AD A062124

DDC FILE COPY

Technical Report

"Abstract Model of MSG:
Final Version"

N00039-78-G-0020

LEVEL II

12

6

Abstract Model of MSG

First Phase of an Experiment in Software Development

by

10

Glenn H./Holloway,
William R./Bush
George H./Mealy

9 Final ~~Version~~ rept.

11 Oct[redacted]78

12 184 p

Sponsored by

Defense Advanced Research Projects Agency (DoD)

ARPA Order No. 3079.3

Monitored by Naval Electronic Systems Command

Under Contract #N00039-78-G-0020

15 ARPA Order-3079

ACCESSION for

NTIS

DDC

Per ltr. on file

A

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Abstract Model of MSG: First Phase of an Experiment in Software Development -- Final Version

5. TYPE OF REPORT & PERIOD COVERED
technical report

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Glenn H. Holloway, William R. Bush, and George H. Mealy

8. CONTRACT OR GRANT NUMBER(s)
N00039-78-G-0020

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Department of the Navy
Naval Electronics Systems Command
Washington, D.C. 20360

12. REPORT DATE
October, 1978

13. NUMBER OF PAGES
175

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
same as above

15. SECURITY CLASS. (of this report)
unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

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

18. SUPPLEMENTARY NOTES

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

software development	software maintenance
program families	stepwise refinement
National Software Works	MSG

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

This report describes the first phase of an experiment designed to demonstrate techniques for software development and evolution. The experiment involves the production of a family of functionally similar systems on dissimilar host computers with markedly different operating systems. The basic technique being used is machine assisted stepwise refinement from an abstract model program that embodies the desired characteristics of the family members

cont next pag.

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Abstract Model of MSG: First Phase of an Experiment in Software Development -- Final Version	5. TYPE OF REPORT & PERIOD COVERED technical report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Glenn H. Holloway, William R. Bush, and George H. Mealy	8. CONTRACT OR GRANT NUMBER(s) N00039-78-G-0020
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy, Naval Electronics Systems Command, Washington, D.C. 20360	12. REPORT DATE October, 1978
	13. NUMBER OF PAGES 175
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) same as above	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

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

unlimited

18. SUPPLEMENTARY NOTES

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

software development
program families
National Software Works
software maintenance
stepwise refinement
MSG

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

This report describes the first phase of an experiment designed to demonstrate techniques for software development and evolution. The experiment involves the production of a family of functionally similar systems on dissimilar host computers with markedly different operating systems. The basic technique being used is machine assisted stepwise refinement from an abstract model program that embodies the desired characteristics of the family members

over
page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) without overconstraining the individual implementations.

The example system being developed in this experiment is MSG, the interprocess communication component of the National Software Works (NSW). MSG has already been specified and implemented by conventional means for several host computers. Our experiment consists of: production of an abstract model of the MSG family, realization of MSG on two actual hosts, to study the incremental costs of producing new instances by our techniques from a suitable model, and evolution of the MSG family in accord with actual changes to the MSG specification as they arise, to evaluate the efficiencies of maintaining families in unison.

Summary

This report describes the first phase of an experiment designed to demonstrate techniques for software development and evolution. The experiment involves the production of a family of functionally similar systems on dissimilar host computers with markedly different operating systems. The basic technique being used is machine assisted stepwise refinement from an abstract model program that embodies the desired characteristics of the family members without overconstraining the individual implementations.

The exemplary system being developed in this experiment is MSG, the interprocess communication component of the National Software Works (NSW). MSG has already been specified and implemented by conventional means for several host computers. Our experiment consists of production of an abstract model of the MSG family, realization of MSG on two actual hosts, to study the incremental costs of producing new instances by our techniques from a suitable model, and evolution of the MSG family in accord with actual changes to the MSG specification as they arise, to evaluate the efficiencies of maintaining families in unison.

The first phase of the project, creation of a preliminary MSG model, is complete. This report is a guide to readers of the model. Section 1 gives an introduction to the experiment and to MSG. Section 2 is an overview of the model, discussing its internal structure and external interfaces. Communication between MSG and the local processes it serves is described, as are the sequencing and buffering algorithms by which interprocess messages are delivered. The method used to encapsulate the detailed representations of network protocol items is presented, and the schemes for cleaning up MSG's data base when transactions time out or are rescinded are also discussed. Finally, section 3 of the report provides a guide to the language in which the model is written

The text of the model appears in the appendix.

1. Introduction

An experiment in program development and evolution is under way at Harvard's Center for Research in Computing Technology. Its purpose is to test techniques intended to reduce the cost and enhance the reliability of software maintenance.

The experiment involves the implementation of a system of moderate size and complexity that has been designed for the Defense Department and has been implemented for several host computers by conventional means. The exemplary system is thus actually a family of programs sharing a common specification but having a number of markedly different implementations.

Such families occur quite frequently. Operating systems that must provide similar environments on different physical configurations, communications processors that must observe a common protocol while running on a variety of host machines, and compilers of the same programming language for different target computers can all be regarded as comprising families.

Our experiment deals particularly with the creation and maintenance of such program families, although the techniques we employ are effective in developing single-instance systems as well. Our aim is to show that by concentrating from the outset on the development of a family of programs, designers can dramatically reduce the costs of producing a new member of the family and of maintaining its members in unison.

1.1 The Exemplary System: MSG

The specific program we are experimenting with, called MSG, is the interprocess communications handler for the National Software Works (NSW) [Spec]. The NSW is a distributed operating system now being developed for the ARPA network It provides users of the ARPANET with a uniform means of accessing the tools and data bases of the network without having to know where they reside or how to deal with the idiosyncracies of individual tool-bearing hosts.

An instance of MSG runs on each host participating in the NSW. In effect, it extends the local host operating system. It is responsible for starting and managing the processes which implement various NSW functions, such as the Works Manager, which assigns tasks to tool bearing hosts, the File Package which implements a distributed filing system, the Front End, which provides users with a uniform interface to the NSW, and the Foremen, which supervise the various tools.

MSG routes messages between NSW processes, and, when necessary, establishes direct network connections between them. Messages may be either _generically_ or _specifically_ addressed. A generically addressed message causes MSG to seek or create a process of a particular class (e.g., Works Manager). Specifically addressed messages are routed by MSG to a precisely designated destination process. They are normally used after communication has been established using a generically addressed message.

MSG provides facilities for ordering messages and synchronizing message streams with respect to unusual events or exceptions. Normally, MSG does not guarantee that messages will be delivered in the order submitted. The sending process, however, may specify that a subset of its messages to a given destination must be delivered in the sequence sent, and that such sequenced messages must not be sent if earlier members of the sequence were not deliverable for some reason.

To permit interprocess signalling of exceptional events, MSG provides for high priority transmission of short communications, called alarms. Alarms are handled independently of any message traffic between the sending and receiving processes. They can be used to facilitate resynchronization of a message interchange.

To aid synchronization of messages with alarms, MSG permits certain messages to be stream-marked. A stream-marked message will be delivered after any prior messages to the same destination and before all those sent subsequently.

Instances of MSG on different hosts communicate using a protocol that provides for the establishment and termination of MSG-to-MSG network connections and for all three types of communication supported by MSG: messages, alarms, and direct connections between processes. The message protocol includes items that facilitate flow control so that the burden of buffering messages can be distributed between sender and receiver when traffic is heavy.

Further details on the functions performed by MSG and on the MSG-to-MSG protocol can be found in the MSG Design Specification [Spec]. A familiarity with the specification will be assumed in section 2 of this report.

1.2 Method of Development and Evolution

Our hypothesis is that program maintenance can be made less costly and more dependable if system designers view their task as the design of a program family whose members are derived from a common ancestor, or root, which we call the abstract model for the family. Use of the term "abstract" is not intended to imply that the model is a formal mathematical object. Rather it is a program written with sufficient generality that members of the family can be obtained by specialization of it.

Concrete program instances are derived by a sequence of refinements from the abstract model. Each refinement encapsulates related design decisions that distinguish a class of concrete instances. A refinement may give definition to a procedure, a data structure, or a control pattern left unbound at the abstract level, or it may modify or augment such constructs. The program family thus forms a tree, with the abstract model at the root, the concrete instances at the leaves, particular refinements labeling branches, and interior nodes that represent classes of instances sharing a common set of refinement choices.

We have developed, and are constantly improving, tools that aid developers by maintaining the family tree in a data base and by mechanizing the task of applying refinement sequences. Using these tools, a new member of an existing family is less expensive to produce than one developed separately because the basic model and most of the refinements that yield the new version will be shared with existing instances. Moreover, modifications to broad subfamilies can be effected in unison, by applying altered refinements that reflect revised design decisions and simply reapplying those not affected.

Our techniques are not restricted to programs like MSG that must exist in many functionally similar but internally disparate versions at the same time. The history of every evolving program is a series of closely related versions, which themselves comprise a family. The clear isolation of the individual design decisions leading to a particular instance presents the maintainer with a picture of its structure that enables him to understand the ramifications of a particular modification better than can a single-level representation of a system.

This approach is particularly beneficial if the software engineer charged with modifying or expanding a system has not been involved in its prior development. Instead of confronting a finished product together with some documentation about what it does, he has access to information about how it was constructed, why particular decisions were taken, and what the effects of changing them would be. He has tools that permit him to take the program apart and reconstruct it easily, to generate a new version or to revert quickly to an old one.

In short, with these techniques, the roles of initial developer and maintainer tend to merge. The maintainer can afford to leave the system in as clean and well-structured a state as its originators, even after several stages of evolution. By contrast, systems developed by conventional methods often reach an overmaintained state after a few successive versions have been produced. After this point, the quality of each version may well deteriorate rather than improve [Belady].

Another aspect of software construction that fits naturally into the family maintenance framework is the use of library procedures and data abstractions. An implementer often needs to take a general algorithm or data definition, with known properties of correctness and performance, and specialize it to a particular task, with particular physical data representations. Sometimes many distinct realizations of the same abstract notion will be used in a single system. The specialized versions of such library modules also constitute a family whose members can be produced and maintained using the tools we are developing.

1.3 Reasons for Choosing MSG

The MSG paradigm is a good basis for our experiment because it is a family of communicating programs and because teresting maintenance requirements, drawn from experience with the NSW can be anticipated in the near future.

Instances of MSG should appear to function identically in spite of highly disparate host systems. Since they must communicate effectively, the behavior of each must be closely coupled to that of the others. When one MSG instance is altered in an externally observable way, all should be altered in unison.

The MSG-to-MSG protocol is relatively simple, but the operating environment is imperfect. Hosts may die, or pause indefinitely; processes may also die, or they may change their minds; interhost connections may drop. These possibilities introduce complexities in interhost MSG communication which are virtually impossible to capture in a nonprocedural English, item-by-item protocol specification.

Such a specification, indeed, is aimed at minimizing the description of external behavior in order to give maximum flexibility to implementers and maintainers of individual instances. Unfortunately, the use of imprecise interface specifications when interfaces are potentially complex can lead to inefficient, defensive strategies, or much worse, to inconsistent interpretations by different implementers.

Our view is that each instance of MSG should be constructed as if it were to communicate with an exact replica of itself. The abstract model of MSG thus provides a precise description of the behavior of any instance in any circumstance, and this knowledge can be used in realizing any concrete version. There is no harm in this mutual knowledge because the family members are maintained in unison. Useful flexibility, that needed to accomodate differences in host facilities and resources, results from the abstractness of the model.

1.4 Phase One: The Abstract Model

During the first phase of the project, we have constructed an abstract model of MSG. In later phases, we plan to refine the model to two concrete instances on dissimilar target machines, the DEC PDP-10 and PDP-11, running the TENEX and UNIX operating systems, respectively.

The remaining sections of this report are intended as a guide to readers of the model, which appears in the appendix. Section 2 provides an overview of the model itself. It assumes some familiarity with the MSG Design Specification. Section 3 describes the modelling language and highlights some of the features we use to aid both the readability and refinability of abstract algorithms.

2. Guide to the MSG Model

2.1 An Overview

MSG instances on each of the participating hosts link the NSW together by carrying out communication between the processes that comprise it, as depicted in figure 1. The double shafted arrows in this picture represent channels for interprocess communication. Those linking processes on different hosts will be implemented using network connections. Those between processes on the same host will be realized using whatever interprocess communication (IPC) facility is most suitable on that host. Some hosts, such as CCN (360/91), offer an IPC facility that closely resembles the channel abstraction we have used in modelling MSG. On others, such as TENEX, direct communication via shared memory may be best. We neither assume nor prohibit hierarchical relationships between MSG and the processes it serves. Depending on the host operating system (if any), it may be useful to organize processes hierarchically in order to achieve the most effective IPC.

In our model, each MSG instance is structured as a collection of _paths_ serving specialized functions _and sharing a common data base_. Paths, like processes, are potentially concurrently executable control units. We distinguish "path" from "process" to emphasize that paths internal to MSG may or may not be implemented using the process management facilities of the host operating system.

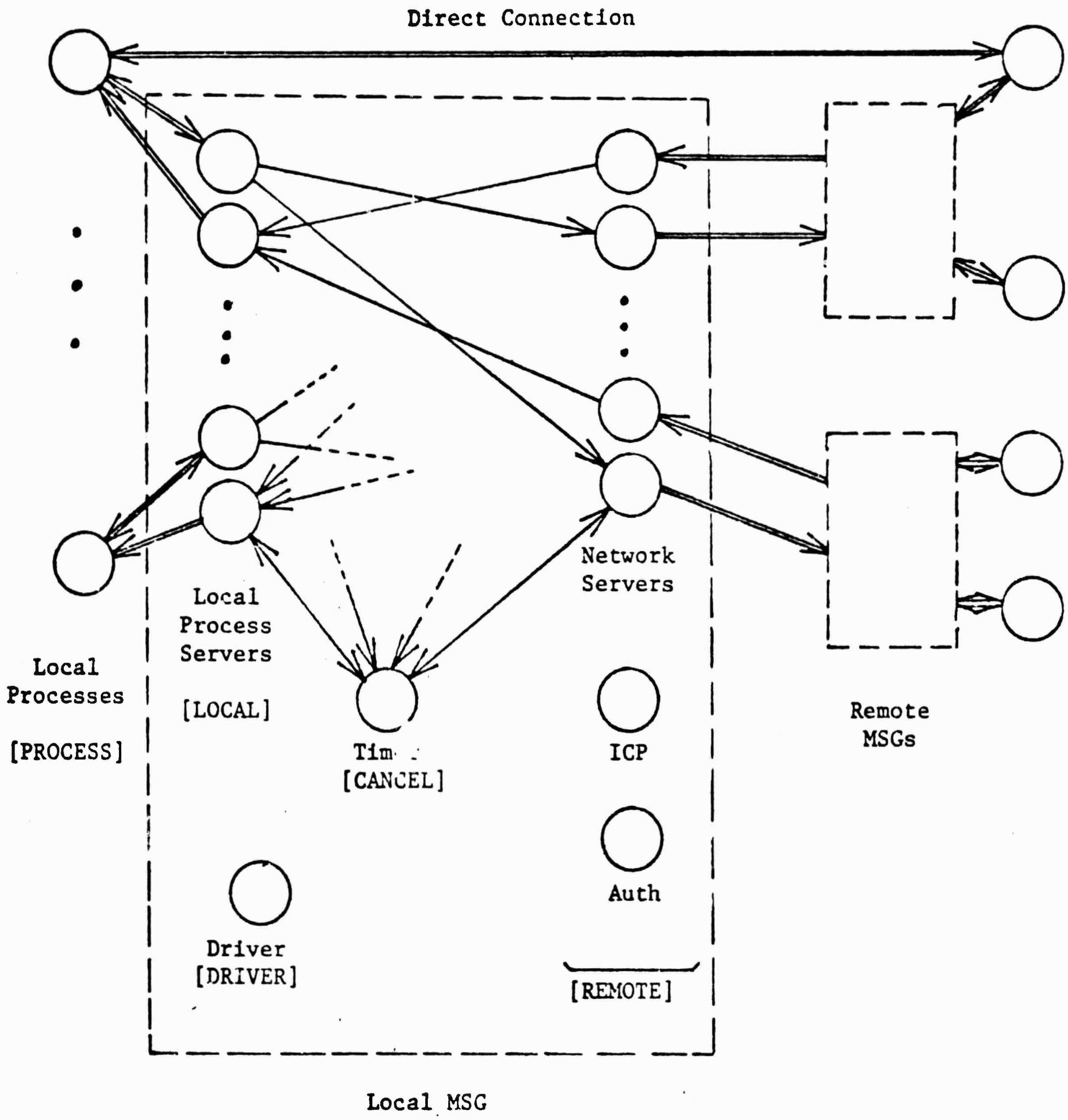


Figure 1. MSG Internal Structure and External Interfaces

For each transaction that MSG processes, that is, for each message and alarm sent or received and for each direct connection established, the data base holds a record describing the state of the transaction.

The modules that comprise the MSG model are: LOCAL, which defines the interface between MSG and local processes, REMOTE, which contains the network interface, QUEUE, which includes routing algorithms and data base management routines, CANCEL, which expunges transactions that, for one reason or another, cannot be completed, and DRIVER, which is responsible for initialization of MSG and of local processes. Another module, GLOBAL, collects the common (among modules) data definitions for the model. Finally, PROCESS is a small module that contains the routines that must be embedded in processes for use in initiating MSG primitive operations.

Within MSG there is a <u>server pair</u> for each user process. A server pair is a pair of paths whose separate functions correspond roughly, though not exactly, to input and output. In the case of a process server, the two halves correspond to the two phases of many MSG primitive operations. The first is the <u>call</u> phase, in which MSG accepts and validates arguments. If the call is acceptable, the process is allowed to continue execution while MSG proceeds with its request. When the transaction is complete, the second, or <u>delivery</u> phase of the primitive operation occurs, in which the disposition of the transaction is transmitted to the process, along with any incoming information, such as message text, an alarm code, or a network connection designator. Primitive operations that occur in two stages this way are said to create pending events for later completion.

The model does not fix the number of user processes assigned to a given server pair. There may be one pair per process, one per generic class, one for all processes, or some other configuration if convenient. If there are multiple server pairs, each is identical in operation to the next. They share the procedures given in module LOCAL.

Symmetrical with the set of user process servers is another set of server pairs called network servers. A network server implements the inter-MSG protocol by formatting and transmitting protocol items from the local MSG to instances of MSG on other hosts, and receiving items from other hosts and translating them into internal MSG records. In a given realization of MSG, each network pair may handle a single remote host or several of them. Like the local process servers, each network server is identical with the next. All share the algorithms given in module REMOTE of the model.

The paths labelled ICP and Auth in figure 1 are also defined in the module REMOTE. They take part in the establishment of connections with remote hosts. ICP (which stands for Initial Connection Protocol) is activated when another host initiates a connection with the local MSG. Auth is used in authenticating the identity of the local MSG when it has initiated a connection with an MSG instance on another host.

Timer is MSG's clock-watching path. For most of the states in which a transaction can be, it is desirable that a limit be enforced on the amount of time allowed to elapse without some progress being made. Time limits are specified when a user process issues a primitive call, and they are also set in certain states by MSG in order to implement its flow control policies. The Timer path, defined in the module CANCEL, manages a queue of transactions sorted in order of their expiration times. It uses a real time clock interrupt to generate timeout signals, and it responds to them by aborting expired transactions and expunging their records from the system. The intent is that, given a long enough quiescent period with no incoming or outgoing transmissions, MSG will tend to return to its initial state. (There are exceptions to this rule, however. Direct connections to other hosts, for instance, do not expire whether or not they are used.)

The remaining internal path shown in figure 1, called Driver, is the first to be executed when an MSG instance is executed. It performs initialization tasks, including the creation of local processes and server paths. In some realizations of MSG, the Driver may be a superior of the paths comprising MSG, and indeed may act as a scheduler for them. These details are omitted from the abstract DRIVER module, however, since they depend critically on the nature of the underlying host's facilities.

The single-shafted arrows in figure 1 denote direct communication among internal MSG processes via a shared data base containing records for each transaction known to the system at any moment. The routines that manage this data base are contained in the MSG modules called QUEUE and CANCEL. The data base manager is not a separate path. Rather, the routines in QUEUE and CANCEL are called by the internal paths of MSG to access and alter the data base in response to particular stimuli such as the arrival of a protocol item on the network, the execution of a user primitive, or the expiration of a time limit for some transaction. QUEUE is so named because its primary task is to enqueue items for output over the network or for delivery to local processes. CANCEL contains routines to discard incomplete transactions when they must be aborted.

The data base is composed mainly of transaction records defined by the data types MessBlock, AlarmBlock, and ConnBlock, one for each of the three modes of communication supported by MSG. These types are defined in the module GLOBAL. To emphasize the fact that these records are shared among multiple paths and are accessible via several routes at once in the data base, the model usually manipulates "handles" on records. The data type MessHandle, for instance, has the abstract behavior of a pointer to a MessBlock (though it need not be realized as a pointer in concrete versions of MSG). Collections of message records are expressed as sets or queues of MessHandles.

The anchor points from which most records in the data base are accessed are the data structures ProcessTable, ServerTable, HostS, TransactionTable, GenericTable, and TimerQ. ProcessTable maps process names (and, in particular, the process instance identifiers within process names) to ProcessHandles. Every local process managed by MSG has a ProcessHandle in ProcessTable. The data accessible through a ProcessHandle (see module GLOBAL for all global data types) give the state of MSG's interaction with the process, including the pending primitives waiting to be completed, incoming messages and alarms waiting for delivery, and direct network connections to other processes.

ServerTable contains a record for each of the process server and network server pairs internal to MSG. This is, in general, a dynamically varying collection, since there might be a server pair for each active user process and remote host connection. An important component of the record for each server is its delivery queue, which is filled with transaction records produced by other paths and emptied by the server as the items are attended to. The delivery queue of a local process server holds the pending events ready to be completed. That of a network server contains protocol items awaiting transmission.

HostS includes an entry for each remote host with which MSG has established contact. Among other fields, the remote host entry holds the incarnation number of the MSG instance running on the remote host, and a set of ChannelHandles that describe the connections currently open to that host for use by MSG.

Since some transactions handled by MSG involve the exchange of several protocol items, it is convenient to have each MSG instance assign a unique identifier to the transactions in which it is involved. TransactionTable is used in the assignment of these identifiers, which are designed to repeat very infrequently. It also gives quick access to the data base record for any transaction, given its transaction identifier as a key.

GenericTable has an entry for each generic process class, such as Works Manager or File Package. It contains records needed to start a new process of a given class, and it indicates which NSW hosts support each generic class, so that an outgoing generically addressed message can be routed. GenericTable also contains holding places for incoming generic messages when it is not possible to assign them immediately to specific processes of the proper class.

TimerQ is the queue of transactions being timed by Timer, sorted in order of expiration time.

Of course, with multiple parallel paths accessing and altering shared records, it is necessary to employ a record locking mechanism to guarantee orderly sequential access. This mechanism is invoked in the model through the primitives Seize, TestSeize, and Release. Seize blocks the calling process until exclusive access can be granted to the caller. Release unlocks a previously seized resource. TestSeize attempts to seize a resource, but does not block if it is unsuccessful. Instead, it returns a Boolean value indicating success or failure.

Although these primitives are used in the MSG model, so that it can be checked for errors such as deadlock, the implementations of Seize, TestSeize, and Release are deliberately left unspecified. In some purely sequential realizations, there will be no need for locks and these primitives can be refined away.

Storage management of records presents a similar situation. The routine Allocate is used to enter new records in the data base, and Free is used to indicate that one reference to a record has been deleted. However, the record may only be reclaimed when all such references have been eliminated. If Free cannot safely reclaim storage, it simply performs a Release. The detailed mechanisms for ensuring that storage can be reclaimed safely are deferred to a lower level of refinement.

2.2 The Interface Between MSG and Local Processes

The modules PROCESS and LOCAL model the interface between MSG and the local processes it serves. PROCESS contains routines to be loaded with user processes and called by them to initiate MSG primitives. These routines are quite simple; they merely transmit arguments to MSG and guide results into place. LOCAL contains the procedures and tables used by each pair of server paths which implement the primitive operations.

2.2.1 Communication Between MSG and Local Processes

In the abstract MSG model, processes communicate with MSG by way of signals and channels. The particular mechanisms used were chosen to make the model as simple as possible within the constraint that they must be realizable in a variety of host systems.

A signal is a dataless communication between two processes used to herald events and achieve synchronization. In MSG, signal identifiers of type SignalType are used to distinguish among the several signals with different meanings that may be in use between two processes. The expression SIGNAL(<signal identifier>) sends the designated signal. WAIT(<signal set>) blocks the calling process until a member of the designated set of signals is received. It returns the identifier of that signal as its result.

A *channel* is opened between two processes when each calls CHOPEN with the same channel identifier and compatible channel descriptors. SEND(<data list>, <channel>) is used to send a group of data over a channel. SEND always blocks the calling process until the data have been received. RECEIVE(<variable list>, <channel>, <block flag>) fills the elements of a list of variables by reading from a channel. RECEIVE's third argument is a Boolean which, when TRUE, causes the calling process to block until the whole variable list has been read. If this argument is FALSE, the caller may proceed while the data are being read. Presumably, in the latter case, the caller has another way of knowing when the transmission is complete. For example, the sender may use a signal to alert the receiver.

A channel may be closed from either side using the CHCLOSE operator, but it will be closed only after data in the channel have been received.

CHOPEN, CHCLOSE, SEND, and RECEIVE are specified in module GLOBAL.

PROCESS and LOCAL use a few notational extensions for dealing with channels. The expression <X1, ..., Xk> =¦> CH is equivalent to SEND(<X1, ..., Xk>, CH), which sends the successive values X1 though Xk over channel CH. Similarly, <V1, ..., Vk> <¦= CH stands for a non-blocking RECEIVE call:

RECEIVE(<V1, ..., Vk>, CH, FALSE)

while < , ..., Vk> <||= CH stands for a blocking RECEIVE:

RECEIVE(<V1, ..., Vk>, CH, TRUE).

Also, <F1, ..., Fn>(M) is equivalent to <M.F1, ..., M.Fn>.

2.2.2 Primitive Routines Within User Processes

Module PROCESS contains a routine definition for each primitive defined in the MSG specification. Each consists of a call on one general routine, PCall. PCall has four arguments: Op identifies the particular primitive being invoked; SendList is the list of data that must be sent to MSG to allow it to execute the operation; ReceiveList is a list of variables to be filled by the time the primitive completes; and CreatesPendingEvent is bound to TRUE for those primitives that occur in two stages, i.e., that create pending events. In addition, PCall sets the variable PendingEventID, which is an output parameter of every pending event creating primitive routine.

As an example, the body of the ReceiveSpecificMessage primitive is:

```
PCall("ReceiveSpecificMessage",
      <Signal, Timer>,
      <Text, SourceProcess, Handling, Disposition>,
      TRUE)
```

Here Signal represents the signal identifier chosen by the calling process for MSG's use in signalling completion of the Receive. By convention, a special value of SignalType can be used if the caller wishes the primitive to block until completion instead of returning while the transaction proceeds. Timer is the maximum delay the calling process will allow before satisfaction of the Receive request. Text, SourceProcess, Handling, and Disposition are the output variables into which the results of the Receive are to be placed. Text will hold the actual message; SourceProcess will be the name of the sending process, Handling will indicate what special handling the message may have, and Disposition is a code that indicates whether the operation succeeded or failed, and if it failed, then why.

Every process has a channel, called MSGChannel, permanently open to MSG. It is used for those primitives that do not create pending events and for the call phase of those that do. PCall uses MSGChannel to send the identifier Op and the SendList parameters to MSG, and then waits for its reply on the same channel. MSG answers with a disposition code. If this code indicates some error, PCall terminates the primitive by returning the error code. If MSG's reply is normal, and the primitive does not create a pending event, then the rest of the data for the ReceiveList come over MSGChannel, and the primitive returns an indication of normal completion.

For pending event primitives that get past initial validation, MSG sends back an identifier which will be uniquely associated with the current transaction. PCall assigns this identifier to PendingEventID. It can be used by the calling process to rescind a pending event before it has completed.

PendingEventID is also used to open a new channel to be used exclusively for the completion of the current primitive. Then PCall issues a RECEIVE for the ReceiveList. Depending on whether the caller has asked to block until completion or to be allowed to run, PCall will block or return immediately. In the latter case, MSG uses the given signal to alert the user process that the primitive has completed.

2.2.3 The Local Process Servers

The other side of the PROCESS-LOCAL interface is encoded in the server pair assigned to the user process. The main procedures for the pair of paths comprising the server are UserCallServer and UserDeliveryServer. These in turn select procedures for handling the individual MSG primitives. All of these procedures are contained in module LOCAL.

The handling of a SendSpecificMessage is typical of how the server pair deals with pending event primitives. UserCallServer monitors the permanent channels to the processes it is responsible for. When it receives an operation code over one of these channels, it dispatches to the appropriate handler, as given by OpTable. The handler is given the OpCode (since the same handler may implement more than one primitive), the channel to the local process, and a ProcessHandle for the process. In the case of a SendSpecificMessage, UserCallServer calls StartSendMessage.

StartSendMessage receives the parameters sent by the user process and validates them. If it finds an error it aborts the primitive right away, sending an error code to the process. Otherwise, it indicates that the call phase will complete normally by sending the code "Incomplete". It assigns a transaction identifier and sends it to the user process, and it uses this identifier to open a new channel for completion of the event. Finally, a transaction record is created for the message and is passed to EnQOutputMess, one of the routines exported by QUEUE.

Eventually, the transaction will be placed in the delivery queue of the server and will be found there by the UserDeliveryServer. This path uses the original operation code to dispatch to a handler for the delivery phase. In the case of SendSpecificMessage, the delivery handler is EndSendMessage. EndSendMessage finds the channel opened for this transaction and sends the disposition code (the only output value in this case) to the process. It then uses the signal supplied by the process to signal completion of the event, and it frees the transaction record for the message.

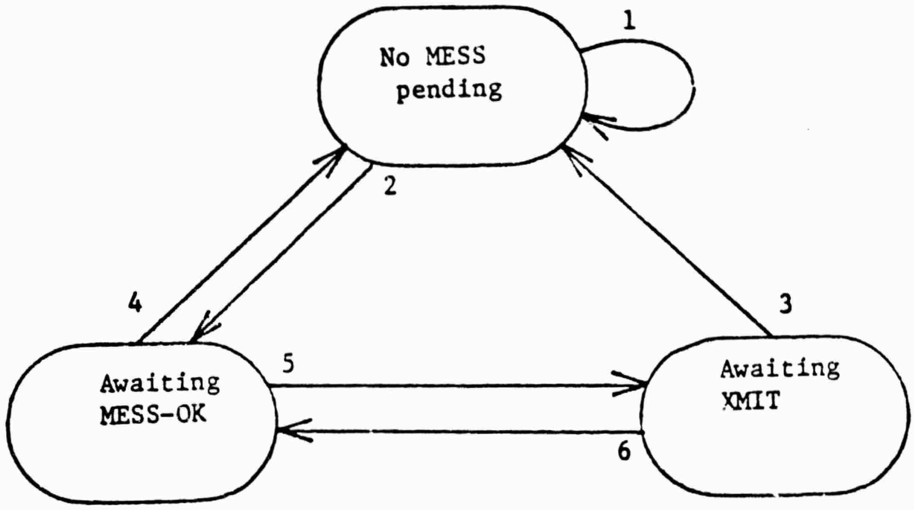
2.3 Message Routing and Data Base Management

As mentioned in the overview, the modules QUEUE and CANCEL contain routines that respond to events which drive MSG: execution of primitives, receipt of protocol items, timeout of events, and so on. In particular, QUEUE embodies most of MSG's routing and buffering algorithms: finding a specific destination for generically addressed messages, deciding whether to buffer or reject messages that cannot be delivered immediately, sequencing the transmission of "special handling" messages, and the like. This section summarizes QUEUE's algorithms for each class of communication supported by MSG. The next section discusses the CANCEL module.

2.3.1 Message Handling

2.3.1.1 Outgoing Messages

Figure 2 is a state transition diagram for an outgoing message. The initial state is labelled "No MESS pending". Arcs representing transitions have labels of the form Stimulus/Response. Here "stimulus" describes the event giving rise to the transition, and "response" describes MSG's action (if any) upon entering the new state. "SendMess/send MESS" (transition 2), for example, denotes the execution of a SendGenericMessage or SendSpecificMessage primitive by a local process. MSG's response is to transmit a MESS protocol item to the destination and place the



1: MESS-OK/ --
   MESS-REJ/ --
   MESS-HOLD/send MESS-CANCEL
2: SendMess/send MESS
3: MESS-REJ/abort SendMess
   Buffer needed or Timeout/send MESS-CANCEL,
     abort SendMess
4: MESS-OK/complete SendMess
   MESS-REJ/abort SendMess
   MESS-HOLD/send MESS-CANCEL,
     abort SendMess
   Timeout/abort SendMess
5: MESS-HOLD/send HOLD-OK
6: XMIT/retransmit MESS

Figure 2. State Transitions for Cutgoing Messages

transaction in the state "Awaiting MESS-OK". At the same time, a MessHandle representing the transaction is placed in the OutputMessQ associated with the the sending process. The response to a MESS item can be acceptance (MESS-OK), outright rejection (MESS-REJ), or a request to hold the message until the receiving MSG can ask for retransmission (MESS-HOLD). If the source MSG decides to accept the request to hold, it signifies by sending HOLD-OK and places the transaction in the state "Awaiting XMIT" (transition 5). Receipt of an XMIT protocol item will stimulate retransmission of the MESS (transition 6), or the transaction may die through receipt of a rejection or through old age (transition 3).

The self-loop from the "No MESS pending" state (transition 1), which responds to a MESS-HOLD for a non-existent transaction by sending MESS-CANCEL, is the result of the model's handling of timeouts. If the transactions's time limit expires while it is in the "Awaiting MESS-OK" state (transition 4), no MESS-CANCEL is sent to indicate that the message has been aborted. Thus, if the remote MSG later shows an interest in continuing the transaction, it is necessary to cancel it explicitly.

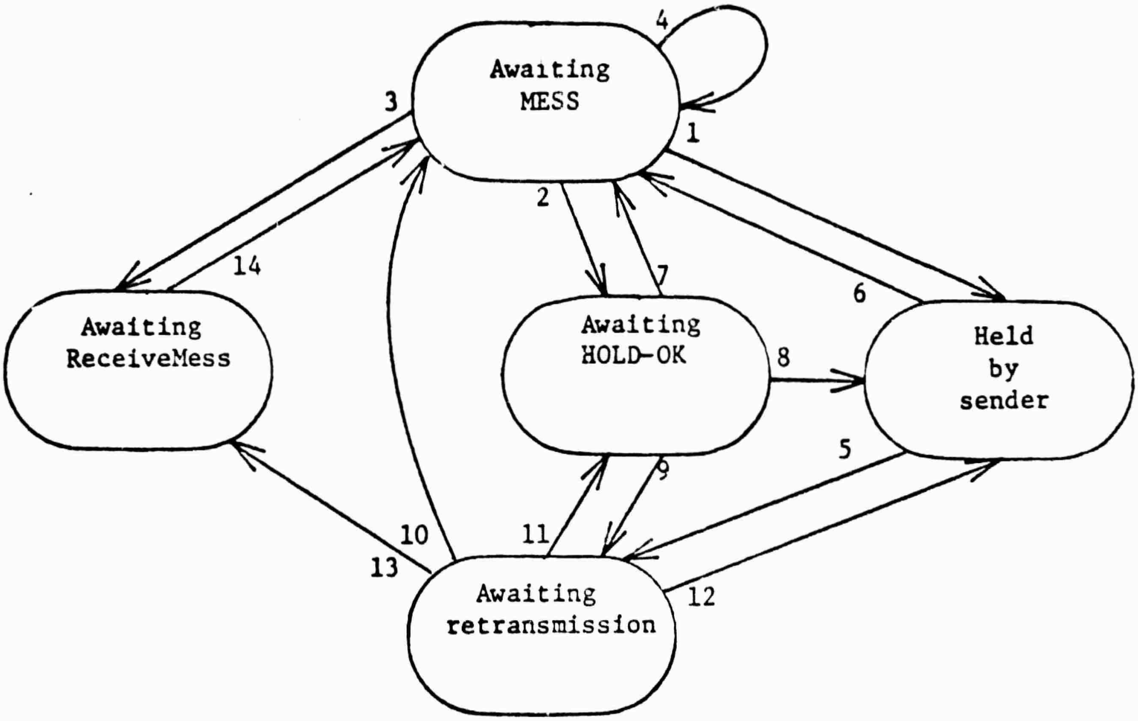
The routines in QUEUE that implement transitions shown in figure 2 are EnQOutputMess, RecordMESS\OK, RecordMESS\REJ, RecordMESS\HOLD, RecordXMIT.

2.3.1.2 Incoming Messages

Figure 3 shows what happens at the receiving end of a similar transaction. Here the initial state is labelled "Awaiting MESS". In this state, either there is no transaction pending at all, or a ReceiveMess primitive has preceded the arrival of a matching MESS item. In the latter case, a MessHandle for the transaction is queued in the ReceiveMessQ of the receiving process.

When a MESS item arrives, it may be accepted outright (transitions 3 and 4), or rejected outright (transition 4), or the receiving MSG may ask the sender to buffer the message (transitions 1 or 2). Transition 3 is taken when no matching ReceiveGenericMessage or ReceiveSpecificMessage primitive has been issued by a local process. Transition 1 occurs when the sending MSG has offered in advance to buffer the message, if necessary, and the receiving MSG has accepted the invitation.

When there is some delay in delivering the message, its MessHandle waits in the InputMessQ of either the appropriate generic class (if the message is generic) or the destination process (if it is specific) until it can be disposed of. In the "Awaiting ReceiveMess" state, either a Receive primitive is executed and the message is delivered, or the time limit placed on the transaction expires and the MessHandle is deleted (transition 14).



1: MESS(HoldOk)/send MESS-HOLD
2: MESS/send MESS-HOLD
3: MESS/send MESS-OK
4: MESS/send MESS-OK,
    complete ReceiveMess
   MESS/send MESS-REJ
   HOLD-OK/send XMIT
   MESS-CANCEL/ --
   ReceiveMess/ --
5: Buffer free/send XMIT
6: MESS-CANCEL/ --
   Timeout/send MESS-REJ
7: MESS-CANCEL/ --
   Timeout/ --
8: HOLD-OK/ --
9: HOLD-OK/send XMIT
10: MESS/send MESS-OK,
    complete ReceiveMess
   MESS/send MESS-REJ
   MESS-CANCEL/ --
   Timeout/ --
11: MESS/send MESS-HOLD
12: MESS(HoldOk)/send MESS-HOLD
13: MESS/send MESS-OK
14: ReceiveMess/complete ReceiveMess
   Timeout/ --

Figure 3. State Transitions for Incoming Messages

In the "Awaiting HOLD-OK" state, a proper stimulus is either MESS-CANCEL or HOLD-OK. MESS-CANCEL terminates the transaction (transition 7) and causes its record to be deleted. HOLD-OK may cause MSG either to send an XMIT (transition 9), if buffer space for the message has become available, or simply to put the transaction in the "Held by sender" state (transition 8).

Transactions get out of the "Held by sender" state either because the sender can no longer afford to hold a message and sends MESS-CANCEL (transition 6), or because sufficient buffer space becomes available to enable MSG to request retransmission of the message by sending XMIT (transition 5). The state "Awaiting retransmission" is almost equivalent to the initial state "Awaiting MESS" since the same responses are possible for a retransmitted message as for an initial message (though a request to re-hold is unlikely). One point of difference is that the initial state has transitions for HOLD-OK and MESS-CANCEL (transition 4). Like the initial state self-loop in figure 2, these result from the decision not to flatly terminate a transaction which has timed out. In this case, if a timeout occurs in the state "Awaiting HOLD-OK" (transition 7), MSG deletes the transaction without sending MESS-REJ. If, later, a HOLD-OK arrives for that transaction, MSG responds by inviting retransmission (XMIT). If a MESS-CANCEL comes in for a discarded transaction, it is just ignored.

The routines that implement the transitions of figure 3 are EnQInputMess, EnQReceiveMess, RecordHOLD\OK, and RecordMESS\CANCEL.

2.3.1.3 Generically Addressed Messages

An aspect of MSG's message handling procedures not displayed in figures 2 and 3 is its treatment of generic messages. On output, a generic message with a specific destination host is treated just like a specifically addressed message. If no host is specified, however, MSG must choose one from a list of hosts supporting the chosen generic category, e.g., Works Manager. If the message is not accepted by the first host, MSG tries the others in the list until either the message is accepted or the list is exhausted. To implement this search, MSG (in EnQOutputMess) marks such a message as having been hostless, and then assigns it the first host suitable for its generic class. The handler of message rejections (RejectOutputMess, a subroutine of RecordMESS\REJ) recognizes a transaction that has been marked as hostless, and chooses the next host in the list. If the list is exhausted, the pending SendGenericMessage is aborted.

When a generic message is received, MSG first looks for a destination process of the right category that has issued a ReceiveGenericMessage. If none is available, it will try to start a new process of the generic category and deliver the message to the new process. If the quota for such processes has been reached, and the option of holding the message has not been ruled out by the sender, MSG will queue the message in the InputMessQ for the generic category until a process issues a matching Receive or until a new one can be created.

2.3.1.4 Message Flow Control

To help determine when to request that a message be held by its sender and when to prompt its retransmission, MSG maintains two counts for each potential destination process and for each generic category. The first, FreeBufferSize, gives the amount of text buffer space alloted to the process that is not already in use by messages to or from that process. The second, CommittedBufferSpace, is the total text length of the messages currently being retransmitted by their senders. The value of FreeBufferSize less CommittedBufferSpace is called VirtualFreeSpace. VirtualFreeSpace is used to decide when to stimulate retransmission by sending XMIT.* When its value rises above a threshold, either through the freeing of some buffer space or the timeout of an expected retransmission, MSG looks in the appropriate InputMessQ for a held message to call in. Of course, the commitment of buffer space is not absolute. Unanticipated messages may arrive and be accepted before the one retransmitted, forcing its further delay. It seems likely however, that the simple algorithm chosen will perform well in most cases. If not, the model leaves ample latitude for changing it.

2.3.1.5 Sequencing of Messages Marked For Special Handling

As mentioned in the introduction, the MSG specification defines two classes of messages, called sequenced and stream-marked messages, for which special rules govern delivery order. All sequenced messages from a given source to a given destination must be delivered in the order submitted, and failure of one such message inhibits any further sequenced message flow between the processes until the source process executes a Resynch primitive. Stream-marked messages must be delivered after completion of any prior output messages to the same destination and before delivery of any subsequent messages. Failure of a stream-marked message inhibits all further message traffic from the source to the destination until a Resynch is issued by the sender.

Message sequencing is effected entirely by the sending MSG (see EnQHostSpecificMess), which checks before transmitting any message whether there is an existing incomplete message to the same destination that blocks transmission of the current message under the rules. (This is the reason OutputMessQ is a queue and not a set.) The transaction is placed in the special state "Awaiting prior MESS completion".

Later, when a prior message transaction completes successfully, the subroutine AcceptOutputMess scans the output queue for later messages that are unblocked by the completion, and it sends these messages. When a prior message is aborted, a similar scan occurs (in subroutine RejectOutputMess); however, in this case it may be necessary to abort some of the blocked message transactions because a sequenced or stream marked message has failed. If so, a set of "inhibited destinations" (InhibitedDestS) is updated for the source process, to ensure that further messages (or possibly just sequenced messages) to the same destination are inhibited pending a Resynch.

2.3.1.6 Strictly Local Transactions

When the destination of a message is a process on the same host as the source process, the network interface is of course not used. However, the queue management routines behave in most respects as though the message were being transmitted between distinct hosts. A separate transaction record is created for the "incoming" message (though not necessarily a separate copy of its text) when it is passed from the output handlers to the input handlers. It may even happen that the message will be "Held by sender", if the buffers alloted to the destination process are too full to permit accepting it. Locally held messages are given priority over others, however, since the response to a local "XMIT" request is instantaneous.

2.3.3 Handling of Alarms

Alarm handling in MSG is just like message handling, but without several of the complications. Generic addressing and sequencing are not issues. Text buffering is no problem since alarms have only a fixed length, very short alarm code. The MSG-to-MSG protocol for alarms does not permit requests analogous to MESS-HOLD, so the state diagrams for input and output alarms are trivial. Handling of local (intraMSG) alarm transmissions is identical to that for messages. The only special feature of alarm transactions is that a process may choose to reject all incoming alarms by setting its IAccept flag to FALSE (using the AcceptAlarms primitive). MSG simply checks this flag before accepting an alarm, and rejects the alarm when the flag is off.

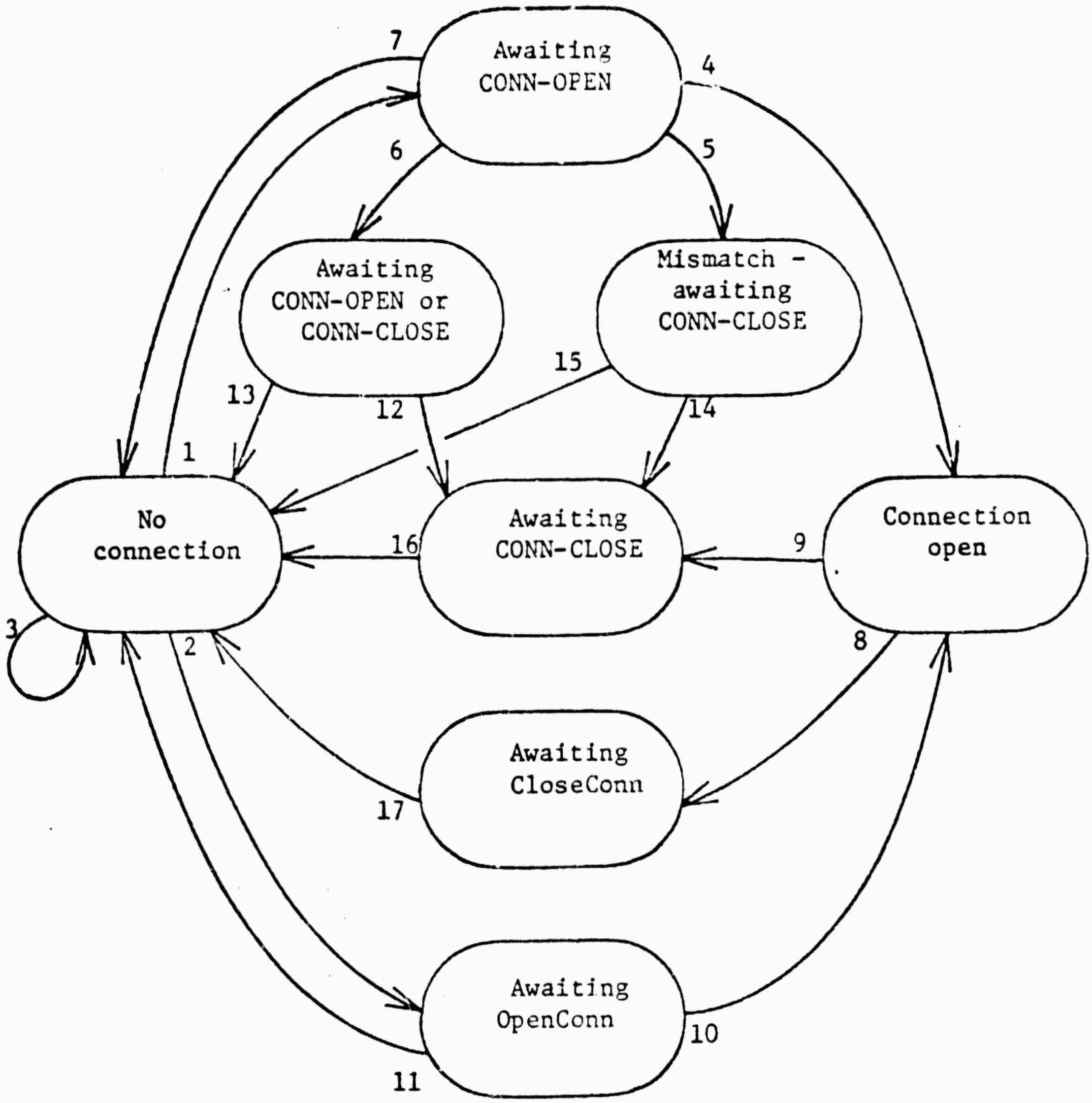
The model uses the name "ReceiveAlarm" instead of "EnableAlarm" (used in the MSG specification), to emphasize the analogy with ReceiveMess. The routines in QUEUE that deal with alarms are EnQOutputAlarm, EnQInputAlarm, RecordALARM\OK, and RecordALARM\REJ.

2.3.4 Handling of Direct Connections

The establishment of direct connections between MSG user processes differs in several respects from the transmission of messages or alarms. The roles of sender and receiver do not exist. Instead, both processes agree to open the connection by executing matching OpenConn primitives, and they close it (in the normal case) by issuing similar CloseConn primitives. A connection transaction, unlike a message or alarm, does not end with the completion of the primitive that initiated it, since a record must be maintained for use in closing the connection. Special provision is also necessary for signalling a local process that a connection has been broken other than by a normal closing sequence. MSG's handling of direct connections is diagrammed in figure 4.

The events that give rise to transitions are the execution of OpenConn and CloseConn primitives by the local process and the receipt of CONN-OPEN, CONN-CLOSE, and CONN-REJ protocol items from a remote MSG. MSG instances exchange CONN-OPEN items before opening a connection between the two processes requesting it. They exchange CONN-CLOSE items either when requested to close a connection or when a connection cannot be opened because of a mismatch in parameters. CONN-REJ is used to reject either a CONN-OPEN or a CONN-CLOSE that is invalid.

In the initial state of figure 4, labelled "No connection", the receipt of a CONN-OPEN item (transition 2) creates a transaction record and leaves it in state "Awaiting OpenConn", that is, awaiting local execution of an OpenConn primitive for a matching connection. When the



1: OpenConn/send CONN-OPEN
2: CONN-OPEN/ --
3: CONN-CLOSE/send CONN-REJ
CONN-REJ/ --
4: CONN-OPEN/open connection,
complete OpenConn
5: CONN-OPEN/send CONN-CLOSE
6: CloseConn/send CONN-CLOSE,
abort OpenConn
7: CONN-CLOSE/abort OpenConn,
send CONN-CLOSE
CONN-REJ/abort OpenConn
Timeout/abort OpenConn
send CONN-CLOSE
8: CONN-CLOSE/ --
9: CloseConn/send CONN-CLOSE
Error/send CONN-CLOSE
10: OpenConn/send CONN-OPEN,
complete OpenConn

11: OpenConn/send CONN-REJ,
abort OpenConn
Timeout/send CONN-REJ
12: CONN-OPEN/ --
13: CONN-CLOSE/abort CloseConn
CONN-REJ/abort CloseConn
Timeout/abort CloseConn
14: CloseConn/abort OpenConn
15: CONN-CLOSE/abort OpenConn
CONN-REJ/abort OpenConn
Timeout/abort OpenConn
16: CONN-CLOSE/complete event
CONN-REJ/complete event
Timeout/complete event
17: CloseConn/send CONN-CLOSE,
close connection,
complete CloseConn
Timeout/close connection,
signal broken connection

Figure 4. State Transitions for Direct Connections

local open is issued, the connection types, directions, and byte sizes supplied by the two sides are compared. If they match, the transaction is placed in the "Connection open" state (transition 10), the network connection is opened (by the process delivery server), and the open primitive is completed normally. If the connection specifications do not match, then a CONN-REJ is sent to the remote MSG, and the OpenConn is aborted (transition 11).

Other possibilities while "Awaiting OpenConn" are that the remote MSG may wish to close the connection before it is open by sending CONN-CLOSE, or that the local transaction may time out. In the former case, MSG responds with an answering CONN-CLOSE; in the latter, it sends CONN-REJ as a reply to the original CONN-OPEN. In either case, the transaction is deleted from the data base (transition 11).

If the local open precedes the remote open, the state of the transaction reaches "Connection open" via transitions 1 and 4 and the intermediate state "Awaiting CONN-OPEN". In this case, however, if connection parameters do not match, a full-fledged connection close sequence is used, since a full open sequence will have been completed by the time the mismatch is discovered. The local MSG sends a CONN-CLOSE item (transition 5) to terminate the transaction, and places the transaction in the state "Mismatch - awaiting CONN-CLOSE" to wait for an answering CONN-CLOSE.

If, while MSG is "Awaiting CONN-OPEN", a CONN-REJ is received instead, or if the transaction times out waiting for a reply, the local OpenConn is aborted and the transaction expunged (transition 7).

Once the connection is open, an exchange of CONN-CLOSE items will lead to its orderly closing. The path from "Connection open" to "No connection" leads either through "Awaiting CloseConn" or "Awaiting CONN-CLOSE" (transitions 8 or 9), depending on the order of events. If the transaction times out while a local close is awaited, a special "connection broken" signal will be sent to the user process, if such a signal has been specified in opening the connection.

The transition diagram is complicated by the fact that it is possible for a user process to execute an OpenConn primitive and then a CloseConn before the OpenConn has completed. There are two states in which an OpenConn can be pending, "Awaiting CONN-OPEN" and "Mismatch - awaiting CONN-CLOSE". In both cases, the pending open is aborted immediately when the CloseConn is issued. From "Mismatch - awaiting CONN-CLOSE", the successor state in this case (transition 14) is "Awaiting CONN-CLOSE", the usual intermediate state when a CloseConn is pending. The CONN-CLOSE item from the local to the remote MSG will already have been sent when the mismatch was discovered. From "Awaiting CONN-OPEN" a CloseConn takes the transaction into the special state labelled "Awaiting CONN-OPEN or CONN-CLOSE" (transition 13). A CONN-CLOSE is sent by the local MSG, but it may or may not reach the remote MSG in time to prevent a CONN-OPEN in reply to the original CONN-OPEN sent. The special state is included to receive and ignore that reply, if it comes (transition 12).

The routines in QUEUE that implement handling of direct connections are EnQOutputOpenConn, EnQOutputCloseConn, EnQInputOpenConn, EnQInputCloseConn, and RecordCONN\REJ.

2.4 Cancellation of Incomplete Transactions

The CANCEL module contains the routines that dispose of transactions that cannot be successfully completed. This may happen because 1) the user Rescinds the transaction, 2) the user process terminates and a StopMe is performed, 3) a protocol item is waiting to be output and the host to which the item is addressed dies, or 4) the transaction is timed and the timer expires. When a transaction is cancelled two basic types of action are necessary -- the transaction must be finally disposed of locally, and the remote MSG must also cancel its version of the transaction. The precise form of the action depends on the transaction's state.

Local cleanup of a transaction generally consists of removing it from its owning process's transaction lists (the owning process is the one for which the transaction was created and whose lists contain it) and, if it is a pending event, delivering it to that process. For example, cancelling a transaction in the "Awaiting CONN-OPEN" state involves removing it from its owning process's connection set and completing the OpenConn pending event that initiated it. Local message transactions are treated specially. They are exceptional in that both source and destination transactions are available, and are thus both cancelled locally at the same time.

Remote cancellation is generally accomplished by sending a protocol item to the remote MSG. For example, sending a MESS-CANCEL item cancels a transaction in the "Awaiting XMIT" state. It may be that a prior protocol item related to the transaction being cancelled is waiting to be sent, in which case the item can simply be changed to reflect cancellation. For example, a transaction may be in the "Awaiting XMIT" state with a HOLD-OK item waiting to be sent -- changing the item to MESS-CANCEL cancels the transaction. It is also the case that cancellation can be effected by not sending a protocol item. Such an item, initiating an MSG-MSG negotiation and waiting to be sent, can be removed and the transaction nipped in the bud. For example, a transaction may be in the "Awaiting CONN-OPEN" state with a CONN-OPEN item waiting to be sent -- removing the CONN-OPEN item cancels the transaction before the remote MSG has heard anything. A transaction may also be cancelled by doing nothing, relying on the system's response to the receipt of spurious protocol items. For example, a transaction in the "Awaiting HOLD-OK" state can be cancelled simply by removing all local knowledge of the transaction. Later, if the remote MSG sends a HOLD-OK item for a transaction that no longer exists, the system will reply with a MESS-REJ.

The four routines that call LocalCancel and RemoteCancel reflect the four causes for transaction cancellation -- RescindPendingEvent, StopTransaction, HostDeadTransaction and TimeoutTransaction. The first two are invoked from the user call servers, the third from the network output servers, and the fourth from the timeout handler -- from at least three MSG paths. This gives rise to contention issues. In order to dispose of a transaction both the transaction record and the record for its owning process must be seized. The user call server has the process and needs the transaction, while the network output server and the timeout handler have the transaction and need the process. The user call server is given priority. The network output server and timeout handler do a non-blocking test-seize of the transaction and its owning process. If the test fails they go on to other transactions, returning later and trying again.

Whether or not a pending event is rescissible depends on whether the event can, with certainty, be cancelled remotely. For example, an event in the "Awaiting XMIT" state can be cancelled with certainty by sending a MESS-CANCEL item. On the other hand, an event in the "Awaiting MESS-OK" state, with a MESS protocol item already sent, cannot be rescinded because the message may already have been accepted remotely (with a MESS-OK response on the way). Sending a MESS-CANCEL item would not necessarily succeed. Using this certainty test, RescindPendingEvent determines, based on the state of the event passed to it, whether that event is rescissible.

The CANCEL module's province also includes transaction timing. Transactions are timed so that users receive definitive dispositions of pending events without having to wait indefinitely in uncertainty, and so that the MSG resources used by a transaction may be freed after an inordinate delay in completing the transaction. A transaction is timed when it is in the queue of the timeout handler. The queue is sorted by timeout deadline, the transaction timing out earliest being at the front of the queue. StartTiming inserts an entry into the timer queue and StopTiming removes one.

TimeoutHandler, the routine comprising the timeout handler path, processes transactions which have timed out. The SIGNAL mechanism is used to awaken TimeoutHandler when timeout occurs. TimeoutHandler obtains and WAITs on a signal activated when the deadline of the transaction at the front of the timer queue expires, and cancels the transaction when the signal is received. TimeoutHandler WAITs additionally for a signal from StartTiming. If the transaction passed to StartTiming will time out earlier than the earliest existing timer queue entry, TimeoutHandler must be notified to wait for the new, earlier deadline rather than the old, later one.

2.5 The Network Interface

The routines that maintain communication with other MSGs are found in the REMOTE module. They establish inter-MSG network connections and exchange protocol items over them.

Connections are opened for two reasons. A local user process may wish to communicate with a remote one or a host with which no connection exists, in which case the local MSG initiates an Initial Connection Protocol sequence with the MSG on the desired host. Alternatively, a remote MSG may desire to establish a connection, starting an ICP sequence to which the local MSG must respond.

When a connection is initiated locally, the user call server enters the remote host in HostS, assigns the host to a network server pair, possibly creating new servers for this purpose, and notifies the output server, via a special entry in the server's delivery queue, that an ICP should be initiated. (This occurs in the SeizeHostHandle routine).

The output server performs the ICP request, establishing the sockets to be used for the connection (in RequestICP). If the ICP request should fail, all protocol items waiting to be sent to the unresponsive host are cancelled via host dead action. The other MSG's authentication demand, in response to the ICP request, is handled asynchronously by the authentication handler path (AuthenticationHandler).

When a remote MSG initiates a connection the ICP handler path (ICPHandler) responds to its request. It authenticates the requesting entity as an MSG, establishes the sockets to be used for the connection, makes the HostS entry, allocates servers, and finally notifies the output server, by enqueuing a special entry in its delivery queue, that a connection should be opened.

In both cases the output server (in the routine NewHost) opens the actual MSG-to-MSG connection, exchanges synchronization information with the remote MSG, cancels transactions to the new host with the wrong incarnation number, and notifies the input server that the new host exists.

The basic action taken when an error occurs on a network connection is the closing of the connection and the deallocation of the socket. This is done by the server that detects the error. Additionally, if the connection closed was the only one to that remote host, the connection is replaced by entering a Start ICP item at the front of the queue of the appropriate output server.

When properly functioning connections with another MSG exist they are used for exchanging protocol items, the units of inter-MSG com ication. The process of sending and receiving protocol items has two parts. Internal data structures must be converted to and from the external protocol formats recognized by all MSGs, and those external items must be sent and received over the network. In the abstract model, these conversions are defined in terms of pseudo data structures, thus making them non-procedural and separate from network data transmission. This method of specifying conversion provides clarity and modularity, aiding understanding and maintenance.

The conversion specifications are used as templates over buffers containing network data, imposing order on an undifferentiated mass of bits. They are defined in a form similar to the EL1 STRUCT operator. The FORMAT operator takes a list of fields, each field corresponding to one in the external protocol item being converted. Fields in the FORMAT definition appear in the same order as in the external item. In general a field is described by a field name, the data type of the entity represented by the field, and the length in network bytes of the field in the external protocol item. For example, in the definition of MessFormat, the first field is Length, which contains the number of network bytes in the item. Its desired type is

ShortInt and it is two network bytes long.

Some fields do not have this form. Boolean fields have no network byte length -- they are known to be individual bits. Other fields consist of several subfields occurring in a common pattern. Such fields have, instead of a mode and length, a "~" (indicating the existence of subfields) and the name of the FORMAT where the subfields are defined. For example, in MessFormat SourceProcess and DestProcess have the structure of an MSGProcessName, which is defined in a separate FORMAT. Other fields are converted by routines. In some cases these routines are named in and used by FORMAT. Such routines, preceded by the "^" operator, are specified in the type position of the field using them. For example, in MSGProcessName GenericName is converted by GenericClass. In other cases conversion routines are invoked independent of FORMAT. Fields converted in such a manner appear in FORMAT definitions as documentation. They are indicated by a "---" in the type position. For example, the message text field in MessFormat is handled by special text moving routines, but the text's presence is documented by the Text field.

Many external protocol fields can be converted and assigned to and from their internal representations directly, without further processing. For example, in ConnOpenFormat ConnID is logically equivalent to ConnID in ConnBlock, the target internal representation. On the other hand, ConnOpenFormat's Type Booleans do not correspond to ConnBlock's ConnType and ConnDirection, with further processing necessary. Those fields that can be assigned directly are indicated in a FORMAT by a "@". They must have the same name, mode and meaning as the target internal field. Such fields are susceptible to collective assignment, where a whole item is assigned to and from its internal equivalent without individual fields being mentioned.

FORMAT-defined conversions are used in the network input and output servers. They are invoked by the "¦¦", "¦", "<¦¦" and "¦¦>" operators. The ¦¦ operator specifies the FORMAT to be imposed on the network buffer, and the ¦ operator references a particular field in the given FORMAT. For example, in ProtocolInput the Header FORMAT is used (¦¦) to retrieve (¦) the Command field in it. The <¦¦ and ¦¦> operators are used to perform the collective assignment. They obviate the need for using the ¦ operator on every field. For example, in InputMess the <¦¦ operator assigns all @-marked fields in MessFormat from the network data buffer to the internal MessBlock, and in OutputMess ¦¦> does the reverse. Both operators also set the FORMAT to be used by the ¦ operator, so that after <¦¦ or ¦¦>, ¦ may be employed for individual fields. For example, in InputConnOpen translation between the external Type Booleans and the internal ConnType and ConnDirection fields is performed using ¦.

Both the network input server and the network output server have a similar structure. Each is organized around a dispatcher, which transfers control to routines specialized for processing specific protocol items. The input dispatcher, ProtocolInput, waits for an item to come in on one of its network channels, while the output dispatcher, ProtocolOutput, sends out items put in its delivery queue. The item-specific routines perform FORMAT-defined assignment to and from the network data buffer and the proper transaction record and, in the case of input, call the appropriate QUEUE routine.

3. The Language of the Model

The MSG abstract model is written in EL1 [Manual], the base language of the ECL system, which hosts the testbed for our program development experiments. During refinement, EL1 text will be reduced to SPECL, a systems programming version of EL1, and then into BLISS, an implementation language. BLISS was chosen because there are BLISS compilers for both of the machines for which we plan to make MSG realizations.

Because the systems resulting from refinement must be independent of the ECL runtime facilities, we have taken care to ensure that dependence on certain ECL features will be easy to remove. The model uses explicit freeing of records allocated in the data base, for example, because garbage collection will usually not be practical in a concrete instance, and because determining by mechanical analysis when storage can be reclaimed remains a difficult research problem.

The modelling language also includes extensions to EL1 which aid readability and provide the freedom to refine certain expressions in a variety of ways. Some of these extensions are specially handled by the tools used in refinement.

3.1 Data Type Generators

The model makes use of operators for type creation that are not part of EL1, but that have behaviors related to similarly named base language type generators. Pointer, for example, constructs types whose instances behave like pointers, but could be implemented either as machine addresses or table indices. Value is the operator used to dereference an abstract pointer. Sequence denotes a class of objects that are homogeneous indexable collections, without pinning down whether its members are of fixed or variable length. Union types represent objects whose types are not fixed until runtime; type tags and hidden pointer levels may be used, or it may be possible to avoid this overhead. Enumeration produces a type whose elements are members of a set of tokens fixed when the type is generated. Constants of an enumeration type are delimited by double quotation marks. (The EL1 builtin type SYMBOL, whose literals are normally indicated by double quotes, is not used per se in the model since it implies runtime hash table maintenance.)

The type operator HAS creates heterogeneous record types from tuples containing field names and component types. HAS will also be used later in MSG development to effect type refinement -- adding additional fields to existing types to satisfy needs that arise at lower levels of abstraction.

3.2 Set and Queue Abstractions

The model makes use of two abstract type generators that are neither explicitly defined nor analogous to base language operators. Set(M) represents an unordered, non-repeating collection of objects of type M. Insert and Remove are generic procedures that enter and delete elements from sets. Queue(M) denotes an ordered, fixed-capacity collection of M-values that need not be unique within the collection. EnQ and DeQ are generic procedures that enter and delete queue elements. Front is a function that returns the least recently entered element of the queue it is applied to. IsFullQ is a predicate that returns TRUE when applied to a queue that has reached its capacity.

The names of sets in the model are conventionally terminated with a capital S (e.g., ConnectionS); names of queues end with a capital Q (e.g., InputAlarmQ).

3.3 Iteration Expressions

To ensure that the model routines that deal with collection abstractions remain independent of the representations chosen, we use abstract iteration expressions. The general form of these loops is

FOREACH E AT P IN C DO S1; ... ; Sk END

This expression will bind the identifier E to successive elements of the collection C and execute the statements S1 through Sk once for each such binding. As with EL1 iterations, any statement Si may have the form B => R, where B and R are expressions and B yields a Boolean result. If, when such a statement is executed, B evaluates to TRUE, then the loop is terminated and the value of R becomes the value of the loop.

The clause AT P is optional. When it appears, the identifier P will be bound on successive iterations to a state variable whose exact definition depends on the data type of the collection C. This state variable can be used to efficiently delete and insert elements in the collection during an iteration. For instance, the generic procedure DeQAt(P), used during an iteration over a queue C, would remove the currently bound element from the queue. It would be equivalent to DeQ(C, E), but would be more efficient.

Normally, the order in which the elements of a collection are bound by an iterator is not defined. To guarantee scanning of queues in least-to-most recent order, we use

```
FOREACH E FromFrontOf Q DO ...  END
```

APPENDIX: Text of the MSG Model

```
' - - - - - - - - - - - - - - - - - - - - -

.       MSG ... Abstract Version

.       The abstract version occupies the following modules:

.       GLOBAL  - Common definitions and tables
.                   (This module.)
.       DRIVER  - MSG initiation and main routine
.       PROCESS - User program linkages to MSG primitives
.       LOCAL   - Code for local process servers
.       QUEUE   - Transaction queue management routines
.       REMOTE  - Code for remote host servers
.       CANCEL  - Pending event timeout management

- - - - - - - - - - - - - - - - - - - - - ' */ "";

' - - - - - - - - - - - - - - - - - - - - -

.       Sets, Queues, and Handles ...

.       In abstract MSG, the only difference between a "set" and
.       a "queue" is that the latter is expected to conform to some
.       queueing discipline as regards adding and deleting entries.
.       No commitment is made about their representations.

.       A "handle" designates a set or queue entry.  It has the
.       abstract properties of a pointer.  More than one set
.       and/or queue may contain the data referenced by a handle
.       and thus share the data referenced by the handle.
.       A handle is not necessarily represented as a physical
.       storage address or pointer.

- - - - - - - - - - - - - - - - - - - - - ' */ "";

^;
```

```
'       Type Definitions        ' */ "";

TransactionID <-
  '
  . Encoded version of TransactionHandle,
  . used to identify pending event for
  . Rescind primitive and as MSG network
  . protocol SourceID and DestID. ' */ . .;

ShortInt <- 'Short integer, unsigned' */ ...;

StateCode <-
  Enumeration("NullState", "Delivered", "Awaiting MESS-OK",
              "Awaiting XMIT", "Awaiting MESS",
              "Awaiting ReceiveMess", "Held by sender",
              "Awaiting prior MESS completion",
              "Awaiting HOLD-OK", "Awaiting retransmission",
              "Awaiting ALARM", "Awaiting ALARM-OK",
              "Awaiting ReceiveAlarm",
              "Awaiting CONN-OPEN or CONN-CLOSE",
              "Awaiting CONN-CLOSE",
              "Mismatch - awaiting CONN-CLOSE",
              "Awaiting CONN-OPEN", "Awaiting CloseConn",
              "Awaiting OpenConn", "Connection open");

ConnDirectionCode <- Enumeration("In", "Out", "InOut");

ConnTypeCode <-
  Enumeration("Binary", "UserTELNET", "ServerTELNET");

ProtocolCode <-
  Enumeration("NullProtocolCommand", "MESS", "MESS-OK",
              "MESS-REJ", "MESS-HOLD", "MESS-CANCEL", "XMIT",
              "ALARM", "ALARM-OK", "ALARM-REJ", "CONN-OPEN",
              "CONN-CLOSE", "CONN-REJ", "Start ICP",
              "Finish ICP");

ConnIDCode <- ShortInt;

HostCode <- ShortInt;

AlarmCode <- ShortInt;

GenericClassCode <- Enumeration("NullCode", ...);
```

```
ReasonCode <-
  Enumeration("Incomplete", "Normal", "ACKing your close",
             "Alarm queue for process is full",
             "Already have connection of that ID",
             "Bad incarnation number on destination process",
             "Bytesize given is invalid",
             "Connection ID inconsistent with local connection info",
             "Connection byte size mismatch",
             "Connection not to process named",
             "Connection not yet open",
             "Connection type mismatch", "Delivered",
             "Destination name/handling - generic/specific mismatch",
             "Destination process connection limit reached",
             "Destination process unknown", "Event rescinded",
             "Event timed out",
             "Failure of prior sequenced message",
             "Failure of prior stream marked message",
             "Generic class given is invalid",
             "Handling given is invalid", "Host died",
             "Insufficient resources to complete command",
             "Invalid connection byte size",
             "Invalid connection type",
             "Invalid host address in process name",
             "Message rescinded or timed out",
             "Open superseded by close",
             "Output messages quenched",
             "Process not accepting alarms now",
             "Process terminated", "Redundant close",
             "Referenced connection transaction does not exist",
             "Refused by all suitable hosts",
             "Send message stream out of synch",
             "SendGenericMessage", "SendSpecificMessage",
             "Sequenced send message stream out of synch",
             "Signal given is invalid", "Superseded by close",
             "That generic class not supported here",
             "Timed out waiting for your CONNECTION-OPEN",
             "Transaction rescinded", "Transaction timed out",
             "Unable to Rescind", "Unknown connection");
```

```
HandlingCode <-
  Enumeration("Normal", "Sequence", "StreamMark", "QWait");

String <- Sequence(Character);

StringPtr <- Pointer(String);

ProcessName HAS
  < Host(HostCode), Incarnation(ShortInt), Instance(ShortInt),
    GenericName(Union(StringPtr, GenericClassCode)) >;

ProcessHandle <- Pointer(ProcessBlock);

ProcessBlock HAS
  < Name(ProcessName), UserChannel(ChannelHandle),
    OutputMessQ(Queue(MessHandle)),
    OutputAlarmS(Set(AlarmHandle)), ConnectionS(Set(ConnHandle)),
    InputMessQ(Queue(MessHandle))
    InputAlarmQ(Queue(AlarmHandle)),
    InhibitedDestS(Set(STRUCT(DestProcess:ProcessName,
                              InhibitAll:BOOL))),
    DeliveryServer(ServerHandle)  IAccept(BOOL),
    FreeBufferSize(ShortInt), CommittedBufferSpace(ShortInt),
    ReceiveMessQ(Queue(MessHandle)),
    ReceiveAlarmQ(Queue(AlarmHandle)),
    CloseConnS(Set(ConnHandle))  Termination(TermHandle) >;

GenericHandle <- Pointer(GenericBlock);

GenericBlock HAS
  < Class(StringPtr), Code(GenericClassCode),
    HostS(Set(HostCode)), RunFile(Filename),
    ServerS(Set(ServerHandle)), InputMessQ(Queue(MessHandle)),
    FreeBufferSize(ShortInt), CommittedBufferSpace(ShortInt),
    ReceiveMessQ(MessHandle), Unsupported(BOOL) >;

ServerHandle <-
  Pointer(Union(InputServerBlock, DeliveryServerBlock));

DeliveryServerBlock HAS
  < DeliveryQ(Queue(TransactionHandle))
    Server(Procedure), ChannelS(Set(ChannelHandle)),
    Running(BOOL), WakeUpSignal(SignalType) >;

InputServerBlock HAS
  < Server(Procedure), ChannelS(Set(ChannelHandle)),
    WakeUpSignal(SignalType) >;

HostHandle <- Pointer(HostBlock);

HostBlock HAS
  < Host(HostCode), Incarnation(ShortInt),
    ConnectionS(Set(ChannelHandle)),
    DeliveryServer(ServerHandle), InputServer(ServerHandle) >;
```

```
Abstract Model of MSG
Module GLOBAL


ChannelHandle <- Pointer(ChannelBlock);

ChannelBlock HAS < User(UserHandle) >;

ContactHandle <- Pointer(ContactBlock);

ContactBlock HAS
  < RemoteHost(HostHandle), RemoteSocket(Integer),
    LocalSocket(Integer), ProtocolCommand(ProtocolCode) >;

MessHandle <- Pointer(MessBlock);

MessBlock HAS
  < Op(OpCode), State(StateCode), Signal(SignalType),
    Disposition(ReasonCode), Deadline(Integer),
    SourceProcess(ProcessName), DestProcess(ProcessName),
    SourceID(TransactionID), DestID(TransactionID),
    Text(StringPtr), TextLength(ShortInt), IsGeneric(BOOL),
    IsSequenced(BOOL), IsMarked(BOOL), NoHold(BOOL),
    HoldOkay(BOOL), QWait(BOOL), IsHostless(BOOL) >;
```

```
ConnHandle <- Pointer(ConnBlock);

ConnBlock HAS
  < Op(OpCode), State(StateCode), Signal(SignalType),
    Disposition(ReasonCode), Deadline(Integer),
    SourceProcess(ProcessName), DestProcess(ProcessName),
    SourceID(TransactionID), DestID(TransactionID),
    ConnID(ConnIDCode), ConnType(ConnTypeCode),
    ConnDirection(ConnDirectionCode), ConnBytesize(ShortInt),
    EndSignal(SignalType), Connection(ChannelHandle),
    LocalSocket(Integer), RemoteSocket(Integer),
    ProtocolCommand(ProtocolCode) >;

AlarmHandle <- Pointer(AlarmBlock);

AlarmBlock HAS
  < Op(OpCode), State(StateCode), Signal(SignalType),
    Disposition(ReasonCode), Deadline(Integer),
    SourceProcess(ProcessName), DestProcess(ProcessName),
    SourceID(TransactionID), DestID(TransactionID),
    Alarm(AlarmCode) >;

TermHandle <- Pointer(TermBlock);

TermBlock HAS
  < Op(OpCode), Signal(SignalType), Disposition(ReasonCode),
    UserChannel(ChannelHandle) >;

UserHandle <- Union(ProcessHandle, HostHandle);

TransactionHandle <-
  Union(MessHandle, AlarmHandle, ConnHandle, TermHandle,
        ContactHandle);

DestHandle <- Union(ProcessHandle, GenericHandle);

^;
```

```
'       DATA    ' */ "";

ProcessTable <- CONST(Sequence(ProcessHandle));

HostS <- CONST(Set(HostHandle));

GenericTable <- CONST(Sequence(GenericHandle));

ServerTable <- CONST(Sequence(ServerHandle));

InitialServerS <-
  CONST(Set(Procedure) OF AuthenticationHandler, ICPHandler);

TimerQ <- CONST(Queue(TransactionHandle));

TransactionTable <- CONST(Sequence(TransactionHandle));

^;
```

```
'       PRIMITIVE PROCEDURES    ' */ "";

' - - - - - - - - - - - - - - - - - - - -

.       Abstract Channels ...

.       The following define abstract
.       channels, used for communication of data between
.       paths.  They are modelled after the channels used
.       in the CCN system.


.       CHOPEN is used to open a channels.  Each path
.       supplies the same ChannelID.  The channel is closed
.       if a CHCLOSE is issued from either side, but
.       will be closed only after data in the channel has
.       been received.

.       Output on the channel is assumed to be blocking
.       SEND (or =|>), while input blocks only if the
.       third argument to RECEIVE (or <|=) is TRUE.
.       Nonblocking input is used in LOCAL to supply
.       locations for the results returned at completion
.       of a pending event.

.       ConnByteSize defaults according to the Type of
.       the channel.

.       =|> and <|= are synonyms for SEND and RECEIVE.
.       In certain implementations (such as TENEX), they
.       will be refined away and be replaced by assignment.

- - - - - - - - - - - - - - - - - - - - - -' */ -;

CHOPEN <-
  EXPR(Id:ChannelID                        /* 'Channel spec',
       Type:ConnTypeCode                   /* 'Operating mode',
       ConnBytesize:ShortInt;
       ChannelHandle) ...;

SEND <-
  EXPR(Data:ANY                            /* 'Data to be transmitted',
       Channel:ChannelHandle;
       ANY) ...;

=|> <- SEND;

RECEIVE <-
  EXPR(Data:ANY                            /* 'Where to put received data',
       Channel:ChannelHandle,
       Blocks:BOOL                         /* 'Blocks if TRUE';
       ANY) ...;
```

```
<|= <- RECEIVE;

<||== <-
  '
  .  Infix operator for blocking RECEIVE
  ' */
    EXPR(Data:ANY, Channel:ChannelHandle; ANY)
      RECEIVE(Data, Channel, TRUE);

MakeId <-
  '
  .  Used to construct a ChannelID
  .  for a network CHOPEN. ' */
    EXPR(Host:Integer                      /* 'Remote host (0 for any)',
         RSocket:Integer                   /* 'Remote socket (0 for listen)',
         LSocket:Integer                   /* 'Local Socket';
         ChannelID) ...;
```

```
' - - - - - - - - - - - - - - - - - - - - -

.       DRIVER - MSG Initialization and main
.                    routine

- - - - - - - - - - - - - - - - - - - - */ -';

InitiateMSG <-
  EXPR(Restart:BOOL)
    BEGIN
      InitializeGenericS()             /* 'Read the data from disk';
      (Restart -> KillUserProcesses()) /* '
                                          . Issue termination signals.
                                          . Also, drop any remote host
                                          .  connections remaining.';
      SetNewIncarnationNumber();
      InitializeServerS()              /* '
                                          . Set up set of servers to
                                          . start up and run';
      PollAndSleep()             /* 'If servers are interrupt-driven,
                                    enable them.  Else, loop over ServerS
                                    and sleep until a signal for
                                    one or more is received. ';
    END;

InitializeServerS <-
  EXPR()
    FOREACH S IN InitialServerS
      DO
        DECL SH:ServerHandle LIKE Allocate(ServerBlock);
        SH.Server <- S;
        Release(SH);
      END;
```

```
' - - - - - - - - - - - - - - - - - - - - -

.       PROCESS ... User Program MSG Calls

- - - - - - - - - - - - - - - - - - - - - - ' */ -;

' - - - - - - - - - - - - - - - - - - - - -

.       PROCESS contains code and data that
.       are included in the user program
.       to call MSG primitives.  The arguments,
.       result type, and the actual abstract
.       call for each MSG primitive are given
.       first below.  Each primitive call is
.       represented as a call of the abstract
.       routine PCall.  Each call is of
.       the form:

.         PCall(PrimitiveName,SendList,
.               ReceiveList,IsPendingEvent);

.       Each PCall call will be refined according
.       to the exact mechanism chosen to communi-
.       cate between the user process and local
.       MSG in a given operating system and
.       language implementation.

.       The SendList and ReceiveList include
.       all arguments of the primitive call,
.       classified as to whether each argument
.       represents data sent to MSG by the process
.       or data received from MSG as a result
.       of executing the MSG primitive.

.       IsPendingEvent is TRUE for those MSG
        primitives which create pending events.
,       The communication algorithm is discussed in
.       the commentary preceding PCall in this file.
' */ -;

^;
```

```
SendGenericMessage <-
  EXPR(Text:StringPtr,
       DestProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode,
       Timer:Integer,
       QWait:HandlingCode;
       ReasonCode)
    PCall("SendGenericMessage",
         < Text, DestProcess, Signal, Timer, QWait >,
         < DestProcess, Disposition >, TRUE);

ReceiveGenericMessage <-
  EXPR(Text:StringPtr,
       SourceProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode,
       Timer:Integer;
       ReasonCode)
    PCall("ReceiveGenericMessage", < Signal, Timer >,
         < Text, SourceProcess, Disposition >, TRUE);

SendSpecificMessage <-
  EXPR(Text:StringPtr,
       DestProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode,
       Timer:Integer,
       Handling:HandlingCode;
       ReasonCode)
    PCall("SendSpecificMessage",
         < Text, DestProcess, Signal, Timer, Handling >,
         < Disposition >, TRUE);

ReceiveSpecificMessage <-
  EXPR(Text:StringPtr,
       SourceProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode
       Timer:Integer,
       Handling:HandlingCode;
       ReasonCode)
    PCall("ReceiveSpecificMessage", < Signal, Timer >,
         < Text, SourceProcess, Disposition, Handling >, TRUE);

^;
```

```
SendAlarm <-
  EXPR(Alarm:AlarmCode,
       DestProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode;
       ReasonCode)
    PCall("SendAlarm", < Alarm, DestProcess, Signal, Timer >,
          < Disposition >, TRUE);

ReceiveAlarm <-
  EXPR(Alarm:AlarmCode,
       SourceProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode,
       Timer:Integer;
       ReasonCode)
    PCall("ReceiveAlarm", < Signal, Timer >,
          < Alarm, SourceProcess, Disposition >, TRUE);

OpenConnection <-
  EXPR(ConnID:ConnIDCode,
       ConnType:ConnTypeCode,
       ConnDirection:ConnDirectionCode,
       ConnBytesize:Integer,
       DestProcess:ProcessName,
       EndSignal:SignalType,
       Signal:SignalType,
       PendingEventID:TransactionID,
       Disposition:ReasonCode,
       Timer:Integer;
       ReasonCode)
    PCall("OpenConnection",
          < ConnID, ConnType, ConnDirection, ConnBytesize,
            DestProcess, EndSignal, Signal, Timer >,
          < Disposition >, TRUE);

CloseConnection <-
  EXPR(ConnID:Tag,
       DestProcess:ProcessName,
       Signal:SignalType,
       PendingEventID:Tag,
       Disposition:ReasonCode,
       Timer:Integer;
       ReasonCode)
    PCall("CloseConnection",
          < ConnID, DestProcess, Signal, Timer >,
          < Disposition >, TRUE);

^;
```

```
TerminationSignal <-
  EXPR(Signal:SignalType,
       Timer:Integer,
       Disposition:ReasonCode;
       ReasonCode)
    PCall("TerminationSignal", < Signal, Timer >,
          < Disposition >, TRUE);

'
.       The following primitives do not
.       create pending events.
' */ -;

StopMe <- EXPR(; ReasonCode) PCall("StopMe");

Rescind <-
  EXPR(PendingEventID:TransactionID; ReasonCode)
    PCall("Rescind", < PendingEventID >, < Result >);

AcceptAlarms <-
  EXPR(IAccept:BOOL; ReasonCode)
    PCall("AcceptAlarms", < IAccept >);

Resynch <-
  EXPR(DestProcess:ProcessName; ReasonCode)
    PCall("Resynch", < DestProcess >);

WhoAmI <-
  EXPR(Name:ProcessName; ReasonCode)
    PCall("WhoAmI", NIL, < Name >);

^;
```

```
PCall <-
  '
  . The infix operators "=|>" and "<|=" denote
  . transmission of data from the user process
  . to MSG (SEND -- cf. CHOPEN in module GLOBAL)
  . and the reverse.  The general protocol is:

  . The operation name of the primitive and the
  . SendList are sent to MSG.  MSG validates the
  . arguments.  It returns a disposition which
  . may be:

  .  "Normal"  Execution is complete, and
  .            the ReceiveList return data (if any)
  .            should be received.  PCall blocks
  .            until reception is complete.

  .  "Incomplete"  A pending event has been created.
  .                The receive is issued to request
  .                the receive list data after
  .                completion of the pending event.
  .                If the Signal is blocking, the
  .                user process blocks; otherwise,
  .                PCall exits immediately.

  .  Other     An error has been noted by MSG.  The
  .            disposition data has been received.
  .            No pending event has been created.
  .            PCall exits immediately.

  . When a pending event has successfully
  . been created, MSG returns a PendingEventID
  . to the user.  It is used by the Rescind
  . primitive.  It is also used in PCall to
  . open (using CHOPEN) an individual abstract
  . channel over which the receive list data
  . is to be returned by MSG at pending
  . event completion.  MSG closes this channel
  . afterwards.  A separate channel for each
  . transaction is used to avoid cluttering the
  . abstract model with the details of multi-
  . plexing use of a single channel.
  ' */
  EXPR(Op:OpCode,
       SendList:List,
       ReceiveList:List,
       CreatesPendingEvent:Bool;
       ReasonCode)
    BEGIN
      ^;
```

```
        DECL Result:ReasonCode LIKE "Normal";
        '
        .  Send Op and SendList arguments
        .  to MSG and wait for validation
        .  result.
        ' */ -;
        Op =¦> MSGChannel;
        SendList =¦> MSGChannel;
        Result <¦¦== MSGChannel;
        /* '
           .    If "Incomplete" (when CreatesPE is TRUE),
           .    or "Normal", IsAbnormal() fails';
        IsAbnormal(Result) => Result;
        CreatesPendingEvent =>
          BEGIN
            '
            .  Receive PendingEventID for Rescind.
            .  Then open channel and ask for ReceiveList
            .  return data. ' */ -;
            PendingEventID <¦¦== MSGChannel;
            DECL CH:ChannelHandle LIKE
              CHOPEN(PendingEventID, In, Binary);
            ReceiveList <¦= CH;
            "Normal";
          END;
         '
         .  Receive any ReceiveList data.
         ' */ -;
         ReceiveList <¦¦== MSGChannel;
         Result;
       END;

 '                DATA              ' */ -;

 MSGChannel <-
   '
   .  Channel used to communicate with MSG.
   .  Opened in StartUserProcess.
   ' */ CONST(ChannelHandle);

 ^;
```

```
' - - - - - - - - - - - - - - - - - - - - -

.       LOCAL ... Local process primitive handlers

- - - - - - - - - - - - - - - - - - - - - ' */ -;

' - - - - - - - - - - - - - - - - - - - - -

.  LOCAL handles (via UserCallServer) user
.  process primitive requests (see PCall
.  in PROCESS) and delivery of results
.  of primitive requests (via UserDelivery-
.  Server.)

.  Handlers for primitives that create
.  pending events consist of a Start routine
.  (found by looking up the OpCode in
.  OpTable) and an End routine (in EndOpTable.)
.  The Start routine receives the SendList from
.  the user process, validates arguments,
.  creates the pending event, and calls
.  QUEUE to have the event enqueued for
.  further processing.  The End routine
.  handles completion of the pending event
.  by sending the ReceiveList data to
.  the user process.

.  Handlers for primitives that do not
.  create pending events (e.g., DoStopMe)
.  send any results to the user process;
.  in these cases, the PCall code blocks the
.  user until the results are received.

- - - - - - - - - - - - - - - - - - - - - ' */ -;

^;
```

```
'       TABLES          ' */ -;

OpCode <-
  Enumeration(SendGenericMessage, ReceiveGenericMessage,
              SendSpecificMessage, ReceiveSpecificMessage,
              SendAlarm, ReceiveAlarm, OpenConnection,
              CloseConnection, TerminationSignal, StopMe,
              Rescind, AcceptAlarms, Resynch, WhoAmI);

OpTable <-
  ' Table of procedures to handle user primitive requests.
  .  StartXXX starts and enqueues a pending event.
  .  DoXXX completes the corresponding primitive before
  .   returning.
  ' */
    CONST(Sequence(Procedure) OF
          StartSendMessage              /* 'Send Generic Message',
          StartReceiveMessage           /* 'Receive Generic Message',
          StartSendMessage              /* 'Send Specific Message',
          StartReceiveMessage           /* 'Receive Specific Message',
          StartSendAlarm                /* 'Send Alarm',
          StartReceiveAlarm             /* 'Receive (enable) Alarm',
          StartOpenConnection           /* 'Start Open Connection',
          StartCloseConnection          /* 'Close Connection',
          StartTerminationSignal        /* 'Termination Signal',
          DoStopMe                      /* 'Stop Me',
          DoRescind                     /* 'Rescind',
          DoAcceptAlarms                /* 'Accept Alarms',
          DoResynch                     /* 'Resynch',
          DoWhoAmI                      /* 'WhoAmI');

EndOpTable <-
  CONST(Sequence(Procedure) OF
        EndSendGenericMessage, EndReceiveGenericMessage,
        EndSendSpecificMessage, EndReceiveSpecificMessage,
        EndSendAlarm, EndReceiveAlarm, EndOpenConnection,
        EndCloseConnection, EndTerminationSignal,
        EndBrokenConnection             /* 'Not a primitive...
                                           . used to implement
                                           . EndSignal for OpenConnection');

TimerDefaults <-
  CONST(Sequence(ShortInt) SIZE LENGTH(EndOpTable));

^;
```

```
' - - - - - - - - - - - - - - - - - - - - -

.        Primitive Handlers

- - - - - - - - - - - - - - - - - - - - - - ' */ -;

StartSendMessage <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL M:MessBlock;
      M.SourceProcess <- SP.Name;
      DECL Handling:HandlingCode;
      < M.Text, M.DestProcess, M.Signal, M.Deadline, Handling > <|=
        UserChannel;
      ConvertTimer(Op, M.Deadline);
      DECL MH:MessHandle;
      M.Disposition <- ReasonCode <<
        BEGIN
          Op = "SendSpecificMessage" ->
            IsValidHost(M.DestProcess.Host) +>
              RETURN("Invalid host address in process name");
          CASE[Handling, Op]
            ["Normal"] => TRUE;
            ["Sequence", "SendSpecificMessage"] => M.IsSequenced;
            ["StreamMark", "SendSpecificMessage"] => M.IsMarked;
            ["QWait", "SendGenericMessage"] => M.IsGeneric;
            TRUE => RETURN("Handling given is invalid");
          END <- TRUE;
          OR(NOT M.IsGeneric,
              EncodeGenericClass(M.DestProcess.GenericClass),
              CheckBlankGenericFields(M.DestProcess)) #>
            "Generic class given is invalid";
          IsValidSignal(M.Signal) #> "Signal given is invalid";
          FOREACH C IN SP.InhibitedDestS
            REPEAT
              C.DestProcess = VAL(M.DestProcess) =>
                BEGIN
                  C.InhibitAll =>
                    RETURN("Send message stream out of synch");
                  M.Sequenced ->
                    RETURN("Sequenced send message stream out of synch");
                END;
            END;
          CheckUserQuench() +> "Output messages quenched";
          MH <- Allocate(MessHandle, M);
          "Incomplete";
        END;
      Disposition =|> UserChannel;
      IsAbnormal(Disposition) => NOTHING;
      MH.UserChannel <-
        CHOPEN(MH.SourceID <- AssignTransactionID(MH) =|>
                  UserChannel, "Out", "Binary");
      EnQOutputMess(MH, SP);
    END;
```

```
EndSendMessage <-
  EXPR(SP:ProcessHandle, MH:MessHandle)
    BEGIN
      DECL UserChannel:UserData LIKE SP.UserChannel;
      MH.IsGeneric ->
        BEGIN
          DecodeGenericClass(MH.DestProcess.GenericClass);
          MH.DestProcess =¦> UserChannel;
        END;
      MH.Disposition =¦> UserChannel;
      SIGNAL(MH.Signal);
      CHCLOSE(UserChannel);
      Free(MH);
    END;

StartReceiveMessage <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL M:MessBlock;
      DECL MH:MessHandle;
      DECL Disposition:ReasonCode;
      M.DestProcess <- SP.Name;
      < Signal, Deadline >(MH) <¦= UserChannel;
      ConvertTimer(Op, M.Deadline);
      Disposition <- ReasonCode <<
        BEGIN
          IsValid(MH.Signal) #> "Signal given is invalid";
          MH <- Allocate(MessHandle, M);
          "Incomplete";
        END;
      Disposition =¦> UserChannel;
      IsAbnormal(Disposition) => NOTHING;
      MH.UserChannel <-
        CHOPEN(MH.DestID <- AssignTransactionID(MH) =¦>
                  UserChannel, "Out", "Binary");
      EnQReceiveMess(MH, SP);
    END;

EndReceiveMessage <-
  EXPR(SP:ProcessHandle, MH:MessHandle)
    BEGIN
      DECL UserChannel:UserData LIKE SP.UserChannel;
      DecodeGenericClass(MH.SourceProcess.GenericClass);
      < Text, SourceProcess, Disposition >(MH) =¦> UserChannel;
      NOT MH.IsGeneric -> (EncodeHandling(MH) =¦> UserChannel);
      SIGNAL(MH.Signal);
      Free(MH);
    END;

^;
```

```
StartSendAlarm <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL A:AlarmBlock;
      DECL AH:AlarmHandle;
      DECL Disposition:ReasonCode;
      A.SourceProcess <- SP.Name;
      < Alarm, DestProcess, Signal, Deadline >(A) <¦=
        UserChannel;
      ConvertTimer(Op, A.Deadline);
      Disposition <- ReasonCode <<
        BEGIN
          IsValidSignal(A.Aignal) #> "Signal given is invalid";
          AH <- Allocate(AlarmHandle, A);
          "Incomplete";
        END;
      Disposition =¦> UserChannel;
      IsAbnormal(Disposition) => NOTHING;
      AH.UserChannel <-
        CHOPEN(AH.SourceID <- AssignTransactionID(AH) =¦>
                   UserChannel, "Out", "Binary");
      EnQOutputAlarm(AH, SP);
    END;

EndSendAlarm <-
  EXPR(SP:ProcessHandle, AH:AlarmHandle)
    BEGIN
      DECL UserChannel:UserData LIKE SP.UserChannel;
      < Disposition >(AH) =¦> UserChannel;
      SIGNAL(AH.Signal);
      Free(AH);
    END;

StartReceiveAlarm <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL A:AlarmBlock;
      DECL AH:AlarmHandle;
      DECL Disposition:ReasonCode;
      A.DestProcess <- SP.Name;
      < Signal, Deadline >(A) <¦= User Channel;
      ConvertTimer(Op, A.Deadline);
      Disposition <- ReasonCode <<
        BEGIN
          IsValidSignal(A.Signal) #> "Signal given is invalid";
          AH <- Allocate(AlarmHandle, A);
          "Incomplete";
        END;
      Disposition =¦> UserChannel;
      AH.UserChannel <-
        CHOPEN(AH.DestID <- AssignTransactionID(AH) =¦>
                   UserChannel, "Out", "Binary");
      EnQReceiveAlarm(AH, SP);
    END;
```

```
EndReceiveAlarm <-
  EXPR(SP:ProcessHandle, AH:AlarmHandle)
    BEGIN
      DECL UserChannel:UserData LIKE SP.UserChannel;
      DecodeGenericClass(AH.SourceProcess.GenericClass);
      < Alarm, SourceProcess, Disposition >(AH) =|> UserChannel;
      SIGNAL(AH.Signal);
      Free(AH);
    END;

 StartOpenConnection <-
   EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
     BEGIN
       DECL C:ConnBlock;
       DECL CH:ConnHandle;
       DECL Disposition:ReasonCode;
       C.SourceProcess <- SP.Name;
       < ConnID, ConnType, ConnDirection, ConnBytesize,
         DestProcess, EndSignal, Signal, Deadline >(C) <|=
         UserChannel;
       ConvertTimer(Op, C.Deadline);
       Disposition <- ReasonCode <<
         BEGIN
           IsValidSignal(C.Signal) #> "Signal given is invalid";
           IsValidHost(C.DestProcess.Host) +>
             "Invalid host address in process name";
           AssignLocalSocket(C.LocalSocket, SP);
           IsValidBytesize(C.ConnBytesize) #>
             "Bytesize given is invalid";
           CH <- Allocate(ConnectionHandle, C);
           "Incomplete ,
         END;
        Disposition =|> UserChannel;
        IsAbnormal(Disposition) => NOTHING;
        CH <-
          CHOPEN(CH.SourceID <- AssignTransactionID(CH) =|>
                      UserChannel, "Out", "Binary");
        EnQOutputOpenConn(CH, SP);
        Free(CH);
      END;

   EndOpenConnection <-
     EXPR(SP:ProcessHandle, CH:ConnHandle)
       BEGIN
         DECL UserChannel:UserData LIKE SP.UserChannel;
         IsAbnormal(CH.Disposition <- OpenDirectConnection(CH, SP)) =>
           EnQOutputCloseConn(CH, SP);
         < Disposition >(CH) =|> UserChannel;
         SIGNAL(CH.Signal);
         Free(CH);
       END;

  ^;
```

```
StartCloseConnection <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL C:ConnBlock;
      DECL CH:ChannelHandle;
      DECL Disposition:ReasonCode;
      < ConnID, DestProcess, Signal, Deadline >(C) <¦=
        UserChannel;
      ConvertTimer(Op, C.Deadline);
      Disposition <- ReasonCode <<
        BEGIN
          IsValidSignal(C.Signal) #> "Signal given is invalid";
          CH <- Allocate(ChannelHandle, C);
          "Incomplete";
        END;
      Disposition =¦> UserChannel;
      IsAbnormal(Disposition) => NOTHING;
      CH.UserChannel <-
        CHOPEN(CH.SourceID <- AssignTransactionID(CH) =¦>
                 UserChannel, "Out", "Binary");
      CH.SourceProcess <- SP.Name;
      EnQOutputCloseConn(CH, SP);
    END;

EndCloseConnection <-
  EXPR(SP:ProcessHandle, CH:ConnHandle)
    BEGIN
      CH.Disposition <- CloseDirectConnection(CH, SP);
      DECL UserChannel:UserData LIKE SP.UserChannel;
      < Disposition >(CH) =¦> UserChannel;
      FreeLocalSocket(CH.LocalSocket, SP);
      SIGNAL(CH.Signal);
      CHCLOSE(UserChannel);
      Free(CH);
    END;

StartTerminationSignal <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL T:TermBlock;
      DECL TH:TermHandle;
      DECL Disposition:ReasonCode;
      TH.Signal <¦= UserChannel;
      Disposition <- ReasonCode <<
        BEGIN
          IsValidSignal(S) #> "Signal given is invalid";
          TH <- Allocate(TermHandle, T);
          "Incomplete";
        END;
      Disposition =¦> UserChannel;
      IsAbnormal(Disposition) => NOTHING;
      TH.UserChannel <-
        CHOPEN(NullTransactionID, "Out", "Binary");
      SP.Termination <- TH;
      Release(TH);
    END;
```



```
EndTerminationSignal <-
  EXPR(SP:ProcessHandle, TH:TermHandle)
    BEGIN
      DECL UserChannel:ChannelHandle LIKE SP.UserChannel;
      < Disposition >(SP) =|> UserChannel;
      SIGNAL(TH.Signal);
      CHCLOSE(UserChannel);
      Free(TH);
      DoStopMe(SP);
    END;

EndBrokenConnection <-
  '
  .  An artifact.  The connection has been broken,
     so is closed.  The Op "StopMe" is posted
  .  and the CH delivered in such a way as to
  .  post the local proc     EndSignal.' */
    EXPR(Op:Opcode, SP:ProcessHandle, CH:ConnHandle)
      BEGIN
        DECL UserChannel:UserData LIKE SP.UserChannel;
        SIGNAL(CH.EndSignal);
        CHCLOSE(UserChannel);
        Free(CH);
      END;

^;
```

```
DoStopMe <-
  '
  .     Cancel all transactions and connections
  .     involving the user process.' */
    EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
      BEGIN
        DECL SP:ProcessHandle LIKE Seize(SP);
        StopTransactions(SP.OutputMessQ, SP);
        StopTransactions(SP.OutputAlarmS, SP);
        StopTransactions(SP.ConnectionS, SP);
        StopTransactions(SP.InputMessQ, SP);
        StopTransactions(SP.InputAlarmQ, SP);
        StopTransactions(SP.ReceiveMessQ, SP);
        StopTransactions(SP.ReceiveAlarmQ, SP);
        StopTransactions(SP.CloseConnS, SP);
        DECL TS:Set(TransactionHandle);
        Seize(GenericTable);
        FOREACH GH IN GenericTable
          DO
            FOREACH TH IN GH.ReceiveMessQ
              DO TH.DestProcess = SP.Name -> Insert(TS, TH) END;
          END;
        Release(GenericTable);
        StopTransactions(TQ, SP);
        Free(SP.Termination);
        "Normal" =¦> UserChannel;
        CHCLOSE(UserChannel);
        Free(SP);
      END;

StopTransactions <-
  EXPR(T:ANY, SP:ProcessHandle)
    FOREACH H IN T
      DO Seize(H); StopTransaction(H, SP); Free(H) END;

DoRescind <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL Event:TransactionID;
      Event <¦= UserChannel;
      DECL TH:TransactionHandle LIKE
        SeizeTransactionHandle(Event);
      RescindPendingEvent(TH, SP) =¦> TH.UserChannel;
      Free(TH);
    END;

DoAcceptAlarms <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    [] SP.IAccept <¦= UserChannel; "Normal" =¦> UserChannel [];
```

```
DoResynch <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    BEGIN
      DECL P:ProcessName;
      P <|= UserChannel;
      FOREACH C IN SP.InhibitedDestS
        DO C.DestProcess = P => Remove(C, SP.InhibitedDestS) END;
      "Normal" =|> UserChannel;
    END;

DoWhoAmI <-
  EXPR(Op:OpCode, UserChannel:ChannelHandle, SP:ProcessHandle)
    SP.Name =|> UserChannel;

^;
```

```
'       SERVERS          ' */ -;

UserCallServer <-
  EXPR(SH:ServerHandle)
    BEGIN
      DECL Op:OpCode;
      DECL WaitS:Set(ChannelHandle) LIKE SH.ChannelS;
      REPEAT
        DECL UserChannel:ChannelHandle LIKE AWAIT(WaitS);
        DECL SP:ProcessHandle LIKE UserChannel.User;
        Seize(SP);
        Op <|= UserChannel;
        OpTable[Op](Op, UserChannel, SP);
        Release(SP);
      END;
    END;

UserDeliveryServer <-
  '
  !     UserDeliveryServer waits until it is SIGNALed, then loops through
  !      its delivery queue, removes entries and dispatches to the
  !      appropriate output routine.  The entries are then dequeued.
  !
  ' */
    EXPR(SH:ServerHandle)
      BEGIN
        DECL QEntry:TransactionHandle;
        DECL LP:ProcessHandle;
        Seize(SH.DeliveryQ);
        REPEAT
          Null(SH.DeliveryQ) ->
            BEGIN
              SH.Running <- FALSE;
              Release(SH.DeliveryQ);
              WAIT({ SH.WakeUpSignal });
              Seize(SH.DeliveryQ);
            END;
          QEntry <- Seize(Front(SH.DeliveryQ));
          Release(SH.DeliveryQ);
          LP <- SeizeProcessHandle(Originator(QEntry));
          EndOpTable[QEntry.Op](LP, QEntry);
          Release(LP);
          Seize(SH.DeliveryQ);
          DeQ(SH.DeliveryQ, QEntry);
          Free(QEntry);
        END;
      END;
```

```
'      Miscellaneous routines  ' */ -;

StartGenericProcess <-
  EXPR(Class:GenericClassCode; ProcessHandle)
    BEGIN
      DECL GH:GenericHandle LIKE GenericTable[Class];
      DECL P:ProcessBlock;
      P.Name.GenericClass <- GH.Code;
      DECL PH:ProcessHandle LIKE Allocate(ProcessHandle, P);
      CompleteProcessName(PH);
      StartUserProcess(GH, PH);
      PH;
    END;

ConvertTimer <-
  EXPR(Op:OpCode, Timer:Integer)
    BEGIN
      Timer = 0 -> Timer <- TimerDefaults[Op];
      MakeIntervalAbsolute(Timer);
    END;

Originator <-
  EXPR(TH:TransactionHandle; ProcessName)
    CASE[TH.Op]
      ["ReceiveGenericMessage"],
        ["ReceiveSpecificMessage"],
        ["ReceiveAlarm"] => TH.DestProcess;
      TRUE => TH.SourceProcess;
    END;
```

```
!
!     QUEUE: Queue Management Module for MSG
!
! Management routines for the data base shared among MSG processes.
! This data base consists principally of queues and sets of
! transaction records. Each record describes the state of a pending
! event, a message, an alarm, or a direct connection.  Most queues
! and sets are associated with particular local user processes,
! and are anchored in the corresponding process handles.
!
! Routines exported by QUEUE are called from the servers that implement
! user process primitives (LOCAL), from those that implement the
! MSG-to-MSG protocol (REMOTE), and from the module CANCEL,
! which implements event timeouts.
!
! The exports fall into three groups: those dealing with messages,
! alarms, and direct connections.
!
! Message handling routines:
!
! EnQOutputMess          SendSpecificMess, SendGenericMess primitives
! EnQReceiveMess         ReceiveSpecificMess, ReceiveGenericMess primitives
! EnQInputMess           MESS protocol item
! RecordMESS\OK          MESS-OK protocol item
! RecordMESS\REJ         MESS-REJ protocol item
! RecordMESS\HOLD        MESS-HOLD protocol item
! RecordHOLD\OK          HOLD-OK protocol item
! RecordMESS\CANCEL      MESS-CANCEL protocol item
! RecordXMIT             XMIT protocol item
! SendMESS\REJ           Reject received MESS
! SendMESS\CANCEL        Cancel previously sent MESS
!
! Alarm handling routines:
!
! EnQOutputAlarm         SendAlarm primitive
! EnQReceiveAlarm        ReceiveAlarm primitive
! EnQInputAlarm          ALARM protocol item
! RecordALARM\OK         ALARM-OK protocol item
! RecordALARM\REJ        ALARM-REJ protocol item
! SendALARM\REJ          Reject received ALARM
!
! Connection handling routines:
!
! EnQOutputOpenConn      OpenConn primitive
! EnQOutputCloseConn     CloseConn primitive
! EnQInputOpenConn       CONN-OPEN protocol item
! EnQInputCloseConn      CONN-CLOSE protocol item
! RecordCONN\REJ         CONN-REJ protocol item
! SendCONN\REJ           Reject received CONN-OPEN or CONN-CLOSE
!
! */ -;

^;
```

```
!
! Conventions observed in this module:
!
! Local variables are declared by
!
!     <name> == <mode>
! or  <name> <== <initial value>
!
! These are equivalent to
!
!     DECL <name>:<mode>
! and DECL <name>:ANY BYVAL <initial value>
!
! respectively.
!
!
! Whenever a shareable record (e.g., a MessHandle or a ProcessHandle)
!  is passed to a QUEUE routine, it is assumed that the caller has
!  locked ("Seize"d) the record, if necessary, and will be responsible
!  for unlocking it.
!
! */ -;

^;
```

```
' - - - - - - - - - - - - - - - - - - - - -
!
!             Message Handling
!
! - - - - - - - - - - - - - - - - - - - - - - - -' */ -;

EnQOutputMess <-
  '
  ! M is an outgoing Mess from user process SP.
  ! Charge SP for the text buffer space used by M.
  ! Divide Messes into two categories:
  !  (1) those having specific destination hosts, and
  !  (2) generic messes that are host-less.
  ! If M is host-less, obtain a trial host and use EnQHostSpecificMess
  !  to try to deliver M. If M is rejected by the chosen host,
  !  RecordMESS\REJ will select a new host and use SendHostSpecificMess
  !  to send it again.
  !
  ' */
   EXPR(M:MessHandle, SP:ProcessHandle)
     BEGIN
       DebitBuffer(SP, M);
       M.IsGeneric AND Null(M.DestProcess.Host) ->
         BEGIN
           M.IsHostLess <- TRUE;
           M.DestProcess.Host <- GetNewGenericHost(M);
         END;
       StartTiming(M);
       EnQHostSpecificMess(M, SP);
     END;

GetNewGenericHost <-
  '
  ! Find a destination host for M appropriate to its generic category.
  ! If its destination host is null, take the first possible host,
  !  otherwise take the successor of the present host. Return a null
  !  host code if the present host is the last in the list.
  !
  ' */
   EXPR(M:MessHandle; HostCode)
     BEGIN
       OldHost <== M.DestProcess.Host;
       GH <== GenericTable[M.DestProcess.GenericName];
       Null(OldHost) => First(GH.HostS);
       Next(GH.HostS, OldHost);
     END;

^;
```

```
EnQHostSpecificMess <-
  '
  ! Check M for special handling requirements. If it is a sequenced or
  !  stream-marked message, check for prior Messes that inhibit
  !  transmission of M. A sequenced message must be held until all
  !  previous sequenced Messesto the same destination have been
  !  acknowledged. A stream-marked Mess must wait for all prior Messes
  !  to the same destination to complete.
  !
  ' */
    EXPR(M:MessHandle, SP:ProcessHandle)
      BEGIN
        PostponeTransmission <== FALSE;
        FOREACH PriorM IN SP.OutputMessQ
          DO
            PriorM.DestProcess = M.DestProcess AND
              (PriorM.IsMarked OR M.IsMarked OR
                M.IsSequenced AND PriorM.IsSequenced) =>
              PostponeTransmission <- TRUE;
          END;
        EnQ(SP.OutputMessQ, M);
        PostponeTransmission =>
          M.State <- "Awaiting prior MESS completion";
        SendHostSpecificMess(M, SP);
      END;

SendHostSpecificMess <-
  '
  ! If destination is a local process, pass a copy of M directly to the
  !  handler for received Messes.
  ! Otherwise, it must go out on the network. Decide, based on current
  !  buffer space for SP, whether to offer to hold the Mess, to warn
  !  that a hold will be impossible, or neither.  Deliver it to the server
  !  for the destination host.
  !
  ' */
    EXPR(M:MessHandle, SP:ProcessHandle)
      BEGIN
        M.DestProcess.Host = LocalHost =>
          BEGIN
            InputM <== CopyMessHandle(M);
            EnQInputMess(InputM, SP);
            Free(InputM);
          END;
        M.HoldOk <- SP.FreeBufferSize GE HoldOkThreshold;
        M.NoHold <- SP.FreeBufferSize LT NoHoldThreshold;
        M.State <- "Awaiting MESS-OK";
        SendMESS(M);
      END;

^;
```

```
EnQReceiveMess <-
   '
   ! DP is to be the destination of a message, for which it is issuing
   !  a ReceiveMess command.
   ! Input generic Messes and ReceiveGenericMess events are queued in the
   !  appropriate GenericTable entry. Therefore define DH to be either DP
   !  itself, if RM is specific, or the appropriate GenericHandle, if RM
   !  represents a ReceiveGenericMess.
   ! If there are any queued input Messes at DH that are complete with
   !  text (either accepted Messes or ones held by a local sender), take one,
   !  merge it with RM, and deliver it to DP.
   ! Otherwise, enqueue RM in DHs ReceiveMessQ, and start timing on RM,
   !  using the interval supplied by the user process DP.
   !
   ' */
     EXPR(RM:MessHandle, DP:ProcessHandle)
       BEGIN
         MR == MessHandle;
         DH <==
           BEGIN
             NOT RM.IsGeneric => DP;
             DECL GH:GenericHandle LIKE
               GenericTable[DP.Name.GenericName];
             REPEAT TestSeize(GH) => GH; Pause(SeizeWait) END;
           END;
         BEGIN
           FOREACH M AT MLocation FromFrontOf DP.InputMessQ
             DO
               NOT Null(M.Text) =>
                 [) MR <- M; Seize(MR); DeQAt(MLocation) (];
             END;
           NOT Null(MR) =>
             BEGIN
               StopTiming(MR);
               MR.SourceProcess.Host = LocalHost ->
                 LocalXMIT(MR, DH);
               MergeMessHandles(RM, MR);
               Free(MR);
               RM.Disposition <- "Normal";
               Deliver(RM, DP);
               CreditBuffer(DH, RM);
             END;
           EnQ(DH.ReceiveMessQ, RM);
           RM.State <- "Awaiting MESS";
           StartTiming(RM);
         END;
         RM.IsGeneric -> Release(DH);
       END;

^;
```

```
LocalXMIT <-
  '
  ! M represents a local Mess transaction. M has been held in the
  !  source process (SP) queue. Now it is to be "transmitted" to the
  !  destination (DH), simulating a network XMIT followed by a MESS
  !  retransmission.
  ! Debit DH and credit SP for the text buffer space.
  ! Complete the transaction for the sender by accepting the message.
  !
  ' */
    EXPR(M:MessHandle, DH:DestHandle)
      BEGIN
        SP <== SeizeProcessHandle(M.SourceProcess);
        DebitBuffer(DH, M);
        CreditBuffer(SP, M);
        AcceptMess(M, SP);
        Release(SP);
      END;

RequestTransmission <-
  '
  ! M represents a Mess to DH that is being held by its sender,
  !  but is now to be retransmitted.
  ! If the sender is remote, send it an XMIT item.
  ! Otherwise, use LocalXMIT
  !
  ' */
    EXPR(M:MessHandle, DH:DestHandle)
      BEGIN
        M.SourceProcess.Host = LocalHost => LocalXMIT(M, DH);
        SendXMIT(M);
        CommitBuffer(DH, M);
      END;

^;
```

```
EnQInputMess <-
  '
  ! The negation of M.QWait is ORed into M.NoHold in case M is generic.
  !  Thus M.NoHold indicates refusal to hold M at the source, for whatever
  !  reason.
  ! Input Messes are distinguished as "old" (those sent in response
  !  to an XMIT request) or "new". Old Messes are recognized by their
  !  non-null destination IDs.
  ! (SP represents the sender of M, if the sender is local.)
  !
  ' */
    EXPR(M:MessHandle, SP:ProcessHandle)
      BEGIN
        M.NoHold <- M.NoHold OR NOT M.QWait;
        NOT Null(M.DestID) => EnQOldInputMess(M);
        EnQNewInputMess(M, SP);
      END;

EnQNewInputMess <-
  '
  ! NewM is a newly received Mess.
  ! SP is the sender of NewM, if local. Otherwise, SP is null.
  ! Define DH to be the destination process handle (if NewM is specifically
  !  addressed) or the appropriate generic class handle (if NewM is generic).
  !  In the latter case, if there is no process with a pending Receive-
  !  GenericMess primitive, try to start a new process of the right class.
  ! Enqueue NewM in DH.InputMessQ if there is room.
  ! Assign a local transaction ID for future reference.
  ! Decide whether to deliver NewM, to accept it pending a local Receive,
  !  to ask the sender to hold it, or to reject it.
  !
  ' */
    EXPR(NewM:MessHandle, SP:ProcessHandle)
      BEGIN
        DH <== SeizeDestHandle(NewM.DestProcess);
        Null(DH) =>
          RejectMess(NewM,
                     BEGIN
                       NewM.IsGeneric =>
                         "That generic class not supported here";
                       "Destination process unknown";
                     END, SP);
        BEGIN
          Null(DH.ReceiveMessQ) AND NewM.IsGeneric ->
            StartGenericProcess(DH.Code);
          IsFullQ(DH.InputMessQ) =>
            RejectMess(NewM,
                       "Insufficient resources to complete command",
                       SP);
          EnQ(DH.InputMessQ, NewM);
          NewM.DestID <- AssignTransactionID(NewM);
          AcceptHoldOrRejectMess(NewM, DH, SP);
        END;
        Release(DH);
```

```
    END;

^;
```

```
EnQOldInputMess <-
  '
  ! NewM has been sent in response to an XMIT.
  ! Use NewMs destination ID to find the original MessHandle (OldM).
  ! If none can be found, or if OldM is the wrong transaction, simply
  !  treat NewM as a new incoming Mess. (The original could have timed out.)
  ! Otherwise, merge NewM into OldM, copying the Text, HoldOk, and
  !  NoHold fields, among others.
  ! Let DH be the destination of OldM.
  ! Decide how to dispose of OldM using the same algorithm as for
  !  new input Messes.
  !
  ' */
    EXPR(NewM:MessHandle)
      BEGIN
        OldM <== SeizeTransaction(NewM.DestID);
        BEGIN
          Null(OldM) OR NOT IsMessHandle(OldM) OR
            NOT ValidXMITResponse(NewM, OldM) =>
            EnQNewInputMess(NewM);
          StopTiming(OldM);
          MergeMessHandles(OldM, NewM);
          DH <= SeizeDestHandle(OldM.DestProcess);
          RemitBuffer(DH, M);
          AcceptHoldOrRejectMess(OldM, DH);
          Release(DH);
        END;
        Release(OldM);
      END;

ValidXMITResponse <-
  EXPR(NewM:MessHandle, OldM:MessHandle; BOOL)
    OldM.State = "Awaiting retransmission" AND
      NewM.SourceProcess = OldM.SourceProcess AND
      NewM.DestProcess = OldM.DestProcess AND
      NewM.SourceID = OldM.SourceID;

^;
```

```
AcceptHoldOrRejectMess <-
  !
  ! M is an input Mess for destination DH, which is either a GenericHandle
  !  (if M is generic) or a ProcessHandle (if M is specific).
  ! SP is the sending process, if local to this host; otherwise null.
  ! M has been entered in DH.InputMessQ.
  ! Possible outcomes:
  !  M will be accepted and delivered.
  !  M will be accepted and queued.
  !  The sender will be asked to buffer M and retransmit later.
  !  M will be rejected.
  ! If the Text of M has been omitted for lack of sufficient temporary
  !  buffer space, then either request hold or reject.
  ! Else if a matching ReceiveMess primitive is pending, accept and
  !  deliver M.
  ! Else if buffer space permits, accept M and leave it queue pending
  !  the execution of a local ReceiveMess primitive.
  ! Otherwise, either reject M or request that the sender hold it.
  !
  ! */
    EXPR(M:MessHandle, DH:DestHandle, SP:ProcessHandle)
      BEGIN
        Null(M.Text) => HoldOrRejectMess(M, DH, SP);
        RM <== SeizeMatchingReceiveMess(M, DH);
        NOT Null(RM) =>
          BEGIN
            StopTiming(RM);
            DeQ(DH.InputMessQ, M);
            MergeMessHandles(RM, M);
            AcceptMess(M, SP);
            BEGIN
              NOT RM.IsGeneric => Deliver(RM, DH);
              DP <== SeizeProcessHandle(RM.DestProcess);
              Deliver(M, DP);
              Release(DP);
            END;
            Release(RM);
          END;
        NOT RoomToAccept(M, DH) => HoldOrRejectMess(M, DH, SP);
        AcceptMess(M, SP);
        DebitBuffer(DH, M);
        M.State <- "Awaiting ReceiveMess";
        StartTiming(M, AwaitingReceiveInterval);
      END;

^;
```

```
HoldOrRejectMess <-
'
! M is an input Mess for destination DH, which is either a GenericHandle
!  (if M is generic) or a ProcessHandle (if M is specific).
! SP is the sending process, if local to this host; otherwise null.
! M has been entered in DH.InputMessQ.
! If M requires an immediate acceptance decision, reject it.
! If M represents a strictly local transfer, make the decision on holding
!  here: hold if the sender has enough buffer space, otherwise abort the
!  transfer.
!  (Note that in computing SourceM, the original output MessHandle,
!   we have no need to Seize the record: the calling path will already
!   have done so.)
! Otherwise, send a MESS-HOLD to the sender of M.
!
' */
  EXPR(M:MessHandle, DH:DestHandle, SP:ProcessHandle)
    BEGIN
      M.NoHold =>
        BEGIN
          DeQ(DH.InputMessQ, M);
          RejectMess(M,
                     "Insufficient resources to complete command",
                     SP);
        END;
      NOT Null(SP) =>
        BEGIN
          NOT RoomToHold(SP, M) =>
            BEGIN
              DeQ(DH.InputMessQ, M);
              RejectMess(M,
                         "Insufficient resources to complete command",
                         SP);
            END;
          SourceM <== LookupTransaction(M.SourceID);
          SourceM.State <- "Awaiting XMIT";
          M.State <- "Held by sender";
          StartTiming(M, AwaitingFreeBufferInterval);
        END;
      M.Text <- NullText;
      SendMESS\HOLD(M);
      BEGIN
        M.State <- "Held by sender";
        StartTiming(M, AwaitingFreeBufferInterval);
      END;
      M.State <- "Awaiting HOLD-OK";
      StartTiming(M, AwaitingHOLD\OKInterval);
  END;

^;
```

```
SeizeMatchingReceiveMess <-
  '
  ! M is an incoming Mess for DH.
  ! If DH has a pending Receive that will match M,
  !  seize, dequeue, and return it.
  ! Else, return nil.
  !
  ' */
    EXPR(M:MessHandle, DH:DestHandle; MessHandle)
      BEGIN
        Null(DH.ReceiveMessQ) => NullMessHandle;
        RM <== Front(DH.ReceiveMessQ);
        Seize(RM);
        DeQ(DH.ReceiveMessQ);
        RM;
      END;

SeizeDestHandle <-
  '
  ! Return a suitable destination handle for the given process name P.
  ! If P is a generic name, and its class is supported on the local host,
  !  return the corresponding generic handle.
  ! If P is a valid specific process, return its handle.
  ! Otherwise, return a null handle.
  !
  ' */
    EXPR(P:ProcessName; DestHandle)
      BEGIN
|       NOT Null(P.Instance) => SeizeProcessHandle(P);
|       DH <== GenericTable[P.Name.GenericName];
|       REPEAT TestSeize(DH) => NOTHING; Pause(SeizeWait) END;
        NOT DH.Unsupported => DH;
        Release(DH);
        NullDestHandle;
      END;

MergeMessHandles <-
  '
  ! M represents a newly received Mess.
  ! RM is either a ReceiveMess record or the result of an earlier
  !  transmission of M.
  ! Copy the Text of M into RM, as well as other fields which might
  !  be different on retransmission.
  !
  ' */
    EXPR(RM:MessHandle, M:MessHandle)
      BEGIN
        RM.NoHold <- M.NoHold;
        RM.HoldOk <- M.HoldOk;
        RM.SourceProcess <- M.SourceProcess;
        RM.TextLength <- M.TextLength;
        RM.Text <- M.Text;
      END;

^;
```

```
AcceptMess <-
  '
  ! If SP is non-null, then M represents an intrahost Mess and SP is
  !  the source process. In that case, simulate a MESS-OK acknowledgement
  !  of M for SP. (It is unnecessary to seize the original
  !  MessHandle. The calling path has it locked.)
  ! Otherwise, send a MESS-OK to the remote source of M.
  !
  ' */
    EXPR(M:MessHandle, SP:ProcessHandle)
      BEGIN
        Null(SP) => SendMESS\OK(M);
        AcceptOutputMess(LookupTransaction(M.SourceID), SP);
      END;

RejectMess <-
  '
  ! If SP is non-null, then M represents an intrahost Mess and SP is
  !  the source process. In that case, simulate a MESS-REJ acknowledgement
  !  of M for SP. (It is unnecessary to seize the original
  !  MessHandle. The calling path has it locked.)
  ! Otherwise, send a MESS-REJ to the remote source of M.
  !
  ' */
    EXPR(M:MessHandle, Reason:ReasonCode, SP:ProcessHandle)
      BEGIN
        Null(SP) => SendMESS\REJ(M, Reason);
        RejectOutputMess(LookupTransaction(M.SourceID), Reason,
                         SP);
      END;

CancelMess <-
  '
  ! M is an outgoing Mess from SP that is awaiting XMIT, but must be
  !  canceled for lack of buffer space.
  ! If the destination is local, simulate receipt of a MESS-CANCEL by
  !  the destination. Otherwise, send a real MESS-CANCEL on the network.
  ! Then abort the SendMess primitive of SP.
  !
  ' */
    EXPR(M:MessHandle, SP:ProcessHandle)
      BEGIN
        M.Reason <- "Insufficient resources to complete command";
        BEGIN
          M.DestProcess.Host = LocalHost =>
            RecordMESS\CANCEL(Value(M));
          SendMESS\CANCEL(M, M.Reason);
        END;
        RejectOutputMess(M, M.Reason, SP);
      END;

^;
```

```
RecordMESS\OK <-
  !
  ! A MESS-OK has come in for MOK.SourceProcess. If no corresponding
  !  transaction can be found, simply do nothing.  Otherwise, complete
  !  the designated pending event.
  !
  ! */
    EXPR(MOK:MessBlock)
      BEGIN
        OldM <== SeizeTransaction(MOK.SourceID);
        Null(OldM) OR MessMismatch(OldM, MOK) =>
          /* 'Could have timed out';
        OldM.State # "Awaiting MESS-OK" =>
          MSGError('MESS-OK received in wrong state');
        SP <= SeizeProcessHandle(MOK.SourceProcess);
        AcceptOutputMess(OldM, SP);
        Release(OldM);
        Release(SP);
      END;

RecordMESS\REJ <-
  !
  ! A MESS-REJ has come in for MR.SourceProcess. If no corresponding
  !  transaction can be found, simply do nothing.  Otherwise, abort
  !  the designated pending event, giving MR.Reason as the error code.
  !
  ! */
    EXPR(MR:MessBlock)
      BEGIN
        OldM <== SeizeTransaction(MR.SourceID);
        Null(OldM) OR MessMismatch(OldM, MR) =>
          /* 'Could have timed out';
        SP <= SeizeProcessHandle(MR.SourceProcess);
        RejectOutputMess(OldM, MR.Reason, SP);
        Release(OldM);
        Release(SP);
      END;

^;
```

```
AcceptOutputMess <-
   !
   ! M, a Mess sent by SP, has been accepted at its destination.
   !  Complete M, and transmit any subsequent Messes from SP to M.DestProcess
   !  that may have been blocked pending acknowledgement of M.
   ! Scan SPs queue of output Messes from least recent to most recent.
   !  Ignore Messes to other destinations.
   !  When M is found, delete it from the  queue, but continue to scan
   !   for later Messes, QM, which are waiting to be sent.
   !  If QM is found, and is marked, terminate the search, and send it if
   !   removal of M has unblocked it. Any later Messes remain blocked by QM.
   !  If QM exists and is sequenced, its treatment depends on M.Handling:
   !   If M.IsSequenced, QM must now be unblocked. Send QM and stop scanning.
   !   If M.IsMarked, send QM if it is now unblocked, but continue to scan
   !    for newly unblocked Messes.
   !   Otherwise, simply terminate the scan: removal of M has not unblocked
   !    QM, and any later Messes that might have been blocked by M are
   !    also blocked by QM.
   ! After the scan, copy the destination process name into M in case it
   !  was a generic Mess, deliver M to SP, and credit SP with the space
   !  being freed.
   !
   ' */
     EXPR(M:MessHandle, SP:ProcessHandle)
       BEGIN
         MFound <== FALSE;
         PriorMesses <== FALSE;
         PriorSequencedMesses <== FALSE;
         TerminateScan <== FALSE;
         FOREACH QM AT QMLocation FromFrontOf SP.OutputMessQ
           DO
             MatchingProcessNames(QM.DestProcess, M.DestProcess) ->
               BEGIN
                 NOT MFound =>
                   BEGIN
                     QM # M => PriorMesses <- TRUE;
                     MFound <- TRUE;
                     StopTiming(M);
                     DeQAt(QMLocation);
                   END;
                 QM.State # "Awaiting prior MESS completion" =>
                   PriorMesses <- TRUE;
                 QM.IsMarked =>
                   BEGIN
                     NOT PriorMesses ->
                       SendHostSpecificMess(QM, SP);
                     TerminateScan <- TRUE;
                   END;
                 ^;
```

```
                QM.IsSequenced =>
                  BEGIN
                    M.IsSequenced =>
                      BEGIN
                        SendHostSpecificMess(QM);
                        TerminateScan <- TRUE;
                      END;
                    NOT M.IsMarked => TerminateScan <- TRUE;
                    PriorSequencedMesses =>
                      /* 'QM remains blocked';
                    SendHostSpecificMess(QM);
                    PriorMesses <- PriorSequencedMesses <- TRUE;
                  END;
                ASSERT(M.IsMarked)      /* ' QM can't be blocked by a marked Mess
                                           ! between QM and M or else the scan
                                           ! would have terminated by now. ';
                SendHostSpecificMess(QM, SP);
                PriorMesses <- TRUE;
              END;
            TerminateScan =>
              /* 'Later blocked messes remain blocked';
          END;
        M.Disposition <- "Normal";
        M.DestProcess <- DestProcess;
        Deliver(M, SP);
        CreditBuffer(SP, M);
      END;

^;
```

```
RejectOutputMess <-
  '
  ! Purpose:
  ! A Mess M from SP has been rejected by M.DestProcess. If M was originally
  !  a generic Mess without a specified destination host, try to find an
  !  alternate host for it. If successful, send it out again. Otherwise,
  !  abort M, and either abort or transmit subsequent Messes from SP
  !  to M.DestProcess that have been blocked pending acknowledgement of M.
  ! Method:
  ! Scan SPs queue of output Messes from least recent to most recent.
  !  Ignore Messes to other destinations.
  !  When the Mess M corresponding to SourceID is found, either (1) send it to
  !   a new host (if it was originally hostless), or (2) abort M,
  !   but continue to scan for later Messes, QM, which are blocked
  !   pending completion of earlier ones.
  !  If such a QM is found, treat it as follows:
  !   If M.IsMarked, all subsequent Messes are to be aborted, so abort QM.
  !   Else if QM.IsMarked, it may now be unblocked. If so, send it.
  !   Else if both M and QM are sequenced, abort QM; all sequenced Messes
  !    subsequent to M will be aborted until the user process Resynchs.
  !   Otherwise, M was not blocking QM, so it remains blocked.
  !  If M has no special handling, the scan can be terminated after the first
  !   such QM has been seen.
  !  NoDeliver will be TRUE iff SP is terminating; dont deliver anything to it.
  !
  ' */
    EXPR(M:MessHandle,
         Reason:ReasonCode,
         SP:ProcessHandle,
         NoDeliver:BOOL)
      BEGIN
        MFound <== FALSE;
        PriorMesses <== FALSE;
        TerminateScan <== FALSE;
        FOREACH QM AT QMLocation FromFrontOf SP.OutputMessQ
          DO
            MatchingProcessNames(QM.DestProcess, DestProcess) ->
              BEGIN
                NOT MFound =>
                  BEGIN
                    QM # M => PriorMesses <- TRUE;
                    MFound <- TRUE;
                    NOT M.IsHostLess =>
                      AbortMess(M, QMLocation, SP, Reason);
                    M.DestProcess.Host <- GetNewGenericHost(M);
                    Null(M.DestProcess.Host) =>
                      AbortMess(M, QMLocation, SP,
                                "Refused by all suitable hosts");
                    SendHostSpecificMess(M, SP);
                    TerminateScan <- TRUE;
                  END;
                ^;
```

```
              QM.State # "Awaiting prior MESS completion" =>
                PriorMesses <- TRUE;
              M.IsMarked =>
                AbortMess(QM, QMLocation, SP,
                          "Failure of prior stream marked message");
              TerminateScan <- M.Handling = "Normal";
              QM.IsMarked =>
                BEGIN
                  NOT PriorMesses ->
                    SendHostSpecificMess(QM, SP);
                  PriorMesses <- TRUE;
                END;
              NOT (QM.IsSequenced AND M.IsSequenced) =>
                ASSERT(PriorMesses)   /* 'M wasnt blocking QM';
              AbortMess(QM, QMLocation, SP,
                        "Failure of prior sequenced message");
            END;
          TerminateScan =>
            /* 'Later blocked messes remain blocked';
        END;
    END;

  AbortMess <-
    EXPR(M:MessHandle,
         QP:QueueEntryPointer,
         SP:ProcessHandle,
         Reason:ReasonCode,
         NoDeliver:BOOL)
      BEGIN
        StopTiming(M);
        DeQAt(QP);
        M.Disposition <- Reason;
        CreditBuffer(SP, M);
        M.Text <- NullText;
        NOT NoDeliver -> Deliver(M, SP);
      END;

   ^;
```

```
RecordMESS\HOLD <-
  !
  ! A MESS-HOLD item has arrived.
  ! Look for the transaction, OldM, designated. If it cant be found, send a
  !  MESS-CANCEL. OldM may have timed out or been Rescinded.
  ! Check for correct transaction state.
  ! See whether there is room to hold OldM. If not, send MESS-CANCEL.
  !  If so, send HOLD\OK, unless that was implicit in the original MESS
  !  transmission.
  !
  ! */
    EXPR(MH:MessBlock)
      BEGIN
        OldM <== SeizeTransaction(MH.SourceID);
        Null(OldM) =>
          AnswerBelatedItem(MH, "MESS-CANCEL",
                                  "Destination process unknown");
        BEGIN
          IsMessHandle(OldM) AND Null(OldM.DestID) ->
            OldM.DestID <- MH.DestID;
          MessMismatch(OldM, MH) =>
            MSGError('Transaction mismatch');
          OldM.State # "Awaiting MESS-OK" =>
            MSGError('MESS-HOLD received in wrong state');
          SP <== SeizeProcessHandle(OldM.SourceProcess);
          BEGIN
            SP.FreeBufferSize GE MinMessHoldThreshold =>
              BEGIN
                OldM.State <- "Awaiting XMIT";
                NOT OldM.HoldOk -> SendHOLD\OK(OldM);
              END;
            RejectOutputMess(SP, OldM.SourceID, OldM.DestProcess,
                                  "Insufficient resources to complete command");
            SendMESS\CANCEL(OldM,
                                  "Insufficient resources to complete command");
          END;
          Release(SP);
        END;
        Release(OldM);
      END;

^;
```

```
RecordHOLD\OK <-
  '
  ! A HOLD\OK item has arrived.
  ! If the designated transaction has timed out, invite the sender to
  !  retransmit (XMIT).
  ! Otherwise, be sure that HOLD\OK is expected.
  ! Let DH denote the destination, whether generic or specific.
  ! If DH now has room for the Mess, request retransmission by sending XMIT.
  ! Otherwise, leave the transaction in the "Held by sender" state, waiting
  !  for buffer space to become available.
  !
  ' */
    EXPR(HOK:MessBlock)
      BEGIN
        OldM <== SeizeTransaction(HOK.DestID);
        Null(OldM) => AnswerBelatedItem(HOK, "XMIT");
        BEGIN
          MessMismatch(OldM, HOK) =>
            MSGError('Transaction mismatch');
          OldM.State # "Awaiting HOLD-OK" =>
            MSGError('HOLD-OK in wrong state');
          StopTiming(OldM);
          DH <== SeizeDestHandle(OldM.DestProcess);
          BEGIN
            DH.FreeBufferSize GE XMITThreshold =>
              BEGIN
                OldM.State <- "Awaiting retransmission";
                StartTiming(OldM, AwaitingRetransmissionInterval);
                SendXMIT(OldM);
              END;
            OldM.State <- "Held by sender";
            StartTiming(OldM, AwaitingFreeBufferInterval);
          END;
          Release(DH);
        END;
        Release(OldM);
      END;

^;
```

```
RecordMESS\CANCEL <-
  '
  ! A MESS\CANCEL item has arrived.
  ! If the transaction it designates does not exist, just ignore it.
  ! Check the validity of the transaction state, then delete its record from
  !  the appropriate InputMessQ.
  !
  ' */
    EXPR(MC:MessBlock)
      BEGIN
        OldM <= SeizeTransaction(MC.DestID);
        Null(OldM) => /* 'Ignore belated MESS-CANCEL';
        BEGIN
          MessMismatch(OldM, MC) =>
            MSGError('Transaction mismatch');
          OldM.State # "Awaiting HOLD-OK" AND
            OldM.State # "Held by sender" AND
            OldM.State # "Awaiting retransmission" =>
            MSGError('MESS-CANCEL received in wrong state');
          StopTiming(OldM);
          DH <== SeizeDestHandle(OldM.DestProcess);
          DeQ(DH.InputMessQ, OldM);
          Free(OldM);
          Release(DH);
        END;
        Release(OldM);
      END;

^;
```

```
RecordXMIT <-
  '
  ! An XMIT request has arrived.
  ! If the designated transaction has timed out, MESS-CANCEL will already
  !  have been sent, so ignore the XMIT.
  ! Otherwise, after checking the transaction state, retransmit the held Mess.
  !
  ' */
    EXPR(XM:MessBlock) .
      BEGIN
        OldM <== SeizeTransaction(XM.SourceID);
        Null(OldM) => /* 'Ignore belated XMIT';
        BEGIN
          MessMismatch(OldM, XM) =>
            MSGError('Transaction mismatch');
          OldM.State # "Awaiting XMIT" =>
            MSGError('XMIT received in wrong state');
          OldM.State <- "Awaiting MESS-OK";
          SendMESS(OldM);
        END;
        Release(OldM);
      END;

AnswerBelatedItem <-
  EXPR(MB:MessBlock, Command:ProtocolCode, Reason:ReasonCode)
    BEGIN
      M <== Allocate(MessHandle, M);
      M.ProtocolCommand <- Command;
      M.Reason <- Reason;
      DeliverToRemoteHost(M,
                          BEGIN
                            Command = "MESS-CANCEL" =>
                              M.DestProcess.Host;
                            M.SourceProcess.Host;
                          END);
      Free(M);
    END;

MessMismatch <-
  '
  ! OldM is an existing transaction; NewM is a new protocol item.
  ! Return TRUE iff NewM refers to the same transaction as OldM.
  !
  ' */
    EXPR(OldM:MessHandle, NewM:MessBlock; BOOL)
      OldM.SourceID # NewM.SourceID OR OldM.DestID # NewM.DestID OR
        OldM.SourceProcess # NewM.SourceProcess OR
        OldM.DestID # NewM.DestID;

^;
```

```
DebitBuffer <-
  '
  ! Charge the space for M.Text to its destination D, which is either
  !  a specific process or a generic category.
  ! If D is a specific process and free space has dropped below the
  !  minimum desirable level, cancel held Messes until enough buffer
  !  space is reclaimed.
  !
  ' */
    EXPR(D:DestHandle, M:MessHandle)
      BEGIN
        D.FreeBufferSize <- D.FreeBufferSize - M.TextLength;
        IsProcessHandle(D) ->
          FOREACH OldM IN D.OutputMessQ
            DO
              D.FreeBufferSize GE MinMessHoldThreshold =>
                /* 'No need to cancel further held Messes';
              OldM.State = "Awaiting XMIT" ->
                CancelMess(OldM, D);
            END;
      END;

CreditBuffer <-
  '
  ! Credit the space for M Text to its destination D, which is either
  !  a specific process or a generic category.
  ! If free space now permits buffering additional Messes,
  !  request transmission of any being held by their senders
  !  until VirtualFreeSpace drops below a desirable minimum.
  !
  ' */
    EXPR(D:DestHandle, M:MessHandle)
      BEGIN
        D.FreeBufferSize <- D.FreeBufferSize + M.TextLength;
        FOREACH OldM IN D.OutputMessQ
          DO
            VirtualFreeSpace(D) LT RequestXMITThreshold =>
              /* 'No room to request further transmissions';
            OldM.State = "Held by sender" ->
              RequestTransmission(OldM, D);
          END;
      END;

^;
```

```
VirtualFreeSpace <-
  EXPR(D:DestHandle; Integer)
    D.FreeBufferSize - D.CommittedBufferSpace;

CommitBuffer <-
  '
  ! Try to hold some of Ds buffer space for a Mess M known to be coming.
  !
  ' */
    EXPR(D:DestHandle, M:MessHandle)
      D.CommittedBufferSpace <-
        D.CommittedBufferSpace + M.TextLength;

RemitBuffer <-
  '
  ! Inverse of CommitBuffer.
  !
  ' */
    EXPR(D:DestHandle, M:MessHandle)
      D.CommittedBufferSpace <-
        D.CommittedBufferSpace - M.TextLength;

^;
```

```
' - - -
!
! Send message-related protocol items to a remote MSG instance.
!
' */ -;

SendMESS <-
  EXPR(M:MessHandle)
    BEGIN
      M.ProtocolCommand <- "MESS";
      DeliverToRemoteHost(M, M.DestProcess.Host);
    END;

SendMESS\OK <-
  EXPR(M:MessHandle)
    BEGIN
      M.ProtocolCommand <- "MESS-OK";
      DeliverToRemoteHost(M, M.SourceProcess.Host);
    END;

SendMESS\REJ <-
  EXPR(M:MessHandle, Reason:ReasonCode)
    BEGIN
      M.ProtocolCommand <- "MESS-REJ";
      M.Reason <- Reason;
      DeliverToRemoteHost(M, M.SourceProcess.Host);
    END;

SendMESS\CANCEL <-
  EXPR(M:MessHandle, Reason:ReasonCode)
    BEGIN
      M.ProtocolCommand <- "MESS-CANCEL";
      M.Reason <- Reason;
      DeliverToRemoteHost(M, M.DestProcess.Host);
    END;

SendMESS\HOLD <-
  EXPR(M:MessHandle)
    BEGIN
      M.ProtocolCommand <- "MESS-HOLD";
      DeliverToRemoteHost(M, M.SourceProcess.Host);
    END;
```

```
SendXMIT <-
  EXPR(M:MessHandle)
    BEGIN
      M.ProtocolCommand <- "XMIT";
      DeliverToRemoteHost(M, M.SourceProcess.Host);
    END;

^;
```

```
' - - - - - - - - - - - - - - - - - - -
!
!     Alarm Handling
!
! - - - - - - - - - - - - - - - - - - - ' */ -;

EnQOutputAlarm <-
  '
  ! A represents an alarm sent by process SP.
  ! If the destination process is local, make a record for the
  !  input and call EnQInputAlarm directly.
  !
  ' */
    EXPR(A:AlarmHandle, SP:ProcessHandle)
      BEGIN
        A.State <- "Awaiting ALARM-OK";
        Insert(SP.OutputAlarmS, A);
        StartTiming(A);
        A.DestProcess.Host # LocalHost => SendALARM(A);
        InputA <== CopyAlarmHandle(A);
        EnQInputAlarm(InputA, SP);
        Free(InputA);
      END;

EnQReceiveAlarm <-
  '
  ! RA represents a ReceiveAlarm (called "Enable alarm" in the MSG Design
  !  Specification) primitive issued by process DP.
  ! If no matching ALARM item has yet arrived, enqueue RA and set its
  !  timer.
  ! If ALARMs have been accepted, take the first, AR, out of the queue
  !  and merge it with RA. Complete the ReceiveAlarm successfully.
  !
  ' */
    EXPR(RA:AlarmHandle, DP:ProcessHandle)
      BEGIN
        Null(DP.InputAlarmQ) =>
          BEGIN
            EnQ(DP.ReceiveAlarmQ);
            RA.State <- "Awaiting ALARM";
            StartTiming(RA);
          END;
        AR <== Front(DP.InputAlarmQ);
        Seize(AR);
        StopTiming(AR);
        DeQ(DP.InputAlarmQ, AR);
        MergeAlarmHandles(RA, AR);
        Free(AR);
        Deliver(RA, DP);
      END;

 ^;
```

```
EnQInputAlarm <-
  '
  ! A represents an input ALARM item.
  ! SP represents the source process, if local. Otherwise, it is null.
  ! Reject A if its destination process, DP, cannot be found or if it is
  !  not accepting alarms.
  ! If DP has issued a ReceiveAlarm, RA, that is pending, then accept A
  !  and merge it with RA to complete the Receive.
  ! If there is no pending Receive, but DPs input alarm queue has room,
  !  accept A and queue it.
  ! Otherwise, reject A because of the full input queue.
  !
  ' */
    EXPR(A:AlarmHandle, SP:ProcessHandle)
      BEGIN
        DP <== SeizeProcessHandle(A.DestProcess);
        Null(DP) =>
          RejectAlarm(A, "Destination process unknown", SP);
        BEGIN
          NOT DP.IAccept =>
            RejectAlarm(A, "Process not accepting alarms now",
                        SP);
          Null(DP.ReceiveAlarmQ) =>
            BEGIN
              IsFullQ(DP.InputAlarmQ) =>
                RejectAlarm(A, "Alarm queue for process is full",
                            SP);
              AcceptAlarm(A, SP);
              A.State <- "Awaiting ReceiveAlarm";
              EnQ(DP.InputAlarmQ, A);
              StartTiming(A, AwaitingReceiveInterval);
            END;
          AcceptAlarm(A, SP);
          RA <== Front(DP.ReceiveAlarmQ);
          Seize(RA);
          StopTiming(RA);
          DeQ(DP.ReceiveAlarmQ, RA);
          MergeAlarmHandles(RA, A);
          Deliver(RA, DP);
        END;
        Release(DP);
      END;

^;
```

```
MergeAlarmHandles <-
  '
  ! RA represents a ReceiveAlarm primitive into which the information
  !  in A, an input alarm, must be merged, so that RA can be delivered.
  !
  ' */
    EXPR(RA:AlarmHandle, A:AlarmHandle)
      BEGIN
        RA.Alarm <- A.Alarm;
        RA.SourceProcess <- A.SourceProcess;
        RA.Disposition <- "Normal";
      END;

RecordALARM\OK <-
  '
  ! AOK represents an ALARM-OK item.
  ! Find and complete the corresponding SendAlarm event.
  !
  ' */
    EXPR(AOK:AlarmBlock)
      BEGIN
        A <== SeizeTransaction(AOK.SourceID);
        Null(A) OR NOT IsAlarmHandle(A) OR
          A.DestProcess # AOK.DestProcess =>
          /* 'SendAlarm may have timed out';
        SP <== SeizeProcessHandle(A.SourceProcess);
        Remove(SP.OutputAlarmS, A);
        A.Disposition <- "Normal";
        Deliver(A, SP);
        Release(SP);
        REL(A);
      END;

RecordALARM\REJ <-
  '
  ! AR represents an ALARM-REJ item.
  ! Find and abort the corresponding SendAlarm event.
  !
  ' */
    EXPR(AR:AlarmBlock)
      BEGIN
        A <== SeizeTransaction(AR.SourceID);
        Null(A) OR NOT IsAlarmHandle(A) OR
          A.DestProcess # AR.DestProcess =>
          /* 'SendAlarm may have timed out';
        SP <== SeizeProcessHandle(A.SourceProcess);
        Remove(SP.OutputAlarmS, A);
        A.Disposition <- AR.Reason;
        Deliver(A, SP);
        Release(SP);
        Release(A);
      END;

^;
```



```
AcceptAlarm <-
  '
  ! A represents an input Alarm to be accepted.
  ! SP is the source process, if local; otherwise it is null.
  ! If the sender is remote, send an ALARM-OK.
  ! If local, find and complete the original SendAlarm event.
  !  It is unnecessary to seize SourceA: the calling path
  !  has it already.
  !
  ' */
    EXPR(A:AlarmHandle, SP:ProcessHandle)
      BEGIN
        Null(SP) => SendALARM\OK(A);
        SourceA <== LookupTransaction(A.SourceID);
        DeQ(SP.OutputAlarmS, SourceA);
        SourceA.Disposition <- "Normal";
        Deliver(SourceA, SP);
      END;

RejectAlarm <-
  '
  ! A represents an input Alarm to be rejected for the Reason given.
  ! SP is the source process, if local; otherwise it is null.
  ! If the sender is remote, send an ALARM-REJ.
  ! If local, find and abort the original SendAlarm event.
  !  It is unnecessary to seize SourceA: the calling path
  !  has it already.
  !
  ' */
    EXPR(A:AlarmHandle, SP:ProcessHandle)
      BEGIN
        Null(SP) => SendALARM\REJ(A, Reason);
        SourceA <== LookupTransaction(A.SourceID);
        DeQ(SP.OutputAlarmS, SourceA);
        SourceA.Disposition <- Reason;
        Deliver(SourceA, SP);
      END;

^;
```

```
! - - -
!
! Transmit alarm-related protocol items to a remote MSG.
!
! */ -;

SendALARM <-
  EXPR(A:AlarmHandle)
    BEGIN
      A.ProtocolCommand <- "ALARM";
      DeliverToRemoteHost(A, A.DestProcess.Host);
    END;

SendALARM\OK <-
  EXPR(A:AlarmHandle)
    BEGIN
      A.ProtocolCommand <- "ALARM-OK";
      DeliverToRemoteHost(A, A.DestProcess.Host);
    END;

SendALARM\REJ <-
  EXPR(A:AlarmHandle, Reason:ReasonCode)
    BEGIN
      A.ProtocolCommand <- "ALARM-REJ";
      A.Reason <- Reason;
      DeliverToRemoteHost(A, A.DestProcess.Host);
    END;

^;
```

```
! - - - - - - - - - - - - - - - - - - -
!
!      Handle Direct Connections
!
! - - - - - - - - - - - - - - - - - - - - ' */ -;

EnQOutputOpenConn <-
  '
  ! Process SP has issued an OpenConn, represented by OutC.
  ! OutC.ConnBytesize and OutC.ConnType have been validated.
  ! Scan its connection set for one with the same ConnID as OutC,
  !  and reject OutC if another one exists.
  ! If no request for a connection with name OutC.ConnID has yet been received,
  !  then transmit a CONN-OPEN to the remote process (or simulate same if
  !  that process is local), and put OutC in the set of outgoing connection
  !  requests for SP.
  ! If there is a corresponding incoming request, call it InC. If the remote
  !  process names of InC and OutC do not match, then reject OutC.
  ! Check that InC and OutC have agreeable types and byte sizes. If not,
  !  abort the local pending open, which will cause CONN-CLOSE to be sent
  !  to the remote process.
  ! If the connection types agree, complete the OpenConn primitive
  !  successfully, merge the input ConnHandle into OutC, discard InC, and
  !  enter the merged ConnHandle in SPs set of open connections.
  !
  ' */
  EXPR(OutC:ConnHandle, SP:ProcessHandle)
    BEGIN
      InC == ConnHandle;
      FOREACH C IN SP.ConnectionS
        DO C.ConnID = InC.ConnID => InC <- C END;
      NOT (Null(InC) OR InC.State = "Awaiting OpenConn") =>
        BEGIN
          OutC.Disposition <-
            "Already have connection of that ID";
          Deliver(OutC, SP);
        END;
      Null(InC) =>
        BEGIN
          Insert(SP.ConnectionS, OutC);
          OutC.State <- "Awaiting CONN-OPEN";
          SendOpenConn(OutC, SP);
        END;
      ^;
```

```
        OutC.DestProcess # InC.SourceProcess =>
          BEGIN
            OutC.Disposition <-
              "Connection not to process named";
            Deliver(OutC, SP);
          END;
        StopTiming(InC);
        Remove(SP.ConnectionS, InC);
        BEGIN
          CompatibleConnTypes(InC, OutC, OutC.Disposition) =>
            BEGIN
              MergeConnHandles(OutC, InC);
              Insert(SP.ConnectionS, OutC);
              OutC.State <- "Connection open";
              OutC.Disposition <- "Normal";
              SendOpenConn(OutC, SP);
            END;
          DP == ProcessHandle;
          OutC.DestProcess.Host = LocalHost ->
            DP <- SeizeProcessHandle(OutC.DestProcess);
          RejectConn(OutC, OutC.Disposition, DP);
          Release(DP);
        END;
        Deliver(OutC, SP);
        Free(InC);
      END;

MergeConnHandles <-
  !
  ! OutC and InC are matching connection open requests. OutC will
  !  become the permanent record of the connection, so merge info
  !  from InC into it.
  !
  ! */
  EXPR(OutC:ConnHandle, InC:ConnHandle)
    BEGIN
      OutC.DestID <- InC.SourceID;
      OutC.RemoteSocket <- InC.RemoteSocket;
    END;

^;
```

```
EnQInputOpenConn <-
'
  ! InC represents an incoming request to open a connection.
  ! SP is non-null if the request is from a local process, in which case
  !  it is the handle for that process.
  ! Assume that InC.ConnBytesize and InC.ConnType are legal.
  ! Look for destination process. If found, call it DP. If not,
  !  reject the connection request.
  ! Look for a pending OpenConn by DP with same connection ID
  !  as InC.  If none found, then either reject InC (if it mentions
  !  a particular transaction), or enqueue it to await the local open.
  ! If a pending output request is found call it OutC. Check that
  !  the remote process is the same for InC as for OutC and that they
  !  agree on the local transaction ID if InC claims to know it.
  ! Having established that InC and OutC refer to the same connection,
  !  check that a CONN-OPEN is acceptable in its current state. If
  !  not, reject the open.  If state is "Awaiting CONN-OPEN or
  !  CONN-CLOSE", then adjust the state but otherwise ignore the open.
  ! Stop timing on OutC and remove it from DP.ConnectionS.
  ! Check the compatibility of InC and OutC.  If they are compatible,
  !  mark OutC complete and deliver it to DP.  If InC and OutC dont match
  !  abort the local OpenConn, which will cause a CONN-CLOSE to be sent to the
  !  remote process. In either case, enter OutC in DP.ConnectionS, and
  !  discard InC. If the connection has been aborted, OutC sits in
  !  DP.ConnectionS until an answering CLOSE-CONN is received or a
  !  time-out occurs.
  !
  ' */
    EXPR(InC:ConnHandle, SP:ProcessHandle)
      BEGIN
        DP <== SeizeProcessHandle(InC.DestProcess);
        Null(DP) =>
          RejectConn(InC, "Destination process unknown", SP);
        OutC == ConnHandle;
        BEGIN
          FOREACH C IN DP.ConnectionS
            DO C.ConnID = InC.ConnID => OutC <- C END;
          Null(OutC) =>
            BEGIN
              NOT Null(InC.DestID) =>
                RejectConn(InC,
                           "Referenced connection transaction does not exist",
                           SP);
              Cardinality(DP.ConnectionS) GE MaxConnections =>
                RejectConn(InC,
                           "Destination process connection limit reached",
                           SP);
              InC.State <- "Awaiting OpenConn";
              StartTiming(InC, AwaitingOpenConnInterval);
              Insert(DP.ConnectionS, InC);
            END;
          ^;
```

```
          OutC.DestProcess # InC.SourceProcess OR
            NOT Null(InC.DestID) AND OutC.SourceID # InC.DestID =>
            RejectConn(InC,
                       "Connection ID inconsistent with local connection info",
                       SP);
          OutC.State = "Awaiting CONN-OPEN or CONN-CLOSE" =>
            OutC.State <- "Awaiting CONN-CLOSE";
          OutC.State # "Awaiting CONN-OPEN" =>
            RejectConn(InC, "Already have connection of that ID",
                       SP);
          StopTiming(OutC);
          MergeConnHandles(OutC, Inc);
          CompatibleConnTypes(InC, OutC, OutC.Disposition) =>
            BEGIN
              OutC.State <- "Connection open";
              OutC.Disposition <- "Normal";
              Deliver(OutC, SP);
            END;
          OutC.State <- "Mismatch - awaiting CONN-CLOSE";
          SendCloseConn(OutC, OutC.Disposition, DP);
        END;
        Release(DP);
      END;

CompatibleConnectionTypes <-
  '
  ! Returns TRUE iff C1 and C2 have the same byte size and complementary
  !  connection types.
  ! Sets disposition appropriately if there is a mismatch.
  !
  ' */
  EXPR(C1:ConnHandle,
       C2:ConnHandle,
       Disposition:ReasonCode SHARED;
       BOOL)
    BEGIN
      C1.ConnBytesize # C2.ConnBytesize =>
        BEGIN
          Disposition <- "Connection byte size mismatch";
          FALSE;
        END;
      CASE[C1.ConnType, C2.ConnType]
        ["ServerTELNET", "UserTELNET"],
          ["UserTELNET", "ServerTELNET"] => TRUE;
        ["Binary", "Binary"] =>
          CASE[C1.ConnDirection, C2.ConnDirection]
            ["InOut", "InOut"], ["In", "Out"], ["Out", "In"] =>
              TRUE;
            TRUE => FALSE;
          END;
        TRUE => FALSE;
      END => TRUE;
      Disposition <- "Connection type mismatch";
      FALSE;
    END;

^;
```

```
OpenConn <-

! If OutC is addressed to a local process, then simulate
!  transmission of a CONN-OPEN by copying OutC and calling
!  the handler for input CONN-OPENs.
! Otherwise, send a CONN-OPEN to the remote process.
!
! */
  EXPR(OutC:ConnHandle, SP:ProcessHandle)
    BEGIN
      OutC.DestProcess.Host = LocalHost =>
        BEGIN
          InC <== ReverseConnHandle(OutC);
          EnQInputOpenConn(InC, SP);
          Free(InC);
        END;
      OutC.ProtocolCommand <- "CONN-OPEN";
      DeliverToRemoteHost(OutC, OutC.DestProcess.Host);
    END;

SendCloseConn <-
 !
! If OutC is addressed to a local process, then simulate
!  transmission of a CONN-CLOSE by copying OutC and calling
!  the handler for input CONN-CLOSEs.
! Otherwise, send a CONN-CLOSE to the remote process.
!
! */
  EXPR(OutC:ConnHandle, Reason:ReasonCode, SP:ProcessHandle)
    BEGIN
      OutC.Reason <- Reason;
      OutC.DestProcess.Host = LocalHost =>
        BEGIN
          InC <== ReverseConnHandle(OutC);
          InC.Reason <- OutC.Reason;
          EnQInputCloseConn(InC, SP);
          Free(InC);
        END;
      OutC.ProtocolCommand <- "CONN-CLOSE";
      DeliverToRemoteHost(OutC, OutC.DestProcess.Host);
    END;
```

```
ReverseCorrHandle <-
  !
  ! Produce a copy of C with Source and Destination reversed, so that
  !  a reply to C can be sent.
  !
  ! */
    EXPR(C:CorrHandle; CorrHandle)
      BEGIN
        InC <== CopyCorrHandle(OutC);
        InC.SourceID <- OutC.DestID;
        InC.SourceProcess <- OutC.DestProcess;
        InC.DestID <- OutC.SourceID;
        InC.DestProcess <- OutC.SourceProcess;
        InC;
      END;

^;
```

```
EnQOutputCloseConn <-
  '
  ! NewC represents either a CloseConn primitive or an OpenConn for which
  !  an error has occurred trying to establish the connection.
  ! Either way, it is known that an OpenConn has been performed with identifier
  !  NewC.ConnID.  Call the original ConnHandle OldC (which will be identical
  !  to NewC if this call results from an error).
  ! Check that the destination process is consistent with the original open.
  ! Substitute NewC for OldC in the TransactionTable entry
  !  corresponding to this transaction. A single transaction ID will
  !  be used for both primitives.
  ! Take action depending on OldC.State.
  !
  ' */
    EXPR(NewC:ConnHandle, SP:ProcessHandle)
      BEGIN
        OldC <==
          BEGIN
            NewC.Op # "CloseConnection" => NewC;
            SeizeTransaction(NewC.SourceID);
          END;
        FOREACH C IN SP.ConnectionS
          DO
            NewC.ConnID = C.ConnID =>
              [) OldC <- C; OldC # NewC -> Seize(OldC) (];
          END;
        BEGIN
          OldC.DestProcess # NewC.DestProcess =>
            BEGIN
              NewC.Disposition <-
                "Connection not to process named";
              Deliver(NewC, SP);
            END;
          ReplaceTransactionHandle(OldC.SourceID, NewC);
          CASE[OldC.State]
            ["Awaiting CONN-OPEN"] =>
              '
              ! User process issued Close before completion
              !  of Open. Abort the OpenConn, send CONN-CLOSE,
              !  wait for possible CONN-OPEN, then CONN-CLOSE,
              !  and abort the CloseConn after this exchange.
              !
              ' */
                BEGIN
                  StopTiming(OldC);
                  OldC.Disposition <- "Superseded by close";
                  Remove(SP.ConnectionS, OldC);
                  Deliver(OldC, SP);
                  Release(OldC);
                  NewC.State <-
                    "Awaiting CONN-OPEN or CONN-CLOSE";
                  NewC.Disposition <- "Connection not yet open";
                  Insert(SP.ConnectionS, NewC);
                  StartTiming(NewC);
```

```
            SendCloseConn(NewC, "Open superseded by close",
                                SP);
        END;
    ["Connection open"] =>
      '
      ! Could be a normal Close or error during connection attempt.
      ! Replace OldC by NewC in the connection set. (Do this before
      !  freeing OldC, in case OldC = NewC!).
      ! Send CONN-CLOSE and wait for acknowledgement.
      ! NewC.Disposition indicates error, if any.
      !
      ' */
        BEGIN
          Remove(SP.ConnectionS, OldC);
          MergeOpenIntoClose(NewC, OldC);
          Insert(SP.ConnectionS, NewC);
          Free(OldC);
          NewC.State <- "Awaiting CONN-CLOSE";
          StartTiming(NewC, AwaitingCONN\CLOSEInterval);
          SendCloseConn(NewC, NewC.Disposition, SP);
        END;
    ["Mismatch - awaiting CONN-CLOSE"] =>
      '
      ! An OpenConn is pending. Abort it immediately.
      ! CONN-CLOSE has already been sent. Await the reply,
      !  then abort the CloseConn.
      !
      ' */
        BEGIN
          StopTiming(OldC);
          Remove(SP.ConnectionS, OldC);
          MergeOpenIntoClose(NewC, OldC);
          Deliver(OldC, SP);
          NewC.State <- "Awaiting CONN-CLOSE";
          NewC.Disposition <- "Connection not yet open";
          Insert(SP.ConnectionS, NewC);
          StartTiming(NewC);
        END;
    ["Awaiting CloseConn"] =>
      '
      ! Could be normal Close or error during attempt to connect.
      ! NewC.Disposition indicates which.
      ! (Freeing of OldC must follow delivery of NewC in case the
      !  two are identical!)
      !
      ' */
        BEGIN
          StopTiming(OldC);
          Remove(SP.ConnectionS, OldC);
          Deliver(NewC, SP);
          Free(OldC);
        END;
    ["Awaiting CONN-CLOSE"],
      ["Awaiting CONN-OPEN or CONN-CLOSE"] =>
      BEGIN
```

```
                NewC.Disposition <- "Redundant close";
                Deliver(NewC, SP);
              END;
          END;
        END;
        OldC # NewC -> Release(OldC);
      END;

MergeOpenIntoClose <-
    '
    ! CloseC is about to replace OpenC in a set of ConnectionS.
    ! Copy the necessary connection information from OpenC to CloseC.
    !
    ' */
      EXPR(CloseC:ConnHandle, OpenC:ConnHandle)
        BEGIN
          CloseC.DestID <- OpenC.DestID;
          CloseC.Connection <- OpenC.Connection;
          CloseC.LocalSocket <- OpenC.LocalSocket;
          CloseC.RemoteSocket <- OpenC.RemoteSocket;
        END;

RejectConn <-
    '
    ! InC is an incoming ConnOpen or ConnClose request, which must
    !  be rejected.
    ! SP is the source process, if local, otherwise null. In the local case,
    !  it is not necessary to seize the source transaction because
    !  the calling path already has it.

    !
    ' */
      EXPR(InC:ConnHandle, Reason:ReasonCode, SP:ProcessHandle)
        BEGIN
          Null(SP) => SendCONN\REJ(ReverseConnHandle(Inc), Reason);
          RejectOutputConn(LookupTransaction(InC.SourceID), Reason,
                           SP);
        END;

RejectOutputConn <-
    '
    ! OutC is an output OpenConn or CloseConn from process SP.
    ! It is to be rejected for the Reason given.
    ! Be sure OutC is in the proper state to receive a rejection.
    ! If so, abort OutC, passing along the reason for rejection unless
    !  OutC holds a prior error code, which takes precedence.
    !
    ' */
      EXPR(OutC:ConnHandle, Reason:ReasonCode, SP:ProcessHandle)
        BEGIN
          OldC.State = "Connection open" OR
            OldC.State = "Awaiting OpenConn" OR
            OldC.State = "Awaiting CloseConn" =>
            MSGError('Connection rejection in wrong state');
          StopTiming(OldC);
```

```
        IsNormal(OldC.Disposition) -> OldC.Disposition <- Reason;
        Remove(SP.Connections, OldC);
        Deliver(OldC, SP);
      END;

^;
```

```
EnQInputCloseConn <-
  '
  ! NewC represents an incoming connection close request.
  ! If SP is non-null, then the connection is intrahost, and SP represents
  !  the source of this Close.
  ! Look for a connection record that either represents a previous input
  !  CONN-OPEN (State = "Awaiting OpenConn"), or the local response to such
  !  an input.
  ! If no destination or no matching connection can be found, reject the
  !  Close unless the reason accompanying it is "ACKing your close". Such
  !  a belated Close could result from timeouts.
  ! Otherwise, take action appropriate to the state of the connection found.
  !
  ' */
   EXPR(NewC:ConnHandle, SP:ProcessHandle)
     BEGIN
       DP <== SeizeProcessHandle(NewC.DestProcess);
       Null(DP) =>
         NewC.Reason # "ACKing your close" ->
           RejectConn(NewC, "Destination process unknown", SP);
       BEGIN
         OldC == ConnHandle;
         FOREACH C IN DP.ConnectionS
           DO
             MatchingConn(C, NewC) =>
               [) OldC <- C; Seize(OldC) (];
           END;
         Null(OldC) =>
           NewC.Reason # "ACKing your close" ->
             RejectConn(NewC, "Unknown connection", SP);
         CASE[OldC.State]
           ["Awaiting OpenConn"] =>
             '
             ! Sender is withdrawing a previous CONN-OPEN.
             ! Acknowledge and discard transaction.
             !
             ' */
               BEGIN
                 StopTiming(OldC);
                 Remove(DP.ConnectionS, OldC);
                 SendCloseConn(ReverseConnHandle(OldC),
                               "ACKing your close", DP);
               END;
           ["Connection open"] =>
             '
             ! Sender is initiating a connection close exchange.
             ! Wait for local process to respond.
             !
             ' */
               BEGIN
                 OldC.State <- "Awaiting CloseConn";
                 StartTiming(OldC, AwaitingCloseConnInterval);
               END;
           ["Awaiting CloseConn"] =>
```

```
            ["Awaiting CONN-OPEN"] =>
              MSGError('Improper response to CONN-OPEN');
            ["Awaiting CONN-CLOSE"],
              ["Mismatch - awaiting CONN-CLOSE"],
              ["Awaiting CONN-OPEN or CONN-CLOSE"] =>
              BEGIN
                StopTiming(OldC);
                Remove(DP.ConnectionS, OldC);
                Deliver(OldC, DP);
              END;
          END;
          Release(OldC);
        END;
        Release(DP);
      END;

MatchingConn <-
  EXPR(OldC:ConnHandle, NewC:ConnHandle; BOOL)
    BEGIN
      OldC.State = "Awaiting OpenConn" =>
        OldC.SourceProcess = NewC.SourceProcess AND
          OldC.SourceID = NewC.SourceID;
      OldC.DestProcess = NewC.SourceProcess AND
        OldC.DestID = NewC.SourceID AND
        (Null(NewC.DestID) OR OldC.SourceID = NewC.DestID);
    END;

RecordCONN\REJ <-
  '
  ! CR is an incoming CONN-REJ item.
  ! If the designated transaction cannot be found, ignore CR.
  ! Otherwise, abort and deliver the original OpenConn or CloseConn.
  !
  ' */
    EXPR(CR:ConnBlock)
      BEGIN
        OldC <== SeizeTransaction(CR.DestID);
        BEGIN
          Null(OldC) OR NOT IsConnHandle(OldC) OR
            OldC.SourceProcess # CR.DestProcess OR
            OldC.DestProcess # CR.SourceProcess =>
            /* 'Transaction could have timed out';
          SP <== SeizeProcessHandle(OldC.SourceProcess);
          RejectOutputConn(OldC, CR.Reason, SP);
          Release(SP);
        END;
        Release(OldC);
      END;

^;
```

```
! - - - - - - - - - - - - - - - - - -
!
!          Utility Routines
!
! - - - - - - - - - - - - - - - - - - - ' */ -;

DeliverToRemoteHost <-
  '
  ! T represents a transaction for which a protocol item must be sent
  !  out to host H.
  !
  ' */
   EXPR(T:TransactionHandle, H:HostCode)
     BEGIN
       RH <== SeizeHostHandle(T.DestProcess.Host);
       Deliver(T, RH);
       Release(RH);
     END;

Deliver <-
  '
  ! U is either a local process or a remote host.
  ! T is a transaction with output ready for U.
  ! Let SH be the delivery server for U.
  ! If T is an alarm, give it special priority.
  ! If SH is asleep, SIGNAL to wake it up, and set a flag to avoid
  !  redundant SIGNALs by other paths.
  !
  ' */
   EXPR(T:TransactionHandle, U:UserHandle)
     BEGIN
       DECL SH:ServerHandle LIKE Seize(U.DeliveryServer);
       NonAlarmsQueued <== FALSE;
       FOREACH QT AT QTLocation FromFrontOf SH.DeliveryQ
         DO
           NOT IsAlarmHandle(QT) =>
             BEGIN
               NonAlarmsQueued <- TRUE;
               EnQAt(QTLocation, T);
             END;
         END;
       NOT NonAlarmsQueued -> EnQ(SH.DeliveryQ, T);
       IsProcessHandle(U) -> T.State <- "Delivered";
       NOT SH.Running ->
         [) SIGNAL(SH.WakeUpSignal); SH.Running <- TRUE (];
       Release(SH);
     END;
```

```
MatchingProcessNames <-
    '
    ! Return TRUE iff the specific process name Name is compatible
    !  with GName, which may or may not be generic.
    !
    ' */
     EXPR(GName:ProcessName, Name:ProcessName; BOOL)
       GName.Host = Name.Host AND
         GName.Incarnation = Name.Incarnation AND
         GName.GenericName = Name.GenericName AND
         (Null(GName.Instance) OR GName.Instance = Name.Instance);

  EncodeGenericName <-
    '
    ! Translate generic name strings into generic codes.
    ! Return FALSE iff the given string is not a valid name.
    !
    ' */
     EXPR(Class:Union(StringPtr, GenericClassCode) SHARED; BOOL)
       BEGIN
         NOT IsStringPtr(Class) => FALSE;
         Found <== FALSE;
         FOREACH GH IN GenericTable
           DO
             Value(GH.Class) = Value(Class) =>
               [) Found <- TRUE; Class <- GH.Code (];
           END;
         Found;
       END;

   DecodeGenericClass <-
     '
     ! Translate a generic class code into the corresponding string.
     !
     ' */
      EXPR(Class:Union(StringPtr, GenericClassCode) SHARED)
        BEGIN
          IsStringPtr(Class) => /* 'Already decoded';
          Class <- GenericTable[Class].Class;
        END;

   ^;
```

```
! - - -
!
! Transaction Table Access
!
! TransactionTable maps TransactionIDs into TransactionHandles.
!  One possible method of encoding table indices as part of transaction
!  identifiers is described in the MSG Design Specification.
!
! TransactionTable entries are allocated by AssignTransactionID and
!  freed by Free.
!
! */ -;

LookupTransaction <-
  EXPR(ID:TransactionID; TransactionHandle) TransactionTable[ID];

SeizeTransaction <-
  EXPR(ID:TransactionID; TransactionHandle)
    Seize(LookupTransaction(ID));

ReplaceTransactionHandle <-
  EXPR(ID:TransactionID, TH:TransactionHandle)
    LookupTransaction(ID) <- TH;

! - - -
!
! Process Table Access
!
! ProcessTable maps local process instance identifiers to ProcessHandles
!  A method of encoding table indices in instance numbers which results in
!  infrequent reuse of instance identifiers is described in the MSG Design
!  Specification.
!
! */ -;

LookupProcessHandle <-
  EXPR(P:ProcessName; ProcessHandle) ProcessTable[P.Instance];

SeizeProcessHandle <-
  EXPR(P:ProcessName; ProcessHandle)
    Seize(LookupProcessHandle(P));
```

```
!
!----------------------------------------------------------------------
!
!                          Cancelling Transactions
!
!----------------------------------------------------------------------

!    There are four ways a transaction event may terminate abnormally:
!       . Rescind
!       . StopMe
!       . host death
!       . timeout
!    These possibilities are considered in the table which follows. It contains
!all possible transaction states and the action to be taken for each state.
!Table entries have the form:
!       state name
!           conditions precedent to this state
!           remote action taken  ien entering this state
!           local action taken when cancelling
!           remote action taken when cancelling
!               (parenthetical remarks indicate later independent responses)
!           possibility of Rescind
!T and P are the transaction at issue and its owning process.  The S and D
!prefixes indicate source and destination respectively.  R is the delivery
!queue of the network server.  TinR is true when T is in R.';

'                    Mess Output
!
!  Awaiting MESS-OK
!       OutputMess | Awaiting XMIT & XMIT received
!       send MESS
!       [) StopMe => DeQ(SP.OutputMessQ, ST); RejectOutputMess(SP) (]
!       TinR -> [) Old => set MESS-CANCEL; DeQ(R, ST) (]
!        (Not TinR: MESS-HOLD received -- send MESS-CANCEL)
!       TinR -> rescissible
!  Awaiting XMIT (remote)
!       Awaiting MESS-OK & MESS-HOLD received
!       send HOLD-OK
!       [) StopMe => DeQ(SP.OutputMessQ, ST); RejectOutputMess(SP) (]
!       [) TinR => set; send (] MESS-CANCEL
!       rescissible
!  Awaiting XMIT (local)
!       OutputMess
!       --
!       [) StopMe => DeQ(SP.OutputMessQ, ST); RejectOutputMess(SP) (];
        DeQ(DP.InputMessQ, DT)
!       --
!       rescissible
!  Awaiting prior MESS completion
!       OutputMess & (sequence | stream-mark) block
!       --
!       [) StopMe => DeQ(SP.OutputMessQ, ST); RejectOutputMess(SP) (]
!       --
!       rescissible';
```

```
'               Mess Input
!
! Awaiting HOLD-OK
!       MESS received & no buffer space
!       send MESS-HOLD
!       DeQ(DP.InputMessQ, DT)
!       TinR -> set MESS-REJ
!        (Not TinR: HOLD-OK received -- send MESS-REJ)
!       --
! Held by sender (remote)
!       Awaiting HOLD-OK & HOLD-OK received
!       --
!       DeQ(DP.InputMessQ, DT)
!       send MESS-REJ
!       --
! Held by sender (local)
!       local mess received & no dest buffer space & source buffer space
!       --
!       RejectOutputMess(SP); DeQ(DP.InputMessQ, DT)
!       --
!       --
! Awaiting retransmission
!       Held by remote sender & buffer space
!       send XMIT
!       DeQ(DP.InputMessQ, DT); RemitBuffer(DP, DT)
!       TinR -> set MESS-REJ
!       --
! Awaiting ReceiveMess
!       local mess received ¦ Held by local sender & buffer space
!        ¦ MESS received & buffer space
!       remote -> send MESS-OK
!       DeQ(DP.InputMessQ, DT); CreditBuffer(DP, DT)
!       --
!       --';

'               Receive Mess
!
! Awaiting MESS
!       ReceiveMess
!       --
!       DeQ([] IsGeneric(DT) => DPsGenericHandle; DP (].ReceiveMessQ, DT)
!       --
!       rescissible';
```

```
'               Alarm Output
!
! Awaiting ALARM-OK
!       OutputAlarm
!       send ALARM
!       Remove(SP.OutputAlarmS, ST)
!       TinR -> DeQ(R, ST)
!       TinR -> rescissible';

'               Alarm Input
!
! Awaiting ReceiveAlarm
!       ALARM received
!       send ALARM-OK
!       DeQ(DP.AlarmInputQ, DT)
!       --
!       --';

'               Alarm Receive
!
! Awaiting ALARM
!       ReceiveAlarm
!       --
!       DeQ(DP.ReceiveAlarmQ, DT)
!       --
!       rescissible';

'               Conn Open
!
! Awaiting CONN-OPEN
!       OpenConn
!       send CONN-OPEN
!       Remove(P.ConnectionS, T);
!       [) TinR => DeQ(R, T); send CONN-CLOSE (]
!       TinR -> rescissible
! Awaiting OpenConn
!       CONN-OPEN received
!       --
!       Remove(P.ConnectionS, T)
!       send CONN-REJ
!       --
! Connection open
!       OpenConn & CONN-OPEN received
!       --
!       Remove(P.ConnectionS, T)
!       send CONN-CLOSE
!       StopMe only';
```

```
'              Conn Close
!
! Awaiting CONN-CLOSE
!      CloseConn
!      send CONN-CLOSE
!      [) (rescind & TinR & viable connection) => set Connection open;
!       Remove(P.ConnectionS, T) (]
!      (rescind & TinR) -> DeQ(R, T)
!      (viable connection & TinR) -> rescissible
! Awaiting CloseConn
!      CONN-CLOSE received
!      --
!      Remove(P.ConnectionS, T); (timeout -> activate ConnBroken signal)
!      send  CONN-CLOSE
!      --
! Awaiting CONN-OPEN or CONN-CLOSE
!      Awaiting CONN-OPEN & CloseConn
!      send CONN-CLOSE
!      Remove(P.ConnectionS, T)
!      --
!      --
! Mismatch -- awaiting CONN-CLOSE
!      CONN-OPEN mismatch
!      send CONN-CLOSE
!      Remove(P.ConnectionS, T)
!      --
!      --';

'
!      the Delivered state -- Awaiting user | network delivery
!                        No action';

'
!Notes
!    rescind
!      if not rescissible, do nothing
!      stop timing
!      local and remote cleanup
!      direct delivery
!    StopMe
!      stop timing
!      local and remote cleanup
!      no delivery
!    deadhost
!      stop timing
!      local cleanup
!      delivery via server
!    timeout
!      local and remote cleanup
!      delivery via server';

^;
```

```
RescindPendingEvent <-
  !
  !     RescindPendingEvent encodes the part of the above table
  !      having to do specifically with rescission.  First a test is
  !      done to determine if the event is in a rescissible state.
  !      If it is then both network and local action are taken.
  !      Delivery is done by the calling routine.  Cancelling message
  !      transactions where both sender and receiver are local is
  !      handled specially.
  !
  ! */
    EXPR(E:TransactionHandle, EsProcess:DestHandle; ReasonCode)
      BEGIN
        DECL Rescissible:BOOL;
        Rescissible <-
          CASE[E.State]
            ["Awaiting XMIT"],
              ["Awaiting prior MESS completion"],
              ["Awaiting MESS"],
              ["Awaiting ALARM"] => TRUE;
            ["Awaiting MESS-OK"],
              ["Awaiting ALARM-OK"],
              ["Awaiting CONN-OPEN"] =>
              HasPendingProtocolOutput(E);
            ["Awaiting CONN-CLOSE"] =>
              HasPendingProtocolOutput(E) AND
                NOT Null(E.Connection);
            TRUE => FALSE;
          END;
        NOT Rescissible => "Unable to Rescind";
        StopTiming(E);
        IsLocalTransaction(E) =>
          BEGIN
            RescindLocalEvent(E, EsProcess);
            "Event rescinded";
          END;
        E.State = "Awaiting CONN-CLOSE" =>
          BEGIN
            StopProtocolOutput(E)
            E.State <- "Connection open";
            "Event rescinded";
          END;
        RemoteCancel(E, EsProcess,
                     BEGIN
                       MD(E) = MessHandle =>
                         "Message rescinded or timed out";
                       "Transaction rescinded";
                     END);
        E.Disposition <- "Event rescinded";
        LocalCancel(E, EsProcess);
        "Event rescinded";
      END;
```

```
RescindLocalEvent <-
  '
  !     RescindLocalEvent handles "Awaiting XMIT" rescission where
  !      both the sender and receiver are local.  The source event is
  !      rejected and the destination event is dequeued.
  !
  ' */
    EXPR(ST:TransactionHandle, SP:DestHandle)
      BEGIN
        DECL DP:ProcessHandle LIKE
          SeizeDestHandle(ST.DestProcess);
        DECL DT:TransactionHandle LIKE
          SeizeTransaction(ST.DestID);
        ST.Disposition <- "Event rescinded";
        RejectOutputMess(ST, ST.Disposition, SP,
                         TRUE IS NoDeliver);
        DeQ(DP.InputMessQ, DT);
        Release(DP);
        Free(DT);
      END;

^;
```

```
StopTransaction <-
  !
  !     StopTransaction encodes the part of the above table having
  !      to do specifically with StopMe.  It can be called with
  !      T in any state. Both network and local action are taken.
  !      No delivery is done, since the process is terminated.
  !      Local message output transactions are treated specially
  !      (activating messages "Awaiting prior MESS completion" via
  !      RejectOutputMess is avoided).
  !
  ! */
    EXPR(T:TransactionHandle, TsProcess:DestHandle)
      BEGIN
        IsLocalTransaction(T) =>
          StopLocalTransaction(T, TsProcess);
        T.State = "Connection open" =>
          BEGIN
            SendCloseConn(T, "Process terminated", TsProcess);
            Remove(TsProcess.ConnectionS, T);
          END;
        IsPassiveTransaction(T) => NOTHING;
        StopTiming(T);
        RemoteCancel(T, TsProcess, "Process terminated");
        BEGIN
          T.State = "Awaiting MESS-OK" OR
            T.State = "Awaiting XMIT" OR
            T.State = "Awaiting prior MESS completion" =>
            DeQ(TsProcess.OutputMessQ, T);
          LocalCancel(T, TsProcess);
        END;
      END;

StopLocalTransaction <-
  !
  !     Stop handles message transactions where both the sender and
  !      receiver are local.  The source transaction is dequeued if
  !      owned by the stopping process, otherwise it is rejected and
  !      delivered.  The destination transaction is dequeued.
  !
  ! */
    EXPR(T:TransactionHandle, P:DestHandle)
      BEGIN
        DECL IsSender:BOOL LIKE T.State = "Awaiting XMIT";
        DECL MatchingP:ProcessHandle LIKE
          SeizeDestHandle(BEGIN
                            IsSender => T.DestProcess;
                            T.SourceProcess;
                          END);
        DECL MatchingT:TransactionHandle LIKE
          SeizeTransaction(BEGIN
                             IsSender => T.DestID;
                             T.SourceID;
                           END);
        BEGIN
```

```
        IsSender =>
          BEGIN
            DeQ(P.OutputMessQ, T);
            DeQ(MatchingP.InputMessQ, MatchingT);
          END;
|       MatchingT.Disposition <- "Process terminated";
|       RejectOutputMess(MatchingT, MatchingT.Disposition,
|                        MatchingP, TRUE IF NoDeliver);
        DeQ(P.InputMessQ, T);
        Deliver(MatchingT, MatchingP);
      END;
      Release(MatchingP);
      Free(MatchingT);
    END;

HostDeadTransaction <-
|   !
|   !     HostDeadTransaction encodes the part of the above table
|   !      having to do specifically with the failure of an MSG.
|   !      Only local action is taken, and the result is delivered via
|   !      server.  If the owning process cannot be seized FALSE is
|   !      returned.
|   !
|   ! */
    EXPR(T:TransactionHandle; BOOL)
      BEGIN
        IsPassiveTransaction(T) => TRUE;
        DECL TsProcess:DestHandle LIKE
          TestSeizeDestHandle(OwningProcess(T));
        Null(TsProcess) => FALSE;
        StopTiming(T);
        DECL TsOldState:ReasonCode LIKE T.State;
        T.Disposition <- "Host died";
        LocalCancel(T, TsProcess);
        DeliverCancel(T, TsProcess, TsOldState);
        Release(TsProcess);
        TRUE;
      END;

 ^;
```

```
TimeoutTransaction <-
  '
  !     TimeoutTransaction encodes the part of the above table
  !      having to do specifically with timeout.  Both network and
  !      local action are taken, and the result is delivered via
  !      server. If the owning process cannot be seized FALSE is
  !      returned.  Cancelling message transactions where both sender
  !      and receiver are local is handled specially.
  !
  ' */
  EXPR(T:TransactionHandle; BOOL)
    BEGIN
      DECL TsProcess:DestHandle;
      IsLocalTransaction(T) => TimeoutLocalTransaction(T);
      TsProcess <- TestSeizeDestHandle(OwningProcess(T));
      Null(TsProcess) => FALSE;
      DECL TsOldState:StateCode LIKE T.State;
      RemoteCancel(T, TsProcess,
                  BEGIN
                    T.State = "Awaiting CONN-OPEN" OR
                      T.State =
                        "Awaiting CONN-OPEN or CONN-CLOSE" =>
                      "Timed out waiting for your CONNECTION-OPEN";
                    IsMessHandle(T) =>
                      "Message rescinded or timed out";
                    "Transaction timed out";
                  END);
      T.Disposition <- "Transaction timed out";
      LocalCancel(T, TsProcess);
      TsOldState = "Awaiting CloseConn" ->
        T.Op <- "StopMe"             /* 'for connection-broken signal';
      DeliverCancel(T, TsProcess, TsOldState);
      Release(TsProcess);
      TRUE;
    END;

TimeoutLocalTransaction <-
  '
  !     TimeoutLocalTransaction handles message transactions where
  !      both the sender and receiver are local.  The source
  !      transaction is rejected, the destination transaction is
  !      dequeued and the source transaction is delivered.  If the
  !      owning process cannot be seized FALSE is returned.
  !
  ' */
  EXPR(T:TransactionHandle; BOOL)
    BEGIN
      DECL IsSender:BOOL LIKE T.State = "Awaiting XMIT";
      DECL SP:ProcessHandle LIKE
        TestSeizeDestHandle(T.SourceProcess);
      Null(SP) => FALSE;
      DECL DP:ProcessHandle LIKE
        TestSeizeDestHandle(T.DestProcess);
      Null(DP) => [) Release(SP); FALSE (];
```

```
        DECL MatchingT:TransactionHandle LIKE
          TestSeizeTransaction(BEGIN
                                    IsSender => T.DestID;
                                    T.SourceID;
                                  END);
        Null(MatchingT) => [) Release(SP); Release(DP); FALSE (];
        DECL ST:TransactionHandle LIKE
          [) IsSender => T; MatchingT (];
        DECL DT:TransactionHandle LIKE
          [) IsSender => MatchingT; T (];
        ST.Disposition <- DT.Disposition <- "Event timed out";
        RejectOutputMess(ST, ST.Disposition, SP,
                             TRUE IE NoDeliver);
        DeQ(DP.InputMessQ, DT);
        Deliver(ST, SP);
        Release(SP);
        Release(DP);
        Free(MatchingT);
        TRUE;
      END;

  ^;
```

```
LocalCancel <-
 !
 !    LocalCancel encodes the local action part of the above table.
 !     In general this consists of dequeuing the transaction from
 !     the proper process queue.
 !
 ! */
   EXPR(T:TransactionHandle, TsProcess:DestHandle)
     CASE[T.State]
       ["Awaiting MESS-OK"],
         ["Awaiting XMIT"],
         ["Awaiting prior MESS completion"] =>
         RejectOutputMess(T, T.Disposition, TsProcess,
                                 TRUE IE NoDeliver);
       ["Awaiting HOLD-OK"], ["Held by sender"] =>
         DeQ(TsProcess.InputMessQ, T);
       ["Awaiting retransmission"] =>
         BEGIN
           DeQ(TsProcess.InputMessQ, T);
           RemitBuffer(TsProcess, T);
         END;
       ["Awaiting ReceiveMess"] =>
         BEGIN
           DeQ(TsProcess.InputMessQ, T);
           CreditBuffer(TsProcess, T);
         END;
       ["Awaiting MESS"] =>
         BEGIN
           NOT T.IsGeneric => DeQ(TsProcess.ReceiveMessQ, T);
           DECL G:GenericHandle LIKE
             Seize(GenericTable[TsProcess.Name.GenericName]);
           DeQ(G.ReceiveMessQ, T);
           Release(G);
         END;
       ["Awaiting ALARM-OK"] =>
         Remove(TsProcess.OutputAlarmsS, T);
       ["Awaiting ReceiveAlarm"] =>
         DeQ(TsProcess.InputAlarmQ, T);
       ["Awaiting ALARM"] => DeQ(TsProcess.ReceiveAlarmQ, T);
       ["Awaiting CONN-OPEN"],
         ["Awaiting OpenConn"],
         ["Awaiting CONN-CLOSE"],
         ["Awaiting CloseConn"],
         ["Awaiting CONN-OPEN or CONN-CLOSE"],
         ["Mismatch -- awaiting CONN-CLOSE"] =>
         Remove(TsProcess.ConnectionS, T);
       ["Delivered"] => NOTHING;
       TRUE => MSGError('Bad cancel state');
     END;
```

```
RemoteCancel <-
  !
  !     RemoteCancel encodes the network action part of the above
  !      table.  In general this consists of sending a protocol item,
  !      or changing or removing a protocol item ready to be sent.
  !
  ! */
    EXPR(T:TransactionHandle,
         TsProcess:DestHandle,
         Why:ReasonCode)
      CASE[T.State]
        ["Awaiting MESS-OK"] =>
          BEGIN
            HasPendingProtocolOutput(T) =>
              BEGIN
                NOT Null(T.DestID) =>
                  ChangeProtocolOutput(T, "MESS-CANCEL", Why);
                StopProtocolOutput(T);
              END;
          END;
        ["Awaiting XMIT"] =>
          BEGIN
            HasPendingProtocolOutput(T) =>
              ChangeProtocolOutput(T, "MESS-CANCEL", Why);
            SendMESS\CANCEL(T, Why);
          END;
        ["Awaiting HOLD-OK"] =>
          HasPendingProtocolOutput(T) ->
            ChangeProtocolOutput(T, "MESS-REJ", Why);
        ["Held by sender"] => SendMESS\REJ(T, Why);
        ["Awaiting retransmission"] =>
          HasPendingProtocolOutput(T) ->
            ChangeProtocolOutput(T, "MESS-REJ", Why);
        ["Awaiting ALARM-OK"] =>
          HasPendingProtocolOutput(T) -> StopProtocolOutput(T);
        ["Awaiting CONN-OPEN"] =>
          BEGIN
            HasPendingProtocolOutput(T) => StopProtocolOutput(T);
            SendCloseConn(T, Why, TsProcess);
          END;
        ["Awaiting OpenConn"] => RejectConn(T, Why, TsProcess);
        ["Awaiting CloseConn"] =>
          SendCloseConn(T, Why, TsProcess);
        TRUE => NOTHING;
      END;

  ^;
```

```
IsPassiveTransaction <-
  '
  !     IsPassiveTransaction determines whether any cancel action need
  !      be taken.  If the transaction is awaiting local delivery it
  !      can be ignored.
  !
  ' */ EXPR(T:TransactionHandle; BOOL) T.State # "Delivered";

IsLocalTransaction <-
  '
  !     IsLocalTransaction determines whether a transaction is a
  !      message transaction with both sender and receiver local.
  !
  ' */
    EXPR(T:TransactionHandle; BOOL)
      T.State = "Awaiting XMIT" AND
        T.State.DestProcess.Host = LocalHost OR
        T.State = "Held by sender" AND
          T.SourceProcess.Host = LocalHost;

OwningProcess <-
  '
  !     OwningProcess returns the process name of the process for
  !      which the given transaction was created.
  !
  ' */
    EXPR(T:TransactionHandle; ProcessName)
      CASE[T.State]
        ["Awaiting HOLD-OK"],
          ["Held by sender"],
          ["Awaiting retransmission"],
          ["Awaiting ReceiveMess"],
          ["Awaiting MESS"],
          ["Awaiting ReceiveAlarm"],
          ["Awaiting ALARM"] => T.DestProcess;
        TRUE => T.SourceProcess;
      END;

HasPendingProtocolOutput <-
  '
  !     HasPendingProtocolOutput determines whether a transaction
  !      has a protocol item waiting to be sent.
  !
  ' */
    EXPR(T:TransactionHandle; BOOL) NOT Null(T.ProtocolCommand);
```

```
ChangeProtocolOutput <-
    '
    !     ChangeProtocolOutput changes the type of protocol item waiting
    !      to be sent (to one appropriate for cancellation).
    !
    ' */
      EXPR(T:TransactionHandle,
           Action:ProtocolCode,
           Why:ReasonCode)
        [) T.ProtocolCommand <- Action; T.Reason <- Why (];

StopProtocolOutput <-
    '
    !     StopProtocolOutput stops the sending of a protocol item
    !      waiting to be sent by dequeuing its transaction from the
    !      network delivery queue.
    !
    ' */
      EXPR(T:TransactionHandle)
        BEGIN
          DECL HC:HostCode LIKE T.DestProcess.Host;
          DECL HH:HostHandle LIKE
            FOREACH H IN HostS DO HC = H.Host => H END;
          DECL HQ:Queue(Transaction) LIKE
            HH.DeliverServer.DeliveryQ;
          Seize(HQ);
          DeQ(HQ, T);
          Release(HQ);
        END;

DeliverCancel <-
    '
    !     DeliverCancel determines whether a cancelled transaction need
    !      be delivered (via server), and if so delivers it.
    !
    ' */
      EXPR(T:TransactionHandle,
           TsProcess:DestHandle,
           TsOldState:ReasonCode)
        CASE[TsOldState]
          ["Awaiting MESS-OK"],
            ["Awaiting XMIT"],
            ["Awaiting prior MESS completion"],
            ["Awaiting MESS"],
            ["Awaiting ALARM-OK"],
            ["Awaiting ALARM"],
            ["Awaiting CONN-OPEN"],
            ["Awaiting CONN-CLOSE"],
            ["Awaiting CloseConn"],
            ["Awaiting CONN-OPEN or CONN-CLOSE"] =>
            Deliver(T, TsProcess);
          TRUE => NOTHING;
        END;

 ^;
```

```
'
!-----------------------------------------------------------------------
!
!                          Timer routines
!
!-----------------------------------------------------------------------
';

'
!       Deadlines are absolute fixed points, not intervals.
!       The front of the timer queue is the event which will time out earliest.
!        The FOREACH loop runs from the front of the queue to the back.';

StartTiming <-
    '
    !     StartTiming takes a transaction and adds it to the timer
    !      queue.  If a new deadline is supplied via NewTimer, it is
    !      assumed to be a relative interval, and is converted to
    !      absolute form.  The timer queue is scanned, the transaction
    !      inserted and, if the transaction is at the front of the
    !      queue, the timer process is notified.
    !
    ' */
    EXPR(NewEntry:TransactionHandle, NewTimer:Interval)
      BEGIN
        Seize(TimerQ);
        DECL IsAtFront:BOOL LIKE
          Null(TimerQ) OR
            NewEntry.Deadline LT Front(TimerQ).Deadline;
        NOT Null(NewTimer) ->
          NewEntry.Deadline <- MakeIntervalAbsolute(NewTimer);
        BEGIN
          Null(TimerQ) => FALSE;
          FOREACH Entry AT EsLocation FromFrontOf TimerQ
            DO
              NewEntry.Deadline LT Entry.Deadline =>
                [) EnQAt(EsLocation, NewEntry); TRUE (];
              FALSE;
            END;
        END #> EnQ(TimerQ, NewEntry);
        IsAtFront -> SIGNAL(TimerSignal);
        Release(TimerQ);
      END;
```

```
StopTiming <-
  '
  !     StopTiming takes a transaction and removes it from the timer
  !      queue.
  !
  ' */
    EXPR(OldEntry:TransactionHandle)
      BEGIN
        Seize(TimerQ);
        DeQ(TimerQ, OldEntry);
        Release(TimerQ);
      END;

TimeoutHandler <-
  '
  !     TimeoutHandler waits for timeouts and cancels transactions.
  !      It waits until the deadline of the front entry of the timer
  !      queue expires.  It also waits to be signalled from
  !      StartTiming that a new entry has been put on the front of
  !      the timer queue.  When a transaction times out an attempt is
  !      made to seize it and cancel it.  If this is successful the
  !      transaction is removed from the timer queue.  If this fails,
  !      TimeoutHandler waits a brief interval and tries again.
  !
  ' */
    EXPR()
      REPEAT
        Seize(TimerQ)
                /* 'This may wait for StartTiming or StopTiming to Release.';
        DECL TimeoutSignal:SignalType LIKE
          BEGIN
            Null(TimerQ) => NullSignal;
            StartClock(BEGIN
                          Front(TimerQ).Deadline LT CurrentTime() =>
                            MakeIntervalAbsolute(SeizeWait);
                          Front(TimerQ).Deadline;
                        END);
          END;
        Release(TimerQ);
        DECL WaitSignal:SignalType LIKE
          WAIT({ TimerSignal, TimeoutSignal });
        BEGIN
          WaitSignal = TimerSignal => StopClock(TimeoutSignal);
          Seize(TimerQ);
          FOREACH Entry AT EsLocation FromFrontOf TimerQ
            DO
              CurrentTime() LT Entry.Deadline => NOTHING;
              TestSeize(Entry) #> NOTHING;
              TimeoutTransaction(Entry) -> DeQAt(EsLocation);
              Free(Entry);
            END;
          Release(TimerQ);
        END;
      END;
```

```
'
!-----------------------------------------------------------------------
!
!                  MSG-MSG protocol format definitions
!
!-----------------------------------------------------------------------


!    The various fields of protocol items  input  from  the  network  must  be
!converted  from 8-bit byte agglomerations into usable objects, and the reverse
!must be done on output.  The  following  operators  define  and  perform  the
!conversion,  allowing  the  mass  of  network  data  to  be accessed as a data
!structure.
!    The FORMAT operator defines the simulated data structure of the  protocol
!items, with each field being given a name, a mode and a length in bytes.
!    Some field specifications vary from this general  form,  and  employ  the
!following  operators  in  their  definition.  The ~ operator indicates that a
!field is actually composed of a number of subfields, which are defined in  the
!FORMAT  that  ~  takes as its argument.  The ^ operator indicates that a field
!requires special processing to convert it, with ^ taking as its  argument  the
!routine  to do the processing.  The --- operator indicates that a field is not
!converted by FORMAT at all, but is included for documentation purposes.
!    The @ operator indicates a field that may be  assigned  directly  to  and
!from  an  appropriate  internal  data  structure.  Such direct assignments are
!performed with the <|| and ||> operators.
!    The | operator is used to reference  individual  fields,  the  format  to
!which the | applies having been chosen by either the <||, ||> or || operator.
';

Header <- FORMAT(Length:ShortInt(2), Command:ProtocolCode(1));

'
!      Mess-related formats';

MessFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         FirstByte:ShortInt(1),
         IsGeneric:@ BOOL,
         IsSequenced:@ BOOL,
         IsMarked:@ BOOL,
         NoHold:@ BOOL,
         HoldOk:@ BOOL,
         NoQWait:BOOL,
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName,
         Text:---);
```

```
MessOkFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

MessRejFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         Reason:@ ReasonCode(1),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

MessHoldFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

HoldOkFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

MessCancelFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         Reason:@ ReasonCode(1),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

XmitFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);
```

```
'
!       Alarm-related formats';

AlarmFormat
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         Alarm:@ AlarmCode(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

AlarmOkFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

AlarmRejFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         Reason:@ ReasonCode(2),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

'
!       Conn-related formats';

ConnOpenFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         ConnID:@ ConnIDCode(2),
         Type\duplex:BOOL,
         Type\send:BOOL,
         Type\receive:BOOL,
         Type\ServerTELNET:BOOL,
         Type\UserTELNET:BOOL,
         ConnBytesize:@ ShortInt(1),
         RemoteSocket:@ Integer(3),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);
```

```
ConnCloseFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         ConnID:@ ConnIDCode(2),
         Reason:@ ReasonCode(1),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

ConnRejFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:@ TransactionID(2),
         DestID:@ TransactionID(2),
         ConnID:@ ConnIDCode(2),
         Reason:@ ReasonCode(1),
         SourceProcess:@ ~ MSGProcessName,
         DestProcess:@ ~ MSGProcessName);

'
!       Other formats';

NoOpFormat <-
  FORMAT(Length:ShortInt(2), Command:ProtocolCode(1));

EchoFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         Data:ANY(1));

EchoReplyFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         Data:ANY(1));

ExpFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         Function:ANY(1));
```

```
SendStatusFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:TransactionID(2),
         SourceProcess:~ MSGProcessName,
         DestProcess:~ MSGProcessName);

StatusOkFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:TransactionID(2),
         SourceProcess:~ MSGProcessName,
         DestProcess:~ MSGProcessName,
         Status:---);

StatusRejFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         SourceID:TransactionID(2),
         Reason:ReasonCode(2),
         SourceProcess:~ MSGProcessName,
         DestProcess:~ MSGProcessName);

CloseFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         Reason:ReasonCode(2));

SynchFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         MyIncarnation:ShortInt(2),
         YourIncarnation:ShortInt(2),
         Version:ShortInt(2));

PtclErrFormat <-
  FORMAT(Length:ShortInt(2),
         Command:ProtocolCode(1),
         ErrorCode:ShortInt(2),
         BadTransaction:---);
```

```
'
!       The ProcessName format. ;

MSGProcessName <-
  FORMAT(Incarnation:? ShortInt(2),
         Instance:? ShortInt(2),
         GenericName:? ^ GenericClass);

GenericClass <-
   '
!       GenericClass is the routine invoked to parse the GenericName field.
!        The GenericName field has a conditional structure, unlike other
!        fields.  If the first byte of the field is greater than 127 then
!        it is a generic code, and there is no more to the field.  If it is
!        less than or equal to 127 it is the length of a text string naming
!        the class, with the text bytes following.';

NetBuffer <-
   '
   !    NetBuffer is the data structure used to buffer incoming and
   !     outgoing network data.  Data is the actual network data
   !     (2273 is the maximum number of bytes in the longest possible
   !     item).  Channel is the network channel used for transfers to
   !     and from the buffer.  Host is the target host.  Scan is a
   !     pointer to the last byte read or written (it is advanced
   !     by the <||, ||> and | operators as they process fields).
   !     Read points to the last byte RECEIVEd from the network.
   !
   ' */
    'STRUCT(Data:VECTOR(2273, NetByte),
         Channel:ChannelHandle,
         Host:HostCode,
         Scan:Integer,
         Read:Integer)';

NetReceiveItem <-
   '
!       NetReceiveItem reads a complete protocol item into the NetBuffer passed
!        to it.  It RECEIVEs the first two bytes (the length of the item) and
!        then RECEIVEs the rest.  If either a protocol error is detected
!        or a channel error occurs, an error ReasonCode is returned.';
```

```
NetSendItem <-
    '
!       NetSendItem SENDs the protocol item in the NetBuffer passed to it.
!        If a channel error occurs, FALSE is returned.';
^;
```

```
'
!-------------------------------------------------------------------------
!
!                        Protocol input routines
!
!-------------------------------------------------------------------------
';

ProtocolInput <-
    '
    !     ProtocolInput waits until either there is a new channel for
    !      it to listen to, or there is activity on one of the channels
    !      already assigned to it.  When a channel becomes active, the
    !      routine inputs a protocol item and dispatches to the proper
    !      specialized item handler.
    !
    ' */
    EXPR(Ego:ServerHandle)
      BEGIN
        DECL B:NetBuffer LIKE AllocateNetBuffer();
        REPEAT
          BEGIN
            DECL Signalor:UNION(ChannelHandle, SignalType) LIKE
              WAIT({ Ego.ChannelS, Ego.WakeUpSignal });
            Signalor = Ego.WakeUpSignal => NOTHING;
            B.Channel <- Signalor;
            B.Host <- GetHostFromChannel(B.Channel);
            DECL Result:ReasonCode LIKE NetReceiveItem(B);
            Result = "Protocol Error" => ProtocolError(B);
            Result = "Channel Error" =>
              BEGIN
                DECL LocalSocket:Socket LIKE
                  GetSocketFromChannel(B.Channel);
                CHCLOSE(B.Channel);
                DeassignMSGInputSocket(LocalSocket);
                Seize(Ego);
                Remove(Ego.ChannelS, B.Channel);
                Null(Ego.ChannelS) =>
                  BEGIN
                    DECL Restart:ContactHandle LIKE
                      Allocate(ContactHandle);
                    DECL ThisHost:HostHandle LIKE
                      SeizeHost(B.Host);
                    Restart.ProtocolCommand <- "Start ICP";
                    Restart.RemoteHost <- ThisHost;
                    Deliver(Restart, ThisHost);
                    Release(Restart);
                    Release(ThisHost);
                  END;
                Release(Ego);
              END;
            || Header;
            CASE[| Command]
              ["MESS"] => InputMess(B);
```

```
                ["MESS-OK"] => InputMessOk(B);
                ["MESS-REJ"] => InputMessRej(B);
                ["MESS-HOLD"] => InputMessHold(B);
                ["HOLD-OK"] => InputHoldOk(B);
                ["MESS-CANCEL"] => InputMessCancel(B);
                ["XMIT"] => InputXmit(B);
                ["ALARM"] => InputAlarm(B);
                ["ALARM"] => InputAlarmOk(B);
                ["ALARM-REJ"] => InputAlarmRej(B);
                ["CONN-OPEN"] => InputConnOpen(B);
                ["CONN-CLOSE"] => InputConnClose(B);
                ["CONN-REJ"] => InputConnRej(B);
                TRUE => ProtocolError(B);
              END;
            END;
          END;
        END;

^;
```

```
!
!       The following routines parse protocol items and convert them to the
!        appropriate internal data structures.  They verify lengths, generic
!        codes and incarnation numbers, and supply host information.
!;

InputMess <-
    !
    !       InputMess, in addition to the usual input functions:
    !        produces a MessHandle,
    !        converts the NoQWait field,
    !        verifies specific/generic consistency,
    !        and moves the message text (via InputMessText) from the
    !         NetBuffer into the MessHandle.  If InputMessText cannot
    !         allocate storage for the text, it is left null.
    !
    ! */
    EXPR(B:NetBuffer)
      BEGIN
        DECL MB:MessBlock;
        MB <¦¦ MessFormat;
        MB.QWait <- NOT ¦ NoQWait;
        B.Scan # ¦ FirstByte => ProtocolError(B);
        EncodeGenericClass(MB.SourceProcess.GenericName) #>
          ProtocolError(B);
        EncodeGenericClass(MB.DestProcess.GenericName) #>
          ProtocolError(B);
        MB.SourceProcess.Host <- B.Host;
        MB.DestProcess.Host <- LocalHost;
        DECL MH:MessHandle LIKE
          Allocate(MessHandle, MB)      /* 'implicit Seize';
        MH.DestProcess.Incarnation # LocalIncarnation =>
          BEGIN
            SendMESS\REJ(MH,
                         "Bad incarnation number on destination process");
            Release(MH);
          END;
        BEGIN
          Null(MH.DestProcess.Instance) =>
            NOT MH.IsGeneric OR MH.IsSequenced OR MH.IsMarked;
          MH.IsGeneric OR QWait;
        END =>
          BEGIN
            SendMESS\REJ(MH,
                         "Destination name/handling - generic/specific
mismatch");
            Release(MH);
          END;
        MH.TextLength <- ¦ Length - ¦ FirstByte;
        InputMessText(B, MH);
        EnQInputMess(MH);
        Free(MH);
      END;
```

```
InputMessOk <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL MB:MessBlock;
      MB <|| MessOkFormat;
      FormatError(B, MB) => NOTHING;
      MB.SourceProcess.Host <- LocalHost;
      MB.DestProcess.Host <- B.Host;
      RecordMESS\OK(MB);
    END;

InputMessRej <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL MB:MessBlock;
      MB <|| MessRejFormat;
      FormatError(B, MB) => NOTHING;
      MB.SourceProcess.Host <- LocalHost;
      MB.DestProcess.Host <- B.Host;
      RecordMESS\REJ(MB);
    END;

InputMessHold <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL MB:MessBlock;
      MB <|| MessHoldFormat;
      FormatError(B, MB) => NOTHING;
      MB.SourceProcess.Host <- LocalHost;
      MB.DestProcess.Host <- B.Host;
      RecordMESS\HOLD(MB);
    END;

InputHoldOk <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL MB:MessBlock;
      MB <|| HoldOkFormat;
      FormatError(B, MB) => NOTHING;
      MB.SourceProcess.Host <- B.Host;
      MB.DestProcess.Host <- LocalHost;
      RecordHOLD\OK(MB);
    END;
```

```
InputMessCancel <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL MB:MessBlock;
      MB <¦¦ MessCancelFormat;
      FormatError(B, MB) => NOTHING;
      MB.SourceProcess.Host <- B.Host;
      MB.DestProcess.Host <- LocalHost;
      RecordMESS\CANCEL(MB);
    END;

InputXmit <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL MB:MessBlock;
      MB <¦¦ XmitFormat;
      FormatError(B, MB) => NOTHING;
      MB.SourceProcess.Host <- LocalHost;
      MB.DestProcess.Host <- B.Host;
      RecordXMIT(MB);
    END;

InputAlarm <-
    !
    !     InputAlarm, in addition to the usual input functions:
    !      produces an AlarmHandle.
    !
    ! */
      EXPR(B:NetBuffer)
        BEGIN
          DECL AB:AlarmBlock;
          AB <¦¦ AlarmFormat;
          LengthError(B) => ProtocolError(B);
          EncodeGenericClass(AB.SourceProcess.GenericName) #>
            ProtocolError(B);
          EncodeGenericClass(AB.DestProcess.GenericName) #>
            ProtocolError(B);
          AB.SourceProcess.Host <- B.Host;
          AB.DestProcess.Host <- LocalHost;
          DECL AH:AlarmHandle LIKE
            Allocate(AlarmHandle, AB)     /* 'implicit Seize';
          AH.DestProcess.Incarnation # LocalIncarnation =>
            BEGIN
              SendALARM\REJ(AH,
                          "Bad incarnation number on destination process");
              Free(AH);
            END;
          EnQInputAlarm(AH);
          Free(AH);
        END;
```

```
InputAlarmOk <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL AB:AlarmBlock;
      AB <¦¦ AlarmOkFormat;
      FormatError(B, AB) => NOTHING;
      AB.SourceProcess.Host <- LocalHost;
      AB.DestProcess.Host <- B.Host;
      RecordALARM\OK(AB);
    END;

InputAlarmRej <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL AB:AlarmBlock;
      AB <¦¦ AlarmRejFormat;
      FormatError(B, AB) => NOTHING;
      AB.SourceProcess.Host <- LocalHost;
      AB.DestProcess.Host <- B.Host;
      RecordALARM\REJ(AB);
    END;

InputConnOpen <-
  '
  !     InputConnOpen, in addition to the usual input functions:
  !      produces a ConnHandle,
  !      converts MSG connection types to internal ones,
  !      verifies connection type (one and only one),
  !      and verifies bytesize range.
  !
  ' */
    EXPR(B:NetBuffer)
      BEGIN
        DECL Duplex, Send, Receive, ServerTelnet, UserTelnet:BOOL;
        DECL CB:ConnBlock;
        CB <¦¦ ConnOpenFormat;
        Duplex <- ¦ Type\duplex;
        Send <- ¦ Type\send;
        Receive <- ¦ Type\receive;
        ServerTelnet <- ¦ Type\ServerTELNET;
        UserTelnet <- ¦ Type\UserTELNET;
        C.ConnType <-
          BEGIN
            UserTelnet => "UserTELNET";
            ServerTelnet => "ServerTELNET";
            "Binary";
          END;
        C.ConnDirection <-
          [) Send => "Out"; Receive => "In"; "InOut" (];
        LengthError(B) => ProtocolError(B);
        EncodeGenericClass(CB.SourceProcess.GenericName) #>
          ProtocolError(B);
        EncodeGenericClass(CB.DestProcess.GenericName) #>
          ProtocolError(B);
```

```
        CB.SourceProcess.Host <- B.Host;
        CB.DestProcess.Host <- LocalHost;
        DECL CH:ConnHandle SHARED
          Allocate(ConnHandle, CB)       /* 'implicit Seize';
        CH.DestProcess.Incarnation # LocalIncarnation =>
          BEGIN
            SendCONN\REJ(CH,
                        "Bad incarnation number on destination process");
            Free(CH);
          END;
        [) Duplex => 1; 0 (] + [) Send => 1; 0 (] +
          [) Receive => 1; 0 (] + [) ServerTelnet => 1; 0 (] +
          [) UserTelnet => 1; 0 (] # 1 =>
          BEGIN
            SendCONN\REJ(CH, "Invalid connection type");
            Free(CH);
          END;
        CH.ConnBytesize GT MaximumConnBytesize =>
          BEGIN
            SendCONN\REJ(CH, "Invalid connection byte size");
            Free(CH);
          END;
        EnQInputOpenConn(CH);
        Free(CH);
      END;

InputConnClose <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL CB:ConnBlock;
      CB <|| ConnCloseFormat;
      LengthError(B) => ProtocolError(B);
      EncodeGenericClass(CB.SourceProcess.GenericName) #>
        ProtocolError(B);
      EncodeGenericClass(CB.DestProcess.GenericName) #>
        ProtocolError(B);
      CB.SourceProcess.Host <- B.Host;
      CB.DestProcess.Host <- LocalHost;
      DECL CH:ConnHandle LIKE
        Allocate(ConnHandle, CB)         /* 'Implicit Seize';
      CH.DestProcess.Incarnation # LocalIncarnatior =>
        BEGIN
          SendCONN\REJ(CH,
                      "Bad incarnation number on destination process");
          Free(CH);
        END;
      EnQInputCloseConn(CH);
    END;
```

```
InputConnRej <-
  EXPR(B:NetBuffer)
    BEGIN
      DECL CB:ConnBlock;
      CB <|| ConnRejFormat;
      FormatError(B, CB) => NOTHING;
      CB.SourceProcess.Host <- B.Host;
      CB.DestProcess.Host <- LocalHost;
      RecordCONN\REJ(CB);
    END;

FormatError <-
   '
   !     FormatError verifies lengths, generic codes and incarnation
   !      numbers.
   !
   ' */
    EXPR(B:NetBuffer, TB:TransactionBlock; BOOL)
      BEGIN
        LengthError(B) => [] ProtocolError(B); FALSE [];
        EncodeGenericClass(TB.SourceProcess.GenericName) #>
          [] ProtocolError(B); FALSE [];
        EncodeGenericClass(TB.DestProcess.GenericName) #>
          [] ProtocolError(B); FALSE [];
        TB.DestProcess.Incarnation # LocalIncarnation => FALSE;
      END;

LengthError <-
   '
   !     Length Errors result from discrepancies between the official
   !      count sent by MSG and actual data parsed.  If the official
   !      count is long then B.Scan will be less than B.Read.  If it
   !      is short B.Scan will be greater than B.Read, the parsing
   !      routines advancing B.Scan past B.Read without reading.
   !
   ' */ EXPR(B:NetBuffer; BOOL) B.Scan # B.Read;

ProtocolError <-
   '
!        ProtocolError arranges the sending of a PtclErr item based on the
!         NetBuffer passed to it.';

^;
```

```
'
!------------------------------------------------------------------------------
!
!                            Protocol output routines
!
!------------------------------------------------------------------------------
';

ProtocolOutput <-
  '
  !     ProtocolOutput waits until it is SIGNALed, then loops through
  !      its delivery queue, removes entries and dispatches to the
  !      appropriate output routine.  The entries are then dequeued.
  !
  ' */
    EXPR(Ego:ServerHandle)
      BEGIN
        DECL B:NetBuffer LIKE AllocateNetBuffer();
        DECL QEntry:TransactionHandle;
        Seize(Ego.DeliveryQ);
        REPEAT
          Null(Ego.DeliveryQ) ->
            BEGIN
              Ego.Running <- FALSE;
              Release(Ego.DeliveryQ);
              WAIT({ Ego.WakeUpSignal });
              Seize(Ego.DeliveryQ);
            END;
          BEGIN
            Null(Ego.DeliveryQ) => NOTHING;
            QEntry <- Front(Ego.DeliveryQ);
            DECL IsSeized:BOOL LIKE TestSeize(QEntry);
            Release(Ego.DeliveryQ);
            NOT IsSeized =>
              [] Pause(SeizeWait); Seize(Ego.DeliveryQ) [];
            CASE[QEntry.ProtocolCommand]
              ["MESS"] => OutputMess(QEntry, B);
              ["MESS-OK"] => OutputMessOk(QEntry, B);
              ["MESS-REJ"] => OutputMessRej(QEntry, B);
              ["MESS-HOLD"] => OutputMessHold(QEntry, B);
              ["HOLD-OK"] => OutputHoldOk(QEntry, B);
              ["MESS-CANCEL"] => OutputMessCancel(QEntry, B);
              ["XMIT"] => OutputXmit(QEntry, B);
              ["ALARM"] => OutputAlarm(QEntry, B);
              ["ALARM-OK"] => OutputAlarmOk(QEntry, B);
              ["ALARM-REJ"] => OutputAlarmRej(QEntry, B);
              ["CONN-OPEN"] => OutputConnOpen(QEntry, B);
              ["CONN-CLOSE"] => OutputConnClose(QEntry, B);
              ["CONN-REJ"] => OutputConnRej(QEntry, B);
              ["Start ICP"], ["Finish ICP"] =>
                NewHost(Release(QEntry), B);
              TRUE => MSGError('Bad protocol command');
            END;
            QEntry.ProtocolCommand = "Start ICP" OR
```

```
|                 QEntry.ProtocolCommand = "Finish ICP" => NOTHING;
|               || Header;
|               | Length <- B.Span;
|               DECL HH:HostHandle LIKE SeizeHost(B.Host);
|               B.Channel <- First(HH.ChannelS);
|               NetSendItem(B) #>
|                 BEGIN
|                   DECL LocalSocket:Socket LIKE
|                     GetSocketFromChannel(B.Channel);
|                   CHCLOSE(B.Channel);
|                   DeassignMSGOutputSocket(LocalSocket);
|                   Remove(HH.ChannelS, B.Channel);
|                   Null(HH.ChannelS) =>
|                     BEGIN
|                       DECL Restart:ContactHandle LIKE
|                         Allocate(ContactHandle);
|                       Seize(Ego.DeliveryQ);
|                       Restart.ProtocolCommand <- "Start ICP";
|                       Restart.RemoteHost <- HH;
|                       EnQAtFront(Ego.DeliveryQ, Restart);
|                       Release(Ego.DeliveryQ);
|                       Release(Restart);
|                       Release(HH);
|                     END;
|                   Release(HH);
|                 END;
|               Release(HH);
|               QEntry.ProtocolCommand <- NullProtocolCommand;
                Seize(Ego.DeliveryQ);
                DeQ(Ego.DeliveryQ, QEntry);
                Free(QEntry);
              END;
            END;
          END;

| ^;
```

```
'
!       The following routines convert internal data structures to
!        protocol items and determine the target host.
';

OutputMess <-
    '
    !     OutputMess, in addition to the usual output functions:
    !      converts the NoQWait field,
    !      sets the FirstByte field,
    !      and moves the message text from the MessHandle to the
    !       NetBuffer (via OutputMessText).
    !
    ' */
    EXPR(M:MessHandle, B:NetBuffer SHARED)
      BEGIN
        M ¦¦> MessFormat;
        ¦ Command <- "MESS";
        ¦ NoQWait <- NOT M.QWait;
        ¦ FirstByte <- B.Scan;
        B.Host <- M.DestProcess.Host;
        OutputMessText(B, MH);
      END;

OutputMessOk <-
  EXPR(M:MessHandle, B:NetBuffer SHARED)
    BEGIN
      M ¦¦> MessOkFormat;
      ¦ Command <- "MESS-OK";
      B.Host <- M.SourceProcess.Host;
    END;

OutputMessRej <-
  EXPR(M:MessHandle, B:NetBuffer SHARED)
    BEGIN
      M ¦¦> MessRejFormat;
      ¦ Command <- "MESS-REJ";
      B.Host <- M.SourceProcess.Host;
    END;

OutputMessHold <-
  EXPR(M:MessHandle, B:NetBuffer SHARED)
    BEGIN
      M ¦¦> MessHoldFormat;
      ¦ Command <- "MESS-HOLD";
      B.Host <- M.SourceProcess.Host;
    END;
```

```
OutputHoldOk <-
  EXPR(M:MessHandle, B:NetBuffer SHARED)
    BEGIN
      M ¦¦> HoldOkFormat;
      ¦ Command <- "HOLD-OK";
      B.Host <- M.DestProcess.Host;
    END;

OutputMessCancel <-
  EXPR(M:MessHandle, B:NetBuffer SHARED)
    BEGIN
      M ¦¦> MessCancelFormat;
      ¦ Command <- "MESS-CANCEL";
      B.Host <- M.DestProcess.Host;
    END;

OutputXmit <-
  EXPR(M:MessHandle, B:NetBuffer SHARED)
    BEGIN
      M ¦¦> XmitFormat;
      ¦ Command <- "XMIT";
      B.Host <- M.SourceProcess.Host;
    END;

OutputAlarm <-
  EXPR(A:AlarmHandle, B:NetBuffer SHARED)
    BEGIN
      A ¦¦> AlarmFormat;
      ¦ Command <- "ALARM";
      B.Host <- A.DestProcess.Host;
    END;

OutputAlarmOk <-
  EXPR(A:AlarmHandle, B:NetBuffer SHARED)
    BEGIN
      A ¦¦> AlarmOkFormat;
      ¦ Command <- "ALARM-OK";
      B.Host <- A.SourceProcess.Host;
    END;

OutputAlarmRej <-
  EXPR(A:AlarmHandle, B:NetBuffer SHARED)
    BEGIN
      A ¦¦> AlarmRejFormat;
      ¦ Command <- "ALARM-REJ";
      B.Host <- A.SourceProcess.Host;
    END;
```

```
OutputConnOpen <-
   '
   !     OutputConnOpen, in addition to the usual output functions:
   !      converts internal connection types to MSG ones.
   !
   ' */
     EXPR(C:ConnHandle, B:NetBuffer SHARED)
       BEGIN
         C ||> ConnOpenFormat;
         | Type\duplex <-
           C.ConnType = "Binary" AND C.ConnDirection = "InOut";
         | Type\send <- C.ConnDirection = "Out";
         | Type\receive <- C.ConnDirection = "In";
         | Type\ServerTELNET <- C.ConnType = "ServerTELNET";
         | Type\UserTELNET <- C.ConnType = "UserTELNET";
         | Command <- "CONN-OPEN";
         B.Host <- C.DestProcess.Host;
       END;

OutputConnClose <-
  EXPR(C:ConnHandle, B:NetBuffer SHARED)
    BEGIN
      C ||> ConnCloseFormat;
      | Command <- "CONN-CLOSE";
      B.Host <- C.DestProcess.Host;
    END;

OutputConnRej <-
  EXPR(C:ConnHandle, B:NetBuffer SHARED)
    BEGIN
      C ||> ConnRejFormat;
      | Command <- "CONN-REJ";
      B.Host <- C.DestProcess.Host;
    END;

^;
```

```
'
!-----------------------------------------------------------------------
!
!                         ICP routines
!
!-----------------------------------------------------------------------
';

'
!       Receive ICP request
!               OPEN ICP CHANNEL
!               GET HOST AND SOCKET FROM CHANNEL HANDLE
!               AUTHENTICATE -- STOP IF FAILURE
!               GET LOCAL SOCKET FROM SYSTEM
!               SEND LOCAL SOCKET
!               CLOSE ICP CHANNEL
!               OPEN REMOTE CHANNEL(S) ON LOCAL SOCKET
!               RECEIVE SYNCH
!               SEND SYNCH
!               KILL OLD TRAFFIC
!
!       Ser  ICP request
!               GET LOCAL SOCKET FROM SYSTEM
!               OPEN ICP CHANNEL ON LOCAL SOCKET -- STOP IF FAILURE
!               (HANDLE AUTHENTICATION)
!               RECEIVE REMOTE SOCKET
!               CLOSE ICP CHANNEL
!               OPEN REMOTE CHANNEL(S) ON LOCAL SOCKET
!               SEND SYNCH
!               RECEIVE SYNCH
!               KILL OLD TRAFFIC';

ICPHandler <-
   '
   !    ICPHandler is the main routine of the ICP handling path.  It
   !     waits until another entity initiates contact, verifies that
   !     entity as an MSG, establishes the remote and local sockets to
   !     be used in the permanent connection, creates host and servers
   !     if nonexistent, and notifies the network output server that a
   !     connection should be opened.
   !
   ' */
    EXPR()
      REPEAT
        BEGIN
          DECL ICPChannel:ChannelHandle LIKE
            CHOPEN(MakeId(0, 0, ICPSocket), "Out", "Binary", 32)
                                /* 'This blocks until someone connects.';
          DECL RemoteHost:HostCode LIKE
            GetHostFromChannel(ICPChannel);
          DECL RemoteSocket:Socket LIKE
            GetSocketFromChannel(ICPChannel);
          Authenticate(RemoteHost, RemoteSocket) #>
            CHCLOSE(ICPChannel);
```

```
        DECL LocalSocket:Socket LIKE AssignMSGSocket();
        SEND(LocalSocket, ICPChannel);
        CHCLOSE(ICPChannel);
        DECL RemoteHandle:HostHandle LIKE
          SeizeHost(RemoteHost);
        Null(RemoteHandle) ->
          BEGIN
            RemoteHandle <-
              Allocate(HostHandle)      /* 'implicit seize';
            RemoteHandle.Host <- RemoteHost;
            RemoteHandle.InputServer <-
              AllocateNetInputServer(RemoteHost);
            RemoteHandle.DeliveryServer <-
              AllocateNetOutputServer(RemoteHost);
          END;
        DECL FinishContact:ContactHandle LIKE
          Allocate(ContactHandle);
        FinishContact.RemoteHost <- RemoteHandle;
        FinishContact.RemoteSocket <-
          RemoteSocket + 2     /* 'NewHost uses RemoteSocket as input socket';
        FinishContact.LocalSocket <-
          LocalSocket - 2      /* 'NewHost uses LocalSocket+2 as input socket';
        FinishContact.ProtocolCommand <- "Finish ICP";
        Deliver(FinishContact, RemoteHandle);
        Release(FinishContact);
        Release(RemoteHandle);
      END;
    END;

Authenticate <-
  '
!       Authenticate verifies that the entity initiating an ICP is indeed an
!       MSG.  It checks that the remote socket used in the ICP is a valid
!       MSG socket by contacting the remote MSG through its authentication
!       socket and receiving the range or list of that MSGs sockets.';

^;
```

```
SeizeHostHandle <-
   '
   !    SeizeHostHandle is called from the user call server.  It not
   !     only seizes the handle of the host code passed to it, but
   !     also initiates contact with that host if it is unknown.
   !     If the host is new the routine creates host and servers and
   !     notifies the network output server that a connection should
   !     be opened.
   !
   ' */
    EXPR(RemoteHost:HostCode; HostHandle)
      BEGIN
        DECL RemoteHandle:HostHandle LIKE SeizeHost(RemoteHost);
        NOT Null(RemoteHandle) => RemoteHandle;
        RemoteHandle <-
          Allocate(HostHandle)           /* 'implicit seize';
        RemoteHandle.Host <- RemoteHost;
        RemoteHandle.InputServer <-
          AllocateNetInputServer(RemoteHost);
        RemoteHandle.DeliveryServer <-
          AllocateNetOutputServer(RemoteHost);
        DECL StartContact:ContactHandle LIKE
          Allocate(ContactHandle);
        StartContact.RemoteHost <- RemoteHandle;
        StartContact.ProtocolCommand <- "Start ICP";
        Deliver(StartContact, RemoteHandle);
        Release(StartContact);
        RemoteHandle;
      END;

SeizeHost <-
   '
   !    SeizeHost finds (in HostS) and seizes the host handle of the
   !     host code passed to it.  If no handle is found, a null handle
   !     is returned.
   !
   ' */
    EXPR(HC:HostCode; HostHandle)
      FOREACH HH IN HostS
        DO HH.Host = HC => Seize(HH); NullHostHandle END;
```

```
NewHost <-
   '
   !     NewHost establishes a permanent connection with a remote MSG.
   !      It is called from ProtocolOutput.  If the desire for the
   !      connection began locally an ICP request is made on the remote
   !      MSG.  The input and output connections are opened and
   !      inserted in the proper channel sets, synchronization
   !      information is exchanged, transactions addressed to a prior
   !      incarnation of the remote MSG are cancelled, and the input
   !      server is signalled that a new connection exists.  If the
   !      connections cannot be opened, HostDead action is performed.
   !
   ' */
  EXPR(Contact:ContactHandle, B:NetBuffer)
    BEGIN
      Contact.ProtocolCommand = "Start ICP" ->
        RequestICP(Contact);
      Null(RemoteSocket) => HostDied(Contact);
      DECL InputChannel:ChannelHandle LIKE
        CHOPEN(MakeId(Contact.RemoteHost.Host,
                      Contact.RemoteSocket + 1,
                      Contact.LocalSocket + 2), "In", "Binary",
               8);
      Null(InputChannel) => HostDied(Contact);
      DECL OutputChannel:ChannelHandle LIKE
        CHOPEN(MakeId(Contact.RemoteHost.Host,
                      Contact.RemoteSocket,
                      Contact.LocalSocket + 3), "Out",
               "Binary", 8);
      Null(OutputChannel) =>
        [) CHCLOSE(InputChannel); HostDied(Contact) (];
      Synchronize(Contact, InputChannel, OutputChannel, B) #>
        BEGIN
          CHCLOSE(InputChannel);
          CHCLOSE(OutputChannel);
          HostDied(Contact);
        END;
      DECL RemoteHost:HostHandle LIKE
        Seize(Contact.RemoteHost);
      Insert(RemoteHost.ConnectionS, OutputChannel);
      DECL InputServer:ServerHandle LIKE
        Seize(RemoteHost.InputServer);
      Insert(InputServer.ChannelS, InputChannel);
      Release(InputServer);
      Release(RemoteHost);
      DECL Q:Queue(TransactionHandle) LIKE
        RemoteHost.DeliveryServer.DeliveryQ;
      Seize(Q);
      Seize(Contact);
      DeQ(Q, Contact);
      Free(Contact);
      FOREACH T IN Q
        DO
          DECL D:UNION(ProcessName, HostHandle) LIKE
```

```
                RemoteDest(T);
              D.Host = RemoteHost.Host AND
                D.Incarnation # RemoteHost.Incarnation ->
                REPEAT
                  DECL Success:BOOL LIKE
                    BEGIN
                      TestSeize(T) => HostDeadTransaction(T);
                      FALSE;
                    END;
                  Success => NOTHING;
                  Release(Q);
                  Pause(SeizeWait);
                  Seize(Q);
                END;
          END;
        Release(Q);
        SIGNAL(InputServer.WakeUpSignal);
      END;

RequestICP <-
   !
   !     RequestICP, called from NewHost, establishes the local and
   !      remote sockets to be used in a network connection with a
   !      remote host in response to the desire of a user to contact
   !      a process on that host.  The remote socket is obtained from
   !      the ICP handler of the host.  If the host is down the remote
   !      socket is left null.
   !
   ! */
    EXPR(StartContact:ContactHandle)
      BEGIN
        StartContact.LocalSocket <- AssignMSGSocket();
        DECL ICPChannel:ChannelHandle LIKE
          CHOPEN(MakeId(StartContact.RemoteHost.Host, ICPSocket,
                        StartContact.LocalSocket), "In",
                 "Binary", 32);
        Null(ICPChannel) => NOTHING;
        'AuthenticationHandler is consulted now by remote MSG';
        Abnormal(RECEIVE(StartContact.RemoteSocket, ICPChannel)) =>
          NOTHING;
        CHCLOSE(ICPChannel);
      END;
```

```
HostDied <-

      HostDied cleans up when a remote host proves unreachable.  The
        local socket assigned to the aborted connection is freed and,
        if there are no other paths to the host, transactions
        addressed to it are cancelled.

    */
    EXPR(C:ContactHandle)
      BEGIN
        DeassignMSGSocket(C.LocalSocket);
        NOT Null(C.RemoteHost.ChannelS) => NOTHING;
        DECL Q:Queue(TransactionHandle) LIKE
          C.RemoteHost.DeliveryServer.DeliveryQ;
        Seize(Q);
        FOREACH T IN Q
          DO
            BEGIN
              T.ProtocolCommand = "Start ICP" OR
                T.ProtocolCommand = "Finish ICP" =>
                BEGIN
                  T.RemoteHost = C.RemoteHost =>
                    [] Seize(T); DeQ(Q, T); Free(T) [];
                END;
              RemoteDest(T).Host = C.RemoteHost.Host ->
                REPEAT
                  DECL Success:BOOL LIKE
                    BEGIN
                      TestSeize(T) => HostDeadTransaction(T);
                      FALSE;
                    END;
                  Success => [] DeQ(Q, T); Free(T) [];
                  Release(Q);
                  Pause(SeizeWait);
                  Seize(Q);
                END;
            END;
          END;
        Release(Q);
      END;
```

```
RemoteDest <-
  '
  !     RemoteDest returns the process name of the process to which
  !      the given transaction is addressed.  The name contains the
  !      host and incarnation number.
  !
  ' */
    EXPR(T:TransactionHandle; ProcessName)
      CASE[T.ProtocolCommand]
        ["Start ICP"], ["Finish ICP"] => NullProcessName;
        ["MESS-OK"],
          ["MESS-REJ"],
          ["MESS-HOLD"],
          ["XMIT"],
          ["ALARM-OK"],
          ["ALARM-REJ"] => T.SourceProcess;
        TRUE => T.DestProcess;
      END;

Synchronize <-
  '
  !     Synchronize exchanges SYNCH information with the remote MSG,
  !      using the channels opened by NewHost.  The incarnation number
  !      of the remote host is obtained and stored in its host handle.
  ' */
    EXPR(CH:ContactHandle,
         IC:ChannelHandle,
         OC:ChannelHandle,
         B:NetBuffer;
         BOOL)
      BEGIN
        DECL Successful:BOOL;
        ¦¦ SynchFormat;
        B.Host <- CH.RemoteHost.Host;
        FOR I TO 2
          REPEAT
            BEGIN
              CH.ProtocolCommand = "Start ICP" AND I = 1 OR
                CH.ProtocolCommand = "Finish ICP" AND I = 2 =>
                BEGIN
                  ¦ Command <- "SYNCH";
                  ¦ MyIncarnation <- MSGIncarnation;
                  ¦ YourIncarnation <-
                    [) I = 1 => 0; CH.RemoteHost.Incarnation (];
                  ¦ Version <- MSGVersion;
                  ¦ Length <- B.Scan;
                  B.Channel <- OC;
                  Successful <- NetSendItem(B);
                END;
              B.Channel <- IC;
              DECL Result:ReasonCode LIKE NetReceiveItem(B);
              Successful <-
                Result # "Normal" OR ¦ Command # "SYNCH" OR
                  ¦ Version # MSGVERSION #> NOTHING;
```

```
AuthenticationHandler <-
   '
!      AuthenticationHandler sends out MSG socket verification
!       information in response to requests for it.  Such requests
!       are stimulated by RequestICP.
!
' */
    EXPR()
      REPEAT
        DECL AuthenticationChannel:ChannelHandl  LIKE
          CHOPEN(MakeId(0, 0, AuthenticationSo
                 "Out",
                 "Binary",
                 32);
        SEND(MSGSocketRange, AuthenticationChannel);
        CHCLOSE(AuthenticationChannel);
      END;
```

REFERENCES

[Belady] Belady, L. A., and M. M. Lehman. "A model of large program development," IBM Systems Jl, vol. 15, no. 3, pp. 225-252, 1976.

[Manual] ECL programmer's manual. Technical report TR 23-74, Ctr. for Res. in Computing Technology, Harvard Univ., December 1974.

[Spec] MSG design specification, in "Third semi-annual technical report for the National Software Works." Massachusetts Computer Associates, Wakefield, Mass., February 1977.

Index To Model Entities