive was not ready and waiting but a process could be allocated immediately to handle the message.

4. On Signalling the Completion of Send Primitives.

There are two "events" which are (potentially) of interest to a sending process: when MSG returns control from the primitive call and when MSG signals completion of the pending event associated with the call. In the case of the "unblock" signal (see Section 2.7) these two occur simultaneously. The intent of the MSG specification is to allow implementers some freedom in choosing when these two "events" should occur. For a send primitive, the earliest useful time to return control seems to be after MSG has determined the (syntactic) validity of the send parameters; the latest useful time seems to be when the message buffer is free to be reused by the process.

The most useful time to signal completion of the pending event associated with the primitive is when the receiving MSG agrees (or refuses) to accept the message for delivery to the destination process. One might argue that the most useful time to signal completion of a send pending event is when the receiving process reads the message. We agree that from the point of view of the sending process this would be a very useful time. However, we believe that MSG can not provide this signal by itself. We believe that such a signal can be reliably provided only by the receiving process itself by an acknowledging message. That is, there is no externally observable action which corresponds to a process "receiving" a message. Merely depositing the data in the address space of the process and signalling it is insufficient. Receipt must imply some action by the receiving process that involves internal process logic.

It is true that a signal with the success disposition does not necessarily guarantee that the destination process successfully received the message; similarly, it is also true that a signal with a failure disposition does not necessarily mean that the destination process did (will) not successfully receive the message. We believe that it will be unlikely that a message for which the success disposition has been signaled will not be successfully delivered; however, this can occur if the destination host crashes after accepting the message and before delivering it to the process, or if the destination process is buggy and does not read messages which MSG has accepted for it. We further believe that it will be extremely unlikely that MSG will signal a failure disposition for a message that has been (or will be) delivered successfully. Processes which require a higher degree of certainty regarding their message communication should use a process level acknowledgement protocol.

The NSW convention for signalling the completion of a send primitive is to deliver the signal after the message has been accepted (or rejected) by MSG. In the case of inter-host communication, the signal may only be delivered by the sending MSG after the receiving MSG acknowledges the message with a MESS-OK or a MESS-REJ message.

5. On Message Lengths

The format for MSG-to-MSG messages presented in Section 5.1 includes a 16 bit "length" field. This format is capable of supporting interprocess messages of up to approximately 2**16 (8 bit) bytes. The MSG philosophy is that messages will tend to be relatively short and that longer data transmissions should be accomplished using the direct connection facility (i.e., OpenConn, CloseConn). The initial implementation of MSG for the

NSW limits messages to 2**11 bytes. The size of this limit may be increased in the future.

6. Process Creation.

Section 2.10 specifies that MSG may create processes only to handle generically addressed messages or as part of system initialization. To facilitate system debugging and integration activities, it is useful to relax this constraint and to permit MSG to create processes at the request of users (i.e., implementers of NSW system components). This ability to create new processes which execute under the control of MSG affords users more flexibility as they debug and integrate system components.

This process "materialization" capability can be provided by including, as part of the MSG implementation for a host, an interactive module to be used to create new MSG processes. A user interacts with this module to specify the process(es) to be created. This process specification includes the generic class for the process and the core image file to be used to initialize the process address space, as well as certain host dependent MSG options such as a particular debugger for controlling the process. After the user completes the specification, the process is created, made known to MSG, and placed into execution under MSG control.

7. Process Debugging Aids

It is recommended that all MSG implementations include facilities for monitoring and debugging processes. Experience has shown these facilities to be invaluable in debugging communicating processes and in integrating NSW system components. The TENEX and 360/91 implementations support similar debugging aids. These aids can be accessed by typing a special character which activates a simple MSG command language interpreter. By interacting with the command language interpreter, the user can query the status of MSG and the processes currently managed by MSG. In addition, he can exercise control over the MSG processes.

Process status information currently reported includes the process name, the process state (e.g., program counter plus execution status), and any "pending events" created as the result of previously executed, but not yet completed, MSG primitives. A listing of the primitives most recently executed by MSG processes on that host can be obtained. For each primitive, it includes the particular primitive (e.g., ReceiveGenericMessage,

SendSpecificMessage), the name of the executing process, the name of the target process (if appropriate), the time the primitive was executed, the outcome of the primitive (i.e., success or failure plus reason), and other relevant information.

A user can start and stop processes (see item 6 above), as well as invoke debuggers for them. For example, whenever a process is started a user has the option of specifying a debugger. If no debugger is specified, the process is started normally. If one is specified, the debugger for the process is started and the user is free to set break points, etc. before placing the process into execution. Should the process subsequently terminate abnormally (e.g., because it executes an illegal instruction), MSG automatically activates the debugger for it if one has been specified. Otherwise, MSG simply reports the termination. In this case, the user can explicitly invoke a debugger for the process via an MSG command if he wishes to examine the process.

8. MSG and NSW System Initialization

A goal of the NSW system implementation is to be resilient to the failure of system components. One aspect of resilient behavior is continued operation in the presence of host system crashes. Another important aspect is the integration of crashed hosts back into the system after they have been restarted. To achieve this re-integration, certain NSW components resident on a host that has been restarted must perform special crash recovery procedures before they can resume their normal functions. For example, after its host restarts, the host Foreman should attempt to save user files that may have been trapped in tool workspaces when the host crashed.

For a component to execute its recovery procedure, it must first determine that its host has just restarted. The host MSG implementations can help here. When configuring an NSW system, the generic process names known to the system (e.g., Works Manager, Front End, etc.) must be declared. For the TENEX implementation of MSG, the declaration of a particular generic process class may specify that a process in the class should be started automatically and "signalled" whenever MSG restarts. (As far as NSW is concerned, an MSG restart is equivalent to a host restart.) If this specification is made, the TENEX MSG will, upon completion of its own initialization, create an instance of the process. It does this by simulating a message generically addressed for the process class. The process created to receive this message can detect that it is the "first" process in its class, and that it should execute its recovery procedure. (The simulated generically addressed message is specially marked as

from MSG rather than from some NSW process requesting service.)

9. Generic Names for NSW MSG

The NSW convention for process names is that upper and lower case alphabetic characters are equivalent in the string representation for the generic component of a process name (see Section 5.2.1).

Within the NSW the definition of a generic process class includes both a string name for the class and a (internal-to-MSG) generic code for the class (see Section 5.2.1).

The generic names that are currently defined for the NSW are:

String Name	Generic Code	Comment
FE	1	Front End Processes
WM	2	Works Manager Processes
FOREMAN	3	Foreman Processes
FLPKG	4	File Package Processes
NFLPKG	5	(Interim) File Module for WMO
IBS	6	Interactive Batch Submission Process
WMO	7	Works Manager Operator Process

10. Error Codes for NSW MSG

The following codes are defined for use in the "reason" field of MSG-to-MSG messages (see Section 5.4).

Code (Octal)	Meaning
140001	Command not implemented.
140002	Unknown command.
140003	Command syntax error.
140004	MSG message too long.
140005	Incompatible MSG-MSG protocol version # in SYNCH
140101	Destination process unknown.
140102	Destination process message queue full.
140103	Destination name/handling - generic/specific mismatch.
140104	Generic name not legal for destination process.
140105	Bad incarnation number on destination process.
140201	Source process name mal-formed.
140202	Message rescinded or timed out.
140301	Insufficient resources to complete command.
140401	Process not accepting alarms now.
140402	Alarm already queued for process.
140501	That generic class not supported here.

140502	Can't allocate a process for generic message.
140601	User process closed connection.
140602	ACK'ing your CONN-CLOSE.
140603	Received a CLS for connection.
140604	Invalid connection type.
140605	Remote process name mis-match.
140606	Referenced connection transaction does not exist
140607	Local process referenced does not exist.
140610	Connection ID inconsistent with local connection info.
140611	Connection type mis-match.
140612	Connection aborted.
140613	Connection byte size mis-match.
140614	Process at this end of connection died.
140615	Timed out waiting for your CONNECTION-OPEN.
140620	Exceptional condition detected during data transfer.

In addition to the above error codes used to convey information regarding error and failure situations between MSGs, error codes are required at the MSG-to-process interface. The following codes are used in the TENEX MSG implementation and, in the interest of standardization, are recommented to other implementers for use as error and failure codes for MSG primitives.

Code (Octal)	Meaning
100001	Signal given is invalid.
100002	Handling given is invalid.
100003	Process name given is invalid.
100004	Unable to access primitive parameters
100005	Primitive not implemented.
100006	Invalid host address in process name.
100007	Unable to return primitive return parameters
100101	Unable to message area for send primitive
100102	Message length invalid.
100103	Generic code malfunction.
100201	AlarmAccept parameter not a boolean.
100301	EnableAlarm already outstanding.
100401	Invalid Event Handle in Rescind primitive.
100402	Unable to Rescind.
140301	Insufficient resources.
140604	Invalid connection type.
100601	Already have connection of that ID.
140611	Connection type mis-match.
100602	Unknown connection.
100603	Connection not to process named.
100604	Remote site refused connection.
140605	Remote process name mis-match.

11. Socket Numbers

The ICP contact (see Section 4.2) for NSW MSG is 29 (decimal) = 35 (octal). The authentication socket (see Section 4.4) for NSW MSG is 31 (decimal) = 37 (octal).

12. MSG-to-MSG Protocol Version

The current version of the MSG-to-MSG protocol (see Sections 4.2 and 5.4.22) is 1.

APPENDIX C - State Machine for MSG-to-MSG Interprocess Messages

This appendix is a companion to Section 3.3.1. Its purpose is to describe in greater detail how the MSG-to-MSG protocol supports the exchange of interprocess messages. The description is in terms of how a host MSG responds to the various protocol messages from remote MSGs and to the execution of the send and receive primitives by local processes.

To perform its task an MSG must maintain information about the status of interprocess messages in transit between source and destination processes. In addition, it must maintain information about an expected, but not yet received, message when a process executes a receive and MSG has no message to deliver to the process. We introduce the notion of a Message Control Block (MCB), which an MSG uses to maintain such information, as a vehicle for describing the MSG-to-MSG protocol.

The protocol description in this appendix is in terms of how MCB states change in response to various events. Although the intent of this appendix is to clarify the semantics of the protocol, the appendix can also be regarded as the description of a prototypical MSG implementation. In this later regard, we make the obvious comment that there are a variety of other equally valid implementation approaches.

An MCB is assumed to contain the following fields (some of which might be empty):

- MCB state
- source process
- destination process
- source transaction ID
- destination transaction ID
- message data

If MCBs were to be the basis of an implementation, other fields might be included such as the signal to be used to notify the local process that the associated primitive has completed, the location within the address space of the local process where the message is to be delivered, etc.

The remainder of this appendix describes the MCB states and the transitions between them.

Summary of MCB States

In the following, the term "header" refers to the information that must be remembered by an MSG in order to enable transmission of the message data to be requested later. The states and the transitions described below are illustrated in Figure C1.

State	Meaning
S1:	No Message Control Block (MCB) exists.
S2:	Send primitive executed by local process, MESS sent; Waiting for reply from remote MSG.
S3:	HOLD-OK sent; Waiting for reply from remote MSG.
S4:	XMIT received, MESS sent; Waiting for reply from remote MSG.
S5:	MESS received and accepted; Waiting execution of receive primitive by local process.
S6:	MESS received, Header accepted; Waiting execution of receive primitive by local process and HOLD-OK from remote MSG.
S7:	MESS received, Header accepted, HOLD-OK received; waiting execution of receive primitive by local process.
S8:	XMIT sent; Waiting MESS from remote MSG.
S9:	MESS received, header accepted, receive primitive executed; Waiting HOLD-OK from remote MSG.
S10:	Receive primitive executed by local process; Waiting MESS from remote MSG.
S11:	XMIT sent; Waiting MESS and execution of receive primitive by local process.

Note: After accepting a message header but not the message itself, an MSG waits for room to buffer the entire message. The buffer space could become available in a number of ways: for example, as the result of the execution of a receive primitive by a local process to which the message is addressed, as the result of the delivery of another message to another process, etc. We have simplified states S6 and S7 somewhat by considering only the case of waiting for the execution of a local receive primitive.

STATE TRANSITIONS

The transitions are described using the notation:

Transition-name: Event / MSG Actions / New State.

Transitions from:

S1: No MCB exists:

T1: local process does send / create MCB, send MESS to destination MSG / S2

T2: receive MESS / decide to reject message, send MESS-REJ / S1

T3: receive MESS / decide to accept message, create MCB, send MESS-OK / S5

T4: receive MESS / decide to accept header but not message, create MCB, send MESS-HOLD / S6

T5: local process does receive / create MCB / S10

S2: MESS sent; waiting for reply from remote MSG.

T6: receive MESS-OK / notify local sending process, delete MCB / S1
or
receive MESS-REJ / notify local sending process, delete MCB / S1
or
receive MESS-HOLD / decide to not buffer message, send MESS-CANCEL and notify local sending process, delete MCB / S1

or

timeout / decide to flush message, notify local sending process, delete MCB / S1

T7: receive MESS-HOLD / decide to buffer message, send HOLD-OK and notify local sending process / S3

S3: HOLD-OK sent; Waiting for reply from remote MSG.

T8: receive XMIT / send MESS / S4

T9: receive MESS-REJ / delete MCB / S1

or

timeout / decide to flush message, send MESS-CANCEL, delete MCB/ S1

S4: XMIT received, MESS sent; Waiting reply from remote MSG.

T1Ø: receive MESS-OK / delete MCB / S1

or

receive MESS-REJ / delete MCB / S1

or

timeout / decide to flush message, delete MCB / S1

or

receive MESS-HOLD / decide to buffer message no longer, send MESS-CANCEL, delete MCB / S1

T11: receive MESS-HOLD /decide to continue buffering, send HOLD-OK / S3

S5: MESS received and accepted; waiting receive by local process

T12: local process does receive / deliver message and notify process, delete MCB / S1

or

timeout / decide to flush message, delete MCB / S1

S6: MESS received, Header accepted; Waiting receive by local process and HOLD-OK from remote MSG.

T13: receive HOLD-OK / none / S7

T14: receive MESS-CANCEL / delete MCB / S1
or
timeout / decide to flush Header, send MESS-REJ, delete MCB / S1

T15: local process does receive / none / S9

S7: MESS received, Header accepted, HOLD-OK received; Waiting receive by local process.

T16: local process does receive / send XMIT / S8

T17: receive MESS-CANCEL / delete MCB / S1
or
timeout / decide to flush Header, send MESS-REJ, delete MCB / S1

S8: XMIT sent; Waiting MESS from remote MSG.

T18: receive MESS / send MESS-OK, deliver message and notify local receiving process, delete MCB / S1
or
timeout / decide to flush Header, notify local receiving process, delete MCB / S1

T19: receive MESS-CANCEL / clear source info in MCB / S10
or
timeout / decide to flush Header, clear source info in MCB / S10

T20: receive MESS from another source / send MESS-OK to that source, deliver message and notify process / S11

[Note: the timeout in T18 is with respect to the receive initiated by the local receiving process; the timeout in T19 is with respect to the interactions with the remote MSG.]

S9: MESS received, Header accepted, receive done by local process; Waiting HOLD-OK from remote MSG.

T21: receive HOLD-OK / send XMIT / S8

T22: receive MESS-CANCEL / clear source info in MCB / S10

State Machine for MSG-to-MSG Protocol

T23: timeout / decide to flush Header, send MESS-REJ, notify local receiving process, delete MCB / S1

T24: receive MESS from another source / send MESS-OK to that source, deliver message and notify local receiving process / S6

[Note: In this situation, MSG might choose to buffer the new message from the other source until the disposition of the original message is resolved. An MSG making this decision would simply create a new MCB for the new message via transition T3 or T4.]

S10: receive done by local process; Waiting MESS from remote MSG.

T25: receive MESS / send MESS-OK, deliver message and notify local receiving process, delete MCB / S1

or

timeout / decide to abort receive, notify local receiving process, delete MCB / S1

S11: XMIT sent; Waiting MESS from remote MSG and receive by local process.

T26: local process does receive / none / S8

T27: receive MESS / decide to buffer only header, send MESS-HOLD / S6

T28: receive MESS / decide to accept message, send MESS-OK / S5

T29: receive MESS-CANCEL / delete MCB / S1

or

timeout / decide to flush Header, delete MCB / S1

or

receive MESS / decide to buffer neither message nor header, send MESS-REJ, delete MCB / S1

AD-A035 087 BOLT BERANEK AND NEWMAN INC CAMBRIDGE MASS F/G 9/2
MSG: THE INTERPROCESS COMMUNICATION FACILITY FOR THE NATIONAL S--ETC(U)
DEC 76 R H THOMAS, S C SCHAFFNER N00014-75-C-0773

UNCLASSIFIED BBN-3483-REV-1 NL

2 OF 2
AD
A035087

END
DATE
FILMED
3-77

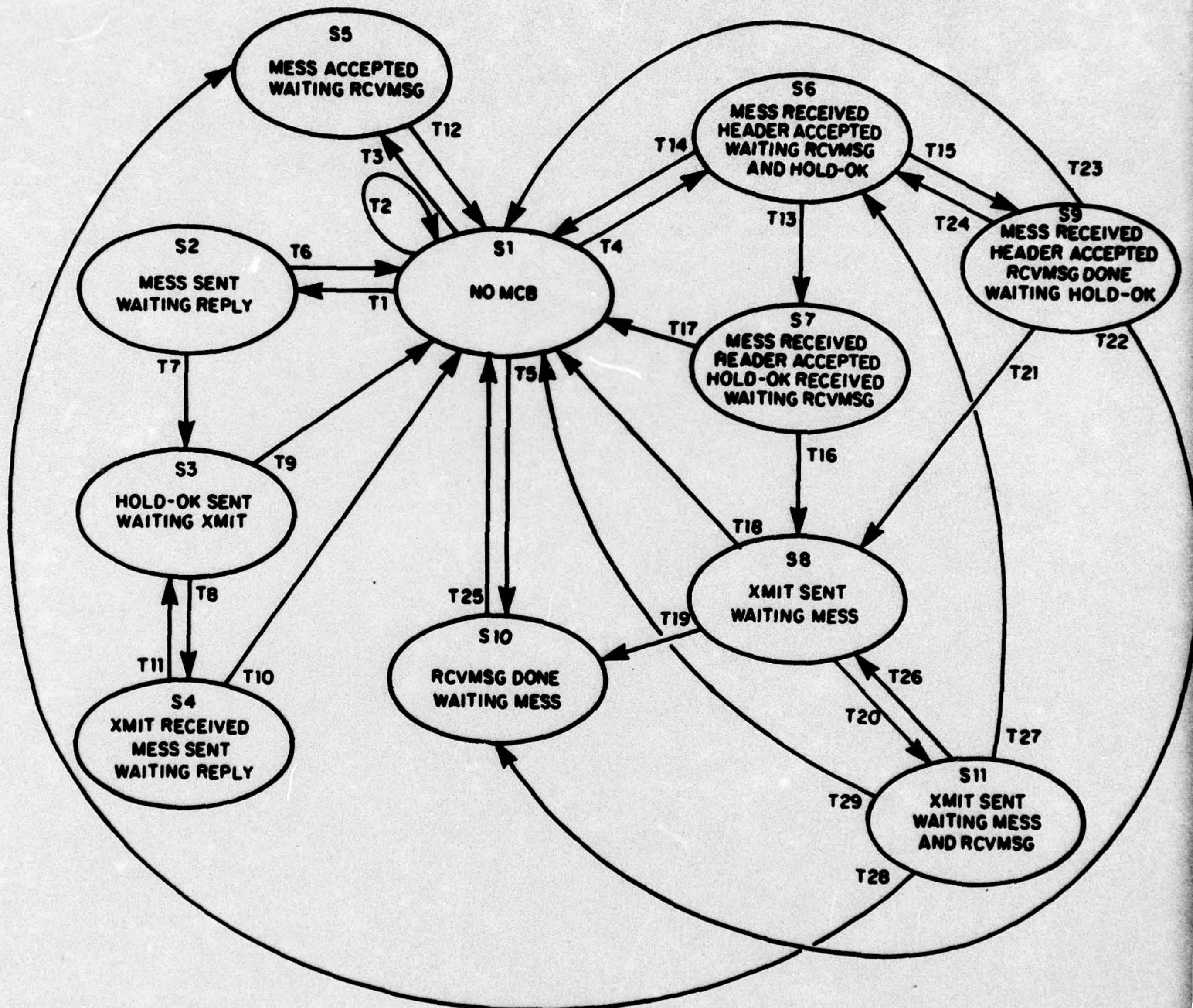


FIGURE C1

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (14) BBN [illegible]-3483-Rev-1
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): (6) MSG: The Interprocess Communication Facility for the National Software Works (Revision 1)
5. TYPE OF REPORT & PERIOD COVERED: Scientific
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): R. Thomas, S. Schaffner (12) 98p.
8. CONTRACT OR GRANT NUMBER(s): (15) N00014-75-C-0773
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman Inc., 50 Moulton Street, Cambridge, Massachusetts 02138
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (11) 23 Dec 76
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency Information Processing Techniques, 1400 Wilson Blvd., Arlington, VA
12. REPORT DATE: December 23, 1976
13. NUMBER OF PAGES: 96
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Code ONR-430D, 800 N. Quincy Street, Arlington, Virginia 22217
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report): Distribution of this document is unlimited. It may be released to the Clearinghouse, Depeartment of Commerce for sale to the general public.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): (10) Robert H. Thomas, Stuart C. Schaffner
18. SUPPLEMENTARY NOTES:
(a) This research supported by DARPA under ARPA Order No. 2901.
(b) This report also published by Massachusetts Computer Associates, Document No. CADD-7612-2411.
19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
National Software Works; Computer Networks; Interprocess Communication; Distributed Systems; Network Protocols; Resource Sharing
20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

This report describes the communication facility, named MSG, which was developed to support interprocess communication for the implementation of the National Software Works (NSW). The more important of the processes which comprise NSW and the pattern of communication which those processes require are described. Next from those patterns a model of interprocess communication which is sufficient for NSW is abstracted. Finally, the details of the MSG facility itself are developed.

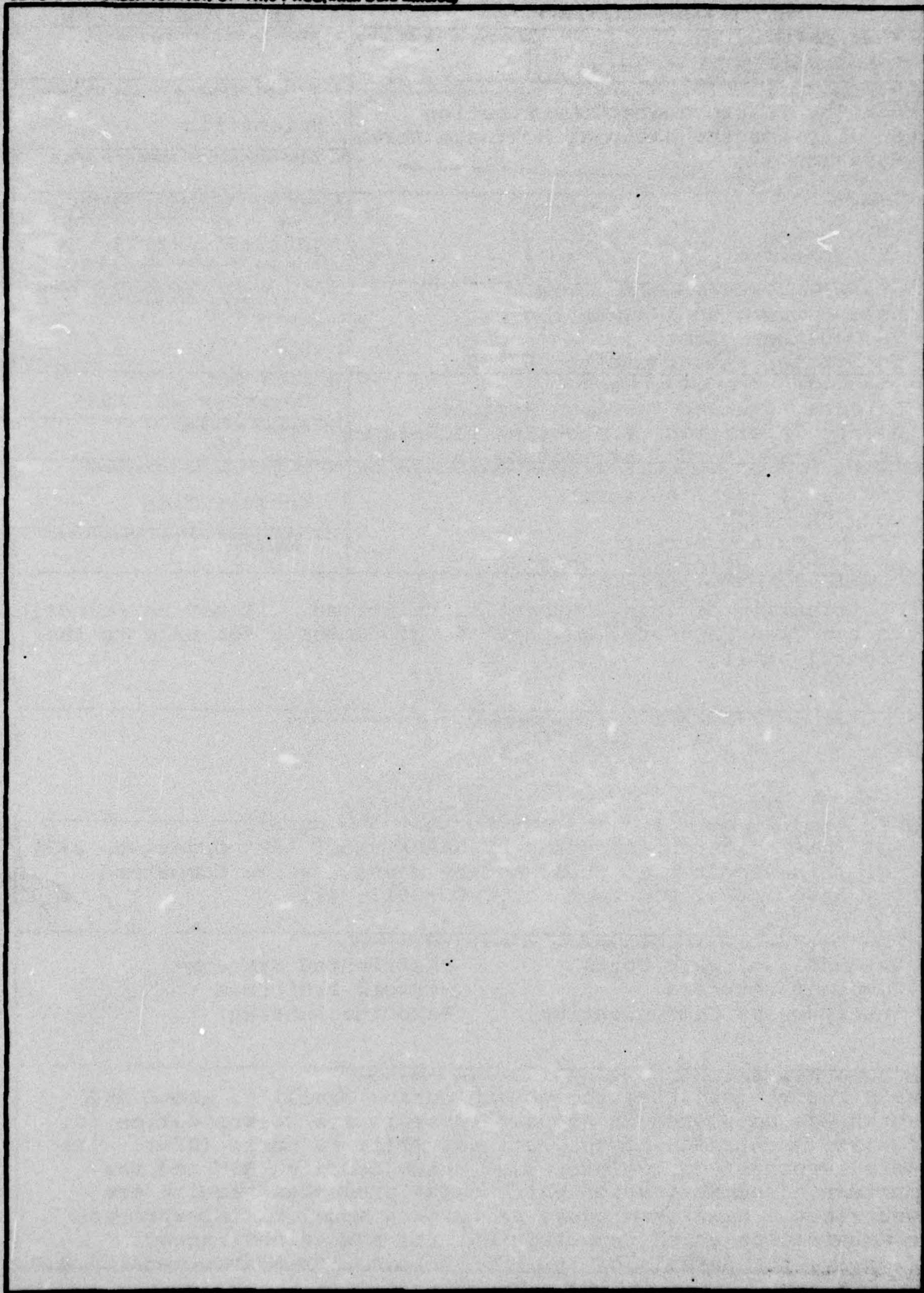
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

060100

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)



SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)