THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE. THIS IS AS RECEIVED FROM CUSTOMER.
STERLING: A PEDAGOGICAL IMPLEMENTATION

OF THE

ISO MODEL FOR OPEN SYSTEM INTERCONNECTION

by

Ronald Curtis Albury

B.S. Rochester Institute of Technology 1976

------------------------

A MASTER'S REPORT

Submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983

Approved by:

[Vincent Walbertone
Major Professor]
# Table of Contents

Preface ........................................................................ iv

Chapter 0  
0.0 Motivations for Networks ......................... 1  
0.1 An Approach to the Study of Networks .... 2

Chapter 1  
1.0 Concurrent Pascal ................................. 7  
1.1 Mailboxes ........................................... 8  
1.2 The Design of Sterling ......................... 11

Chapter 2  
2.0 NET0: Interprocess Communication ......... 14  
2.0 NET1: Communication With A Remote Site .. 19  
2.2 NET2: Presentation Layer ....................... 27  
2.3 NET3: Session Layer ............................ 35  
2.4 NET4: Transport Layer ......................... 45

Further Work .................................................. 58

References ....................................................... 60

Appendix A  
Include Modules ........................................ A1

Appendix B  
Sterling's Protocols ........................................ B1
# LIST OF FIGURES

1. The ISO Network Model ........................................ 6  
2. Mailbox Communication ......................................... 10  
3. 7 Layer Access Diagram ........................................... 12  
4. NET0 Access Diagram .............................................. 16  
5. NET1 Access Diagram ............................................... 23  
6. Logical View of Blackbox .......................................... 24  
7. NET2 Access Diagram ............................................... 30  
8. Vigenere Cipher .................................................... 31  
9. NET3 Access Diagram ............................................... 38  
10. NET3 Session Protocol ............................................ 39  
11. Session Layer Finite State Automaton ......................... 40  
12. NET4 Access Diagram ............................................... 49  
13. NET4 Topology ..................................................... 50  
14. NET4 Connection Protocol ......................................... 51  
15. Transport Layer Finite State Automaton ....................... 52
ACKNOWLEDGEMENTS

This report is dedicated to my father, in fulfillment of a birthday pledge I made to him over twenty years ago. I would like to thank my wife Suzette and my son Sterling for their support during this project, my brother Randy for being such a good role model, and my advisor Dr. Wallentine for refusing to let me do anything less than my best.
PREFACE

In the spring of 1982 I completed course CS-725, Computer Networks, with the assigned text 'COMPUTER NETWORKS' by Andrew Tanenbaum. I found the text to be clear and easily understood, but suffering from several problems.

The Tanenbaum text is built around a discussion of the ISO (International Standards Organization) Reference Model for computer networks. While I agree with the idea of using a central reference when discussing the design of the various networks currently in use, a problem arises because there are no networks that truly follow the ISO model. The author does give segments of Pascal code to illustrate portions of the model, but no comprehensive overview is given.

Another problem is the order in which the layers of the ISO model are presented. It is now generally considered good style to approach problem analysis and program design in a 'top down' manner. Tanenbaum chose, however, to start with the bottom layers of the model and work his way to the top. This results in students having to cope with such problems as error correcting protocols before they even have an understanding of the essential aspects of a computer network and the functions it is to provide.

I believe that Per Brinch Hansen has shown us an excellent way to teach a complex computer system with his SOLO operating system [solo]. He wrote a simplified but operational version of a single user operating system in a concurrent superset of Pascal.
He used a highly structured design to separate the various functions of the operating system and provided sufficient documentation to allow the students to modify the various components of the program.

I intend to correct those problems mentioned above by following Brinch Hansen's example and implementing a simple pedagogical network (STERLING) based on the ISO model. I will discuss the functions of the top four layers, design processes that perform simple subsets of those functions, and implement the designs in a language available at KSU. The report will not be an in-depth study of the ISO model, networking in general, or the language chosen for the implementation. Rather it should simply be viewed as a general workbook and source of assignments for a course in computer networks.

The goals of this project are as follows, to:

1. Provide a reference for teaching the ISO network reference model by implementing a simple version in a language available at KSU;
2. Provide a minimal subset of functions for the top four layers of the ISO model;
3. Design the layers for easy expansion and modification by students;
4. Allow for the function of each layer to be examined separately from other layers; and
5. Design for simplicity and clarity rather than efficiency and robustness.
0.0: Motivations For Networks

Computers were originally expensive, monolithic machines of limited capabilities. They were the nucleus of a small cluster of terminals, printers, and other peripheral devices. Because of the expense of the equipment, it was practical to devise methods for remote facilities to communicate with a central computer rather than purchasing additional equipment. This trend continued with the linking together of large facilities, making it easier to share both hardware and software resources. If one location's facilities became overloaded, part of the burden could be shifted to a remote site. Users could also access proprietary software developed at another location.

As the cost of equipment decreased it became cost effective to install small computers at those remote sites that had none. Data could be partially processed, and its volume greatly reduced, before transmission to a central location for final disposition. As computers became more specialized, small computers were introduced which served as 'front end machines', freeing the large machines from trivial duties and allowing them to concentrate their resources on those problems that they could best solve. Some modern applications of computer networks follow [1].

* Systems for corporate operations of many different types, e.g., order entry systems, centralized purchasing, distributed inventory control, insurance underwriting.
Corporate information networks, marketing information, customer information, product information.

Airline reservations, car rental, hotel booking,

Electronic mail and message sending, two-way interchange of messages.

Electronic transfer of financial transactions between banks and via checking clearing houses.

Consumer check and credit verification in stores and restaurants, and in some cases consumer electronic fund transfer; bank cash dispensers and customer terminals.

Intercorporate networks. For example, a computer in one firm transmits orders or invoices to another. Insurance agents have insurance company terminals, possibly via a shared network. Travel agents have terminals from airlines, shipping lines, hotel chains, etc.

Stock market information systems which permit searches for stocks that meet a certain criteria, performance comparisons, moving averages, and various forecasting techniques, all using dialogues which employ graphics.

Terminal systems for investment advice and management, tax preparation, tax minimization [sic].

Home information services (Such as Prestel [British Post Office], or any which use the home TV set)

0.1: An Approach To The Study Of Networks

Modern software engineering techniques stress the Top Down approach to understanding and solving problems. This paper's method of understanding computer networks will follow this approach of decomposing the problem into smaller modules, or layers. Starting with an application program, it will be determined what services are necessary for the layer to communicate with processes on other machines. A module will be added providing these services, then this module itself will be decomposed. This process will continue until we have reached the
underlying network itself. Each layer will be formalized and
tested before continuing, in order to insure the robustness of
the solution.

A basic principle of layering is to ensure independence of
each layer by defining the services provided by the layer,
regardless of how these services are performed. This layering
permits changes to be made in the way a layer or a set of layers
operates, provided that they still offer the same service to the
next higher layer [2].

This is the approach used by the International Standards
Organization (ISO) Subcommittee 97/16 in formulating the ISO
The members of this organization approached the layering of the
model using the following guidelines [3]:

P1: do not create so many layers as to make difficult the
    system engineering task describing and integrating these
    layers,

P2: create a boundary at a point where the services
description can be small and the number of interactions
across the boundary are minimized [sic],

P3: create separate layers to handle functions which are
    manifestly different in the process performed or the
technology involved,

P4: collect similar functions into the same layer,

P5: select boundaries at a point which past experience has
demonstrated to be successful,

P6: Create a layer of easily localized functions so that the
    layer could be totally redesigned and its protocols
    changed in a major way to take advantage of new advances
    in architectural, hardware or software technology without
    changing the services and interfaces with adjacent
    layers,
P7: create a boundary where it may be useful at some point in
time to have the corresponding interface standardized,

P8: create a layer when there is a need for a different level
of abstraction in the handling of data, e.g., morphology,
syntax, semantics,

P9: Enable changes of functions or protocols within a layer
without affecting the other layers,

P10: create for each layer interfaces with its upper and lower
layer only,

P11: create further subgrouping and organization of functions
to form sublayers within a layer in cases where distinct
communication services need it [sic],

P12: create, where needed, two or more sublayers with a
common, and therefore minimum, functionally [sic] to
allow interface operation with adjacent layers,

P13: Allow by-passing of sublayers,

Figure-1 depicts the ISO model with its seven layers. This
model provides the framework for this study. Layers one, two,
and three comprise a network over which data can be routed to
another computer. Standards for the lower three layers are the
most clearly defined because CCITT and the ISO and ANSI Data
Communication Technical Committees have been working on these
standards for many years [11]. Layers four, five, and six are
the means by which a host machine can access the network, and are
the primary emphases of this work.

STERLING is a series of five programs that demonstrates the
functions of the top layers of the ISO model. A minimal subset
of the layer under study is implemented at each stage of the
top-down decomposition. The programs are strictly pedagogical,
with the emphasis on underlying principles rather than clever
code. They are fully documented, and their modular design lends
itself to easy modification by students. Network services can be added, deleted, and modified with a minimum of difficulty.
I.S.O. NETWORK MODEL

Figure 1
1.0: Concurrent Pascal

Sterling is implemented in Concurrent Pascal (C-Pascal), a superset of standard Pascal developed by Per Brinch Hansen at the California Institute of Technology. C-Pascal is a highly structured language designed to allow the user to specify exactly what concurrent processes can do to shared variables, and to depend on the compiler to check that the programs satisfy these assumptions [5].

C-Pascal has several additional advantages which led to its choice as the implementation language. The original compiler, written in 1974 by Al Hartman, has been fully documented and published [16]. Also, C-Pascal is widely available in academia and is implemented at KSU on the Interdata 8/32 computer. It is a simple extension to standard Pascal, and can be understood easily by anyone with a working knowledge of structured programming languages. Additionally, it has proved successful in other pedagogical implementations of operating systems [5] and networks [4].

A major benefit of Concurrent Pascal is the ability of the programmer to divide the shared data structures of an operating system into small parts and define allowable operations on each of them. Processes perform concurrent operations using monitors to synchronize themselves and exchange data, and access private data structures by means of classes [5].

A Class is a privately owned procedure. It is initialized
once by its parent (a process or monitor), and after initialization its private data structures exist until the termination of its parent. Access rights to the class are owned by its parent. These rights can be passed to other classes owned by the same parent, but not to other processes or monitors.

Monitors, independently introduced by Hoare [6] and Brinch Hansen [7], refer to a shared procedure and its permanent data structures within a zone of mutual exclusion. Mutual exclusion is the mechanism which allows processes to acquire exclusive control of a resource for a finite period of time. Processes competing for the monitors thus gain access to, or control of, them in some sequential order [9]. Synchronization through monitors is enhanced by the two primitives DELAY and CONTINUE.

A process can be delayed in a monitor for any length of time by the process executing a DELAY. When a calling process is delayed in a monitor it loses its exclusive access to the monitor's variables until another process calls the same monitor and executes a CONTINUE. The process issuing the CONTINUE is then removed from the monitor, and the delayed process regains its exclusive access and resumes execution.

1.1: Mailboxes

In STERLING the C-Pascal primitives are used to implement another method of interprocess communication and synchronization: the Mailbox or Message Buffer. A mailbox can be viewed as a restricted monitor, where the only operations allowed on the shared data structure are the storage and retrieval of messages.
Mailboxes were chosen for several reasons. The primitives for the creation, use, and deletion of mailboxes already exist on many operating systems (Data General AOS, Digital VMS, Unix, etc). By strictly limiting the monitor's capabilities to communicate and synchronize, the entire function of a network layer can be viewed within a single process. The two entry points to a mailbox are analogous to performing 'reads' and 'writes', thus making them easy to understand. Finally the user processes are only loosely coupled to each other and therefore need not share common memory, making this method easily adaptable to distributed systems. (see figure 2)

Communication through a mailbox is subject to two resource constraints [7]:

1) the sender cannot exceed the finite capacity of the storage buffer; and
2) the receiver cannot consume messages faster than they are produced.

The messages should also be delivered in the same order that they are sent, without loss or modification of their content.

STERLING satisfies these constraints. If a sender tries to deposit a message in a full buffer, it is delayed until the receiver removes a message from the buffer. If a receiver tries to take a message from an empty buffer, the receiver is delayed until a sender has deposited a message into the buffer. Finally, the buffer is controlled on a strictly First-In-First-Out basis, with no operations performed on it but depositing and removing messages.
MAILBOX COMMUNICATION

PROCESS

MAIL

Figure 2
1.2: Design Of STERLING

Each of STERLING's five programs is designed as a group of independent processes communicating through mailboxes. Each process has a private mailbox through which it receives messages from one or more other processes. The processes may receive mail only through their private mailbox. As each layer of the ISO model is introduced a new process is added to each site of the simulated network representing the function of that layer. Figure 3 contains an illustration of how the communication paths for one node of the network would look with all seven layers of the model implemented.

A node of a network may have several application processes residing on it which access the network at the same time. While the code necessary to multi-thread STERLING's nodes would not be very complicated, the network was implemented with single-user nodes, with multi-threading left as a programming activity for the student.

As the messages are passed through the various layers on the way to the network, new information is added to allow the peer layers to communicate. It is also typical for some layers to break long messages into smaller packets before transmission. This breaking into packets conflicts somewhat with C-Pascal's rigid enforcement of compile-time strong typing. Experiments with variant records made the mailbox mechanism clumsy, and were abandoned in favor of a single fixed-length record (with some fields ignored) for all communication.

C-Pascal, as implemented at KSU, has extremely limited I/O
7 LAYER ACCESS DIAGRAM

Figure 3
capabilities. In order to provide STERLING with the file access it required, it was necessary to perform all input and output via supervisory calls to the operating system. Because of the limited capabilities of the programs, very little was required of these calls, and they were implemented in an easily understood Class procedure. The flag setting necessary for supervisory calls closely parallels the flag setting STERLING uses in communicating with the network, and should help the student to understand the principle of the hiding of low level interfaces within a high level language.

It becomes necessary at the Transport layer of this model for a process to be able to 'time out' if it receives no messages. This facility is not provided by the language, and had to be simulated by adding an additional process and monitor at each node of the network. A minor modification had to be made to the C-Pascal Kernel [10] to allow a process to call the 'Wait' primitive even if all other processes are in a delayed state.

Another problem that became most apparent during the implementation of the Transport layer was the inability of C-Pascal to allocate buffer space dynamically at run-time. Several solutions were examined and rejected because their complexity distracted from the general goals of the layer. The final implementation uses a simplified Transport protocol that does not require buffer management.
CHAPTER 2

2.0: NET0

In the simplest view, a network can be thought of as just the means for two or more processes to communicate. The processes must be sure that any messages sent are delivered to the correct destination in a timely manner, in the correct order, and with no loss of data integrity. These constraints hold no matter if the processes are on the same host computer or are on opposite sides of the earth. It is appropriate, therefore, that the study of networks begins by looking at a simple example of two processes exchanging messages without the use of any formal network structures.

A typical situation calling for interprocess communication is the management of a data base. In this situation it is the duty of one process to control all access to the information stored in the data base. This 'Server' process is the only process with access rights to the information. Any query from a 'Worker' process must be made by sending a request message to the Server. The Server can respond either by accessing the data base and transferring the requested information to the Worker, or by sending back a rejection message.

NET0 (see listing 0) is a simple demonstration of this type of interprocess communication. The Server process controls access to four files (FILE_A to FILE_D). The Worker process receives a request for a file from the terminal, then sends the request to the Server. If the Server is able to fill the request, it
transfers the file to the worker, which displays it on the terminal. If the request cannot be filled (i.e. the file does not exist), the server sends a rejection message to the worker. This sequence cycles until the program is terminated by issuing a break at the terminal.
NETO ACCESS DIAGRAM
(* *Cpascal Prefix*)
INCLUDE NETPFX

(* *******************************************************
  * PROGRAM NETO
  * Interprocess communication.
  *
  *******************************************************
  * Programmer: Ronald C. Albury
  * Date Written: June 1982
  * Computer: Interdata 8/32
  * Copyright 1982 by Ronald C. Albury
  *******************************************************
)

(* **Packet description*)
TYPE
  PACKET_TYPE = RECORD
    TEXT: MESSAGE_TYPE
  END;

(* **Constants for Mail_box_monitor*)
CONST
  MAX_MAIL = 4;
  MAX_SENDERS = 1;

(* **Constants for Resource*)
CONST
  MAX_RESOURCE_USERS = 1;

(* **Types and constants for Message_io_class*)
INCLUDE SVC1PFX

(* *Class to provide fixed record I/O*)
INCLUDE MSGIO

(* *Modified Brinch Hansen FIFO*)
INCLUDE FIFO

(* *Standard Brinch Hansen Resource*)
INCLUDE RESOURCE

(* *Interprocess communication mailbox*)
INCLUDE MAILBOX

(* *Worker application process*)
INCLUDE WORKERO

(* *Server application process*)
INCLUDE SERVERO

(* ** LISTING 0 **)
VAR
    CONSOLE: RESOURCE_MONITOR;
    WORKER_EVT, SERVER_EVT: MAIL_BOX_MONITOR;
    WORKER: WORKER_PROCESS;
    SERVER: SERVER_PROCESS;
BEGIN
    INIT
        CONSOLE,
        WORKER_EVT, SERVER_EVT,
        WORKER(CONSOLE, WORKER_EVT, SERVER_EVT),
        SERVER(SERVER_EVT, WORKER_EVT)
END.
2.1: NET1

A number of considerations must be introduced to the demonstration of interprocess communication if the Worker and Server processes are allowed to reside on different computers. Both machines must agree on certain protocols to insure the integrity of their communication. These protocols are analogous to the hierarchy of people necessary for executives of two companies to communicate.

The Application layer is the executive himself. It is the ultimate source and sink for all data transmitted across the network, and it is also the entry point for the application programs to interface with the rest of the network. In STERLING, these interfaces are provided through access rights which are doled out by the initial process. The following come under the domain of the application layer:

1) Identification of intended communication partners

2) Transfer of information

3) Synchronization of cooperating application processes.

The Presentation layer assumes the role of the assistant to the executive, who makes sure that all messages are in a form that the boss can understand. The following problems must be considered:

1) The machines may use different formats to store files (e.g. 80 byte fixed length records vs. 512 byte blocked variable length records).
2) The machines may use different character codes (e.g. ASCII vs. EBCDIC).

3) They may be exchanging secret information which should be encrypted before it is transmitted over a non-secure medium.

4) If large amounts of data are being transferred, it may be cost effective to use a compression algorithm to reduce the transmission costs.

The Session layer assumes the role of the executive's secretary, who makes the appointments, places the outgoing phone calls, screens the incoming calls, and takes messages. The following problems must be considered:

1) A process may need to 'log on' to a remote machine before it can communicate with any processes on that machine.

2) A process may have restricted access and require a password or some other form of authorization before it will agree to exchange messages.

3) The two processes must agree on such options as who can terminate the session and whether the communication will be half-duplex or full-duplex.

4) A process may be running, but not expecting messages, or already busy, and unable to receive new messages.

5) If a break in the network connection during mid-transaction would prove to be disastrous, it may be necessary to 'brace' or 'chain' together several messages into a single large message at the destination. This is
also known as "quarantining messages" [3].

The Transport layer assumes the role of the company switchboard operator, who manages all telephone connections coming into the business. The following problems must be considered:

1) The host must be able to establish and keep track of all of its connections across the network.
2) The host must be sure that the destination has sufficient buffer space to hold the messages that will be sent.
3) The host must be sure that all of the messages sent were received by the destination in the correct order.

Local Area Network: To complete the analogy there must be a telephone company to handle the actual transmission of the messages from one company to the other. The problems encountered in implementing a local area network [12] [13] [14], however, are beyond the scope of this report.

NET1 (see listing 1) assumes that the Worker and Server processes (the Application layers) are on separate hosts. A new process, the 'Black Box', is added to each host to simulate the protocols being in place to satisfy all of the previously mentioned problems.
The Worker and Server continue to act as if they are communicating directly with each other, while in fact there is another layer of processes that handles the details of the communication. In future chapters the Black Box will be subdivided into further layers, each addressing a specific type of protocol. Currently, the Black Box's only duty is to determine if a message is going into the host or coming out of it.
NET1 ACCESS DIAGRAM

Figure 5
(* *Cpascal prefix*)
INCLUDE NETPFX

(*****************************************************************************
 *
 * PROGRAM NET1
 * Interprocess communication between remote sites.
 *
 *****************************************************************************
 * Programmer: Ronald C. Albury
 * Date Written: July 1982
 * Computer: Interdata 8/32
 * Copyright 1982 by Ronald C. Albury
 *****************************************************************************)

(*NOTE: modifications to previous program are indicated *)
(* by (****) *)

(* **Packet description*)
TYPE
 DIRECTION_TYPE = (INCOMING, OUTGOING); (****)
 PACKET_TYPE = RECORD
   DIRECTION: DIRECTION_TYPE; (****)
   TEXT: MESSAGE_TYPE
 END;

(* **Constants for Mail_box_monitor*)
CONST
 MAX_MAIL = 4;
 MAX_SENDERS = 2; (****)

(* **Constants for Resource*)
CONST
 MAX_RESOURCE_USERS = 1;

(* **Types and constants for Message_io_class*)
INCLUDE SVC1PFX

(* **Class to provide fixed record I/O*)
INCLUDE MSGIO

(* Modified Brinch Hansen FIFO*)
INCLUDE FIFO

(* Standard Brinch Hansen RESOURCE*)
INCLUDE RESOURCE

(* Interprocess communication mailbox*)
INCLUDE MAILBOX

(*Undefined layers of the network*) (****)
INCLUDE BLACKBOX

(** LISTING 1 **)  (** 25 **)
INCLUDE WORKER1

INCLUDE SERVER1

TYPE

NODE_W = RECORD

APP: WORKER_PROCESS;
APP_EVT: MAIL_BOX_MONITOR;
BLACKBOX: BLACKBOX_PROCESS;
BB_EVT: MAIL_BOX_MONITOR
END;

NODE_S = RECORD

APP: SERVER_PROCESS;
APP_EVT: MAIL_BOX_MONITOR;
BLACKBOX: BLACKBOX_PROCESS;
BB_EVT: MAIL_BOX_MONITOR
END;

VAR

CONSOLE: RESOURCE_MONITOR;
WORKER: NODE_W;
SERVER: NODE_S;

BEGIN

INIT

CONSOLE,
SERVER.BB_EVT, WORKER.BB_EVT,
SERVER.APP_EVT, WORKER.APP_EVT,
SERVER.BLACKBOX(SERVER.APP_EVT, SERVER.BB_EVT,
WORKER.BB_EVT),
WORKER.BLACKBOX(WORKER.APP_EVT, WORKER.BB_EVT,
SERVER.BB_EVT),
SERVER.APP(SERVER.APP_EVT, SERVER.BB_EVT),
WORKER.APP(CONSOLE, WORKER.APP_EVT,
WORKER.BB_EVT)

END.

(* * LISTING 1 *)
2.2: NET2

The first category of protocol addressed is the Presentation Protocol. The presentation layer performs functions that are requested sufficiently often to warrant finding a general solution for them, rather than letting each user solve them [15]. These functions could be explicitly invoked by the user in the form of language enhancements or library routines called by the user, or be transparently invoked by the operating system/network interface.

When computers communicate over a network, they are vulnerable to unauthorized use of their transmissions. Sensitive information may be copied or altered on the way to its destination. In many applications (e.g. electronic funds transfer) it is imperative that this be prevented. Data encryption is the means used to assure the security of messages.

Another service which may be provided in the Presentation layer is text compression. Processing costs are currently decreasing much more rapidly than transmission costs. It will become increasingly economical to edit or compress the data so that a smaller number of bits is transmitted over the network. In addition to saving money, compression can reduce response times when the data components are long enough to take many seconds to transmit.

When incompatible machines are connected on a network, problems are introduced which must be corrected before those machines can communicate. The machines may use different character sets,
requiring only simple translation. When a program moves between machines with different instruction sets, the program will have to be recompiled. Some machines store their files as fixed length records and some as variable length records terminated by some combination of control characters.

Even on identical machines there may be incompatible terminals. The computer which a person wishes to access may not know the appropriate control codes to use with his terminal. The Presentation layer may serve as a hidden translator, allowing all application programs on the network to assume that they are communicating with a standard virtual terminal.

NET2 (see listing 2) provides only two presentation services: data encryption and the modification of record delimiters. The data encryption is accomplished through the Vignere cipher, a simple substitution cipher [15]. The encoded message is created by using the original text and the key as the row and column coordinates of a two dimensional array of characters. All nodes on the simulated network are currently initialized with the same key, with no provisions for modifying the key at run-time. The encryption routine is not transparent to the user and must be invoked by the Application Layer setting a flag in the network packet.

The record delimiter routine, on the other hand, is transparent to the user. On this simulated network there are only two kinds of machines: those whose record delimiter is a Carriage Return, and those who use a New Line. All packets automatically have a flag set before transmission indicating the delimiter used. The
destination presentation layer automatically converts any incompatible message to the format required by its host.
NET2 ACCESS DIAGRAM

Figure 7
<table>
<thead>
<tr>
<th>KEY</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>abcdefg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>bcdedefg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>cdefgghi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>defghhij</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>efghijkl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>fghijklm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>ghijklm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>hijklmn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TEXT:** DEAD  
**KEY:** FEED  
**CODE:** lieg

*Vigenere Cipher*

*Figure 8*
PROGRAM NET2

Application, Presentation, and Blackbox.

Programmer: Ronald C. Albury
Date Written: July 1982
Computer: Interdata 8/32
Copyright 1982 by Ronald C. Albury

(*NOTE: modifications to previous program are indicated *)
(* by (****) *)

(* **Packet description*)
TYPE
  DIRECTION_TYPE = (INCOMING, OUTGOING);
  SECURITY_TYPE = (SECRET, PUBLIC); (****)
  FILE_FORMAT_TYPE = (CR_DELIM, NL_DELIM); (****)
  PACKET_TYPE = RECORD
    SECURITY: SECURITY_TYPE; (****)
    FILE_FORMAT: FILE_FORMAT_TYPE; (****)
    DIRECTION: DIRECTION_TYPE;
    TEXT: MESSAGE_TYPE
END;

(* **Constants for Mail_box_monitor*)
CONST
  MAX_MAIL = 4;
  MAX_SENDERS = 2;

(* **Constants for Resource*)
CONST
  MAX_Resource_USERS = 1;

(* **Types and constants for Message_io_class*)
INCLUDE SVC1PPFX

(* **Class to provide fixed record I/O*)
INCLUDE MSGIO

(* Modified Brinch Hansen FIFO*)
INCLUDE FIFO

(* Standard Brinch Hansen RESOURCE*)
INCLUDE RESOURCE

(* Interprocess communication mailbox*)
INCLUDE MAILBOX

(* LISTING 2 *)
(* *Presentation layer record delimitier conversion*) (***)
INClude CR2ML

(* *Presentation layer data encryption*) (***)
INClude CRiPTV

(* *Process to simulate the Presentation layer*) (***)
INClude PRESENT

(* *Undefined layers of the network*)
INClude BLACKBOX

(* *Worker application process*) (***)
INClude WORKER2

(* *Server application process*) (***)
INClude SERVER2

TYPE

NODE_W = RECORD
  APP: WORKER_PROCESS;
  APP_EVT: MAIL_BOX_MONITOR;
  PRES: PRESENT_PROCESS; (***)
  PRES_EVT: MAIL_BOX_MONITOR; (***)
  BLACKBOX: BLACKBOX_PROCESS;
  BB_EVT: MAIL_BOX_MONITOR
END;

NODE_S = RECORD
  APP: SERVER_PROCESS;
  APP_EVT: MAIL_BOX_MONITOR;
  PRES: PRESENT_PROCESS; (***)
  PRES_EVT: MAIL_BOX_MONITOR; (***)
  BLACKBOX: BLACKBOX_PROCESS;
  BB_EVT: MAIL_BOX_MONITOR
END;

VAR

CONSOLE: RESOURCE_MONITOR;
SERVER: NODE_S;
WORKER: NODE_W;

BEGIN

INIT

  CONSOLE,
  SERVER.BB_EVT, WORKER.BB_EVT,
  SERVER.PRES_EVT, WORKER.PRES_EVT,
  SERVER.APP_EVT, WORKER.APP_EVT,
  SERVER.BLACKBOX(SERVER.PRES_EVT, SERVER.BB_EVT,
      WORKER.BB_EVT),
  WORKER.BLACKBOX(WORKER.PRES_EVT, WORKER.BB_EVT,
      SERVER.BB_EVT),
  SERVER.PRES(SERVER.APP_EVT, SERVER.PRES_EVT,

(** LISTING 2 **)            (** 33 **)
SERVER.BB_EVT, CR_DELIM),
WORKER.PRES(WORKER.APP_EVT, WORKER.PRES_EVT,
WORKER.BB_EVT, CR_DELIM),
SERVER.APP(SERVER.APP_EVT, SERVER.PRES_EVT),
WORKER.APP(CONSOLE, WORKER.APP_EVT,
WORKER.PRES_EVT)

END.
2.3: NET3

The session protocols of the ISO model are scattered throughout adjacent layers in most currently implemented networks. The purpose of the Session layer is to assist the higher layers in two ways. First, it controls the establishment of a communication session by having the two processes decide on "ground rules" for the session. This session administration service is also known as "binding the processes". Second, it controls the delimiting and synchronization of data operations through a session dialogue service.

A list of session functions could include the following:

1. Establish communications with the node which owns or controls the requested function or data.

2. Check that the communicating nodes have the software necessary for communication.

3. Exchange information about the protocols to be used in the communication.

4. Convert the high-level statements or requests of the user programs into the protocols of the lower layers.

5. Interpret end-of-record and end-of-file indicators in messages.

6. Perform end-to-end acknowledgement and sequence-number checking, if it is felt necessary to have additional checks on the lower layers.

7. Recover from a temporary break in the network without breaking the session.
8. Divide long messages into segments and use an acknowledgement protocol so that if a crash occurs, at most one segment has to be retransmitted.

9. Screen incoming calls, permitting only those from authorized users.

10. Issue or check passwords.

NET3 (see listing 3) has a Session layer which provides orderly creation and termination of sessions without data loss, full duplex communication paths, access control, and message chaining.

There are three notable additions to NET3. The Mailbox communication mechanism has been altered to allow the network packets to be assigned a priority. One possible priority scheme would segregate the packets into [1]:

1 = Control messages
2 = Real-time or fast interactive
3 = Slow interactive
4 = Batch-processing traffic
5 = Traffic which can be deferred

A new class has been added to provide standard entry points to the network. NETIO allows the application program to access the network just as MSGIO provides it with I/O to the disk and terminal.

A new class has been added to interpret any error messages sent to the application layer from the network and display them on the terminal.
Every host on the simulated network is initialized with a password. Anyone attempting to establish a session must include the appropriate password with the request or the request will be refused.

The message-chaining facility allows a source process to send multiple messages to a destination and have the messages assembled into a single large message at the destination. This chaining continues until the source process either explicitly releases the message or sends a request that all data currently quarantined be discarded. The destination process receives no information that the data being received has been quarantined or if some data has been discarded.
NET3 ACCESS DIAGRAM
NET3 SESSION PROTOCOL

1) LISTEN
2) ESTABLISH
3) REQUEST
4) START/BREAK
5) START/BREAK
6) START
7) BREAK
8) BREAK
9) BREAK

Figure 10
SESSION LAYER

FINITE STATE AUTOMATON

Figure 11
(* "Cpascal Prefix" *)

INCLUDE NETPFX

(* ******************************************************************************
 *  PROGRAM NET3
 *  Application, Presentation, Session, and Blackbox.
 *  ******************************************************************************
 *  Programmer: Ronald C. Albury
 *  Date Written: July 1982
 *  Computer: Interdata 8/32
 *  Copyright 1982 by Ronald C. Albury

--------------------------------------------------------------------)

(*NOTE: modifications to previous program are indicated *)
(* by (****) *)

(* **Packet description**)

TYPE
  PRIORITY_TYPE = (LOW_PRI, MED_PRI, HIGH_PRI); (****)
  SESSION_COMMANDS = (LISTEN, ESTABLISH, REQUEST, (****)
     START, IMMEDIATE, CHAIN, END_CHAIN, ABORT_CHAIN,
     BREAK);
  CHAIN_TYPE = IMMEDIATE..ABORT_CHAIN; (****)
  STATUS_TYPE = (NO_LOCAL_SESSION, (***)
     NO_REMOTE_SESSION, LOCAL_IN_SESSION, REMOTE_IN_SESSION,
     BAD_PASSWORD, SESSION_ENDING, BUSY);
  PKT_STATUS_TYPE = SET OF STATUS_TYPE; (****)
  DIRECTION_TYPE = (INCOMING, OUTGOING);
  SECURITY_TYPE = (SECRET, PUBLIC);
  FILE_FORMAT_TYPE = (CR_DELIM, NL_DELIM);
  PACKET_TYPE = RECORD
     PRIORITY: PRIORITY_TYPE;
     SESSION_CMD: SESSION_COMMANDS;
     STATUS: PKT_STATUS_TYPE;
     SECURITY: SECURITY_TYPE;
     FILE_FORMAT: FILE_FORMAT_TYPE;
     DIRECTION: DIRECTION_TYPE;
     TEXT: MESSAGE_TYPE
      END;

(* **Constants for packet status messages** *) (****)

CONST
  FIRST_ERROR = NO_LOCAL_SESSION;
  LAST_ERROR = BUSY;
  ERROR_MSG = ('NO LOCAL SESSION',
     'NO REMOTE SESSION',
     'LOCAL IN SESSION',
     'REMOTE IN SESSION',
     'BAD PASSWORD',
     'SESSION ENDING',
     'REMOTE SESSION BUSY');

(* ** LISTING 3 **)
ARRAY [STATUS_TYPE] OF MESSAGE_TYPE;

(* *Constants for password check*) (***)  
TYPE  
HOST_ID_TYPE = (HOST_S, HOST_W);  
CONST  
PASSWORD = ('HOST S PASSWORD',  
'HOST W PASSWORD');  
ARRAY [HOST_ID_TYPE] OF MESSAGE_TYPE;

(* *Constants for Mail_box_monitor*)  
CONST  
MAX_MAIL = 6;  
MAX_SENDERS = 2;

(* *Constants for Resource*)  
CONST  
MAX_RESOURCE_USERS = 1;

(* *Constants for Session layer*)  
CONST  
MAX_SESSION_WAIT = 2;  
MAX_CHAIN = 7;

(* *Types and constants for Message_io_class*)  
INCLUDE SVC1PFX

(* *Class to provide fixed record I/O*)  
INCLUDE MSGIO

(* *Modified Brinch Hansen FIFO*)  
IN FIFO

(* *Standard Brinch Hansen Resource*)  
INCLUDE RESOURCE

(* *Prioritized communication mailbox*) (***)  
INCLUDE MAILBOX3

(* *Standard entries to the network*) (***)  
INCLUDE NETIO

(* *Class for reporting network errors*) (***)  
INCLUDE ERROR

(* *Undefined layers of the network*)  
INCLUDE BLACKBOX

(* *Presentation layer record delimiter conversion*)  
INCLUDE CR2NL

(* *Presentation layer data encryption*)  
INCLUDE CRIPtv

(** LISTING 3 **)  
(* 42 *)
(* #Process to simulate Presentation layer#)  
INCLUDE PRESENT

(* #Process to simulate Session layer#)  (****)  
INCLUDE SESSION

(* #Worker application process#)  (****)  
INCLUDE WORKER3

(* #Server application process#)  (****)  
INCLUDE SERVER3

TYPE
NODE_W = RECORD
  APP: WORKER_PROCESS;
  APP_EVT: MAIL_BOX_MONITOR;
  PRES: PRESENT_PROCESS;
  PRES_EVT: MAIL_BOX_MONITOR;
  SESS: SESSION_PROCESS;
  SESS_EVT: MAIL_BOX_MONITOR;
  BLACKBOX: BLACKBOX_PROCESS;
  BB_EVT: MAIL_BOX_MONITOR
END;

NODE_S = RECORD
  APP: SERVER_PROCESS;
  APP_EVT: MAIL_BOX_MONITOR;
  PRES: PRESENT_PROCESS;
  PRES_EVT: MAIL_BOX_MONITOR;
  SESS: SESSION_PROCESS;
  SESS_EVT: MAIL_BOX_MONITOR;
  BLACKBOX: BLACKBOX_PROCESS;
  BB_EVT: MAIL_BOX_MONITOR
END;

VAR
  CONSOLE: RESOURCE_MONITOR;
  SERVER: NODE_S;
  WORKER: NODE_W;

BEGIN
  INIT
  CONSOLE,
  SERVER.BB_EVT, WORKER.BB_EVT,
  SERVER.SESS_EVT, WORKER.SESS_EVT,
  SERVER.PRES_EVT, WORKER.PRES_EVT,
  SERVER.APP_EVT, WORKER.APP_EVT,
  SERVER.BLACKBOX(SERVER.SESS_EVT, SERVER.BB_EVT, WORKER.BB_EVT),
  WORKER.BLACKBOX(WORKER.SESS_EVT, WORKER.BB_EVT,
  SERVER.BB_EVT),
  SERVER.SESS(SERVER.PRES_EVT, SERVER.SESS_EVT,
  SERVER.BB_EVT, HOST_S),

(*** LISTING 3 ***)

(* 43 *)
WORKER.SESSION(WORKER.PRES_EVT, WORKER.SESSION_EVT, WORKER.BE_EVT, HOST_W),
SERVER.PRES(SERVER.APP_EVT, SERVER.PRES_EVT, SERVER.SESSION_EVT, CR_DELIM),
WORKER.PRES(WORKER.APP_EVT, WORKER.PRES_EVT, WORKER.SESSION_EVT, CR_DELIM),
SERVER.APP(CONSOLE, SERVER.APP_EVT, SERVER.PRES_EVT),
WORKER.APP(CONSOLE, WORKER.APP_EVT, WORKER.PRES_EVT)
END.
The transport layer is potentially the most complex protocol of the model. It is the bridge between the services offered to the user and what the network actually offers. It insulates the upper layers from changes in the network. It provides the upper layers with a transparent connection to the network no matter to what kind of network the host is connected.

If the host is connected to a perfect network that guarantees correct delivery of the messages, then the transport layer is primarily concerned with the efficient use of the network. If, however, the network occasionally loses or scrambles a message, the Transport layer must also provide an end-to-end error correcting protocol to ensure that all messages sent were correctly received at the proper destination.

The ISO Model [3] defines the role of the Transport layer as follows:

1) Mapping transport address onto network address
2) End-to-end multiplexing of transport connections onto network connections
3) Establishing and terminating transport connections
4) Controlling the end-to-end sequencing of individual connections
5) Detecting end-to-end errors, and monitoring the quality of the service
6) Recovering from end-to-end errors
7) End-to-end segmenting and blocking of messages
8) Controlling the end-to-end flow on individual connections
9) Providing supervisory functions.
10) Transferring expedited transport-service-data-units

One way that the Transport layer assures efficient use of the network is by multiplexing several sessions onto a single network connection. The average transmission rate during an interactive session is usually less than 20 bits per second for both directions of transmission combined [1]. This grossly underutilizes even a voice grade transmission line. Yet low bandwidth lines cannot be used because of the potential need for delivering large quantities of data quickly. An economical solution to the underutilization of transmission lines is to allow several sessions on one host to use the same high speed transmission facility. The transport protocol rotates between the sessions, sending messages in successive high speed bursts.

The Transport layer must also be sure that its peer has enough buffer space to receive all of the messages it is being sent. One solution to the buffer allocation problem is to send only an agreed-upon quantity of data, then wait for the destination to send a request for more. This solution can be expanded to the general principle of using tokens to indicate the buffer space available at the destination. The sender decrements its number of tokens for each message sent, and the destination replenishes them as its buffers become available. A continuous two-way dialogue can be supported in this way, with messages going in one
direction and tokens simultaneously returning in the other.

A two-way dialogue is also necessary to provide the sessions with an error-free data transfer over an imperfect network. The sender does not assume that the messages reached their destination in good condition unless it receives an acknowledgment, and keeps a copy of all outstanding messages in case they must be re-transmitted. If an acknowledgment is not received in a reasonable period of time, the Transport layer sends the lost message again. This protocol is complicated by the fact that an acknowledgment may itself be lost in transmission, and because messages/acknowledgments may actually only be delayed rather than lost.

Finally, the Transport layer provides connection management for the sessions on the host. It allows a transport user to be identified by a transport address without regard to its location in the network. It controls the connections to the network. And it allows either session-entity to terminate the connection and have its peer session-entity informed of the termination.

NET4 (see listing 4) is the final program of STERLING. Three new nodes have been added to the network, and a Transport layer has been added to the Worker and Server nodes. The Blackbox has been slightly modified and is now assumed to be a Local Area Network. The Worker now has two Servers from which it can request files. One Server oversees files A and B; the other oversees files C and D. A new node is added to the network to simulate transmission loss; this new node fails to forward packets at pre-set intervals. Also, a new process and monitor
have been added to the worker and server nodes to detect lost messages and notify the Transport layer when it should re-transmit a lost packet.

The Transport layer was patterned very loosely on the National Bureau of Standards' Draft Report on Transport Protocols [8]. It uses a simple alternating-bit protocol to insure that all messages sent were received in the correct order. Its primary concerns are with connection management and end-to-end error correction.
NET4 ACCESS DIAGRAM

Figure 12
NET4 TOPOLOGY

Figure 13
NET4 CONNECTION PROTOCOL

1) LISTEN
2) ESTABLISH
3) REQUEST - CONNECT
4) REQUEST - INQUIRE
5) ACCEPT
6) REQUEST
7) START
8) START - DATA_XFER
9) START - DATA_XFER
10) START
11) START

Figure 14
TRANSPORT LAYER

FINITE STATE AUTOMATON
INCLUDE NETPFX

(* ************************************************************
  *
  * PROGRAM NET4
  *
  * Application, Presentation, Session, Transport,
  * and Network layers.
  *
  *************************************************************)

# Programmer: Ronald C. Albury
# Date Written: 10/11/82
# Computer: Interdata 8/32
# Copyright 1982 by Ronald C. Albury

(*NOTE: modifications to previous program are indicated *)
(* by (****))
(*

(* Packet description *)

TYPE
  TRANSPORT_COMMANDS = (CONNECT, DISCONNECT, INQUIRE, (****)
    T_Q_ACCEPT, T_Q_DATA_ACK, ACCEPT, DATA_XFER, DATA_ACK);
  DATA_WINDOW_TYPE = 0..1; (****)
  HOST_ID_TYPE = (HOST_W, HOST_S_1, HOST_S_2); (****)
  PRIORITY_TYPE = (LOW_PRI, MED_PRI, HIGH_PRI);
  SESSION_COMMANDS = (LISTEN, ESTABLISH, REQUEST,
    START, IMMEDIATE, CHAIN, END_CHAIN, ABORT_CHAIN,
    BREAK);
  CHAIN_TYPE = IMMEDIATE..ABORT_CHAIN;
  STATUS_TYPE = (NO_LOCAL_SESSION,
    NO_REMOTE_SESSION, LOCAL_IN_SESSION, REMOTE_IN_SESSION,
    BAD_PASSWORD, SESSIONENDING, BUSY, NO_LOCAL_CONNECTION,
    NO_REMOTE_CONNECTION, NO_SUCH_HOST, DESTINATION_NODE_DOWN,
    CONNECTION_BROKEN); (****)
  PKT_STATUS_TYPE = SET OF STATUS_TYPE;
  DIRECTION_TYPE = (INCOMING, OUTGOING);
  SECURITY_TYPE = (SECRET, PUBLIC);
  FILE_FORMAT_TYPE = (CR_DELIM, NL_DELIM);
  PACKET_TYPE = RECORD
    TRANS_CMD: TRANSPORT_COMMANDS;
    NAME: MESSAGE_TYPE;
    DESTINATION: HOST_ID_TYPE;
    SOURCE: HOST_ID_TYPE;
    S_CONNUP: INTEGER;
    D_CONNUP: INTEGER;
    DATA_SEQ: DATA_WINDOW_TYPE;
    PRIORITY: PRIORITY_TYPE;
    SESSION_CMD: SESSION_COMMANDS;
    STATUS: PKT_STATUS_TYPE;
    SECURITY: SECURITY_TYPE;
    FILE_FORMAT: FILE_FORMAT_TYPE;
    DIRECTION: DIRECTION_TYPE;

(* LISTING 4 *)
TEXT: MESSAGE_TYPE
END;

(* **Constants for packet status messages*)
CONST
FIRST_ERROR = NO_LOCAL_SESSION;
LAST_ERROR = CONNECTION_BROKEN;
ERROR_MSG = ('NO LOCAL SESSION',
              'NO REMOTE SESSION',
              'LOCAL IN SESSION',
              'REMOTE IN SESSION',
              'BAD PASSWORD',
              'SESSION ENDING',
              'REMOTE SESSION BUSY',
              'NO LOCAL CONNECTION',
              'NO REMOTE CONNECTION',
              'NO SUCH SERVER',
              'DEST. NODE OFF LINE',
              'CONNECTION BROKEN');
ARRAY [STATUS_TYPE] OF MESSAGE_TYPE;

(* **Constants for password check*)
CONST
PASSWORD = ('HOST W PASSWORD',
             'HOST S 1 PASSWORD',
             'HOST S 2 PASSWORD');
ARRAY [HOST_ID_TYPE] OF MESSAGE_TYPE;

(* **Constants for transport address look-up*)
CONST
DIRECTORY = ('HOST_W',
              'HOST_S_1',
              'HOST_S_2');
ARRAY [HOST_ID_TYPE] OF MESSAGE_TYPE;

(* **Constants for Mail_box_monitor*)
CONST
MAX_MAIL = 6;
MAX_SENDERS = 3;

(* **Constants for Resource*)
CONST
MAX_RESOURCE_USERS = 1;

(* **Constants for Session layer*)
CONST
MAX_SESSION_WAIT = 2;
MAX_CHAIN = 7;

(* Types and constants for Message_io_class*)
INCLUDE SVC1PPFX

(* *Class to provide fixed record I/O*)

(* LISTING 4 *)

(* 54 *)
INCLUDE MSGIO

(* *Modified Brinch Hansen FIFO*)
INCLUDED FIFO

(* *Standard Brinch Hansen RESOURCE*)
INCLUDED RESOURCE

(* *Prioritized communication mailbox*)
INCLUDED MAILBOX3

(* *Revised network entries*)
INCLUDED NETIO4

(* *Class for reporting network errors*)
INCLUDED ERROR

(* *Process to simulate a local area network*)
INCLUDED LOCNET4

(* *Presentation layer record delimiter conversion*)
INCLUDED CR2NL

(* *Presentation layer data encryption*)
INCLUDED CRIFTY

(* *Process to simulate Presentation layer*)
INCLUDED PRESENT

(* *Process to simulate Session layer*)
INCLUDED SESSION4

(* *Monitor for controlling time-outs*)
INCLUDED CLOCK

(* *Time-out simulator*)
INCLUDED TIMER

(* *Process to simulate Transport layer*)
INCLUDED TRANS

(* *Process to simulate data loss in the network*)
INCLUDED BADNODE

(* *Worker application process*)
INCLUDED WORKER4

(* *Server application process*)
INCLUDED SERVER4

(* LISTING 4 *)

(* 55 *)
TYPE
    NODE_W = RECORD
        APP: WORKER_PROCESS;
        APP_EVT: MAIL_BOX_MONITOR;
        PRES: PRESENT_PROCESS;
        PRES_EVT: MAIL_BOX_MONITOR;
        SESS: SESSION_PROCESS;
        SESS_EVT: MAIL_BOX_MONITOR;
        TRANS: TRANSPORT_PROCESS;
        TRANS_EVT: MAIL_BOX_MONITOR;
        CLOCK: CLOCK_MONITOR;
        TIMER: TIMEOUT_PROCESS;
        NET: NETWORK_PROCESS;
        NET_EVT: MAIL_BOX_MONITOR
    END;

    NODE_S = RECORD
        APP: SERVER_PROCESS;
        APP_EVT: MAIL_BOX_MONITOR;
        PRES: PRESENT_PROCESS;
        PRES_EVT: MAIL_BOX_MONITOR;
        SESS: SESSION_PROCESS;
        SESS_EVT: MAIL_BOX_MONITOR;
        TRANS: TRANSPORT_PROCESS;
        TRANS_EVT: MAIL_BOX_MONITOR;
        CLOCK: CLOCK_MONITOR;
        TIMER: TIMEOUT_PROCESS;
        NET: NETWORK_PROCESS;
        NET_EVT: MAIL_BOX_MONITOR
    END;

    NODE_GLITCH = RECORD (***)
        NET: BAD_NODE_PROCESS;
        NET_EVT: MAIL_BOX_MONITOR
    END;

VAR
    CONSOLE: RESOURCE_MONITOR;
    SERVER1: NODE_S; (***)
    SERVER2: NODE_S; (***)
    WORKER: NODE_W;
    GLITCH: NODE_GLITCH; (***)

BEGIN
    INIT
        CONSOLE,
        SERVER1.NET_EVT, SERVER2.NET_EVT,
        WORKER.NET_EVT, GLITCH.NET_EVT,
        SERVER1.TRANS_EVT, SERVER2.TRANS_EVT,
        WORKER.TRANS_EVT,
        SERVER1.CLOCK, SERVER2.CLOCK,
        WORKER.CLOCK,
        SERVER1.SESS_EVT, SERVER2.SESS_EVT,
WORKER.SESSION.EVT,
SERVER1.PRES.EVT, SERVER2.PRES.EVT,
WORKER.PRES.EVT,
SERVER1.APP.EVT, SERVER2.APP.EVT,
WORKER.APP.EVT,
GLITCH.NET.EVT (GLITCH.NET.EVT, SERVER1.NET.EVT),
SERVER1.NET.EVT(SERVER1.TRANS.EVT, SERVER1.NET.EVT,
    SERVER2.NET.EVT, HOST_S_1),
SERVER2.NET.EVT(SERVER2.TRANS.EVT, SERVER2.NET.EVT,
    WORKER.NET.EVT, HOST_S_2),
WORKER.NET(WORKER.TRANS.EVT, WORKER.NET.EVT,
    GLITCH.NET.EVT, HOST_W),
SERVER1.TIMER (SERVER1.CLOCK, SERVER1.TRANS.EVT),
SERVER2.TIMER (SERVER2.CLOCK, SERVER2.TRANS.EVT),
WORKER.TIMER (WORKER.CLOCK, WORKER.TRANS.EVT),
SERVER1.TRANS (SERVER1.SESSION.EVT, SERVER1.TRANS.EVT,
    SERVER1.NET.EVT, SERVER1.CLOCK, HOST_S_1),
SERVER2.TRANS (SERVER2.SESSION.EVT, SERVER2.TRANS.EVT,
    SERVER2.NET.EVT, SERVER2.CLOCK, HOST_S_2),
WORKER.TRANS (WORKER.SESSION.EVT, WORKER.TRANS.EVT,
    WORKER.NET.EVT, WORKER.CLOCK, HOST_W),
SERVER1.SESSION.EVT (SERVER1.PRES.EVT, SERVER1.SESSION.EVT,
    SERVER1.TRANS.EVT, HOST_S_1),
SERVER2.SESSION.EVT (SERVER2.PRES.EVT, SERVER2.SESSION.EVT,
    SERVER2.TRANS.EVT, HOST_S_2),
WORKER.SESSION.EVT (WORKER.PRES.EVT, WORKER.SESSION.EVT,
    WORKER.TRANS.EVT, HOST_W),
SERVER1.PRES.EVT(SERVER1.APP.EVT, SERVER1.PRES.EVT,
    SERVER1.SESSION.EVT, CR_DELIM),
SERVER2.PRES.EVT(SERVER2.APP.EVT, SERVER2.PRES.EVT,
    SERVER2.SESSION.EVT, NL_DELIM),
WORKER.PRES.EVT(WORKER.APP.EVT, WORKER.PRES.EVT,
    WORKER.SESSION.EVT, CR_DELIM),
SERVER1.APP.EVT(CONSOLE, SERVER1.APP.EVT,
    SERVER1.PRES.EVT, 'A', 'B'),
SERVER2.APP.EVT(CONSOLE, SERVER2.APP.EVT,
    SERVER2.PRES.EVT, 'C', 'D'),
WORKER.APP.EVT(CONSOLE, WORKER.APP_EVT,
    WORKER.PRES.EVT)

END.

(* LISTING 4 *)

(* 57 *)
FURTHER WORK

Sterling is intended as a source of assignments for a course in computer networks. It is designed to be easily understood and easily modified. Once the students have familiarized themselves with the programming style and the functions of the four layers introduced, they can be expected to make major revisions to the programs.

One type of assignment would be to add the bottom layers of the ISO Model to Sterling (see figure 3). The Physical Layer could either be left as a Blackbox, or modified to more closely resemble a bit-oriented data stream. Pascal-like algorithms for several Data Link protocols can be found in Tanenbaum’s book [15]. The Network Layer could either be left as a simple ring network, or expanded to a point-to-point network concerned with routing and congestion control.

A second type of assignment would be to expand and modify the services provided by the existing layers. The layers’ functions listed in this report are only a small subset of possible functions, and only a small subset of these are implemented. Students could also multi-thread the layers, allowing more than one application program to reside on a node.

A third type of assignment would be to attempt to make Sterling more flexible and robust. The Session and Transport protocols are currently unable to recover gracefully from erroneous commands and would be unusable in a production environment. Students could be referred to the National Bureau of
REFERENCES


10. Robert Young, Personal communication.


Appendix A

INCLUDE MODULES
(*$ Netpfx $*)
(*$ C-Pascal Kernel for 8-32 $*)

(*$ Kernel $*)

**Type Kern_svc1_block = array [1..24] of byte;**
**Type attrindex = (caller, heap top, proc line, proc result,**
** runtime);**
**Function attribute (a: attrindex): integer;**
**Procedure setheap (a: integer);**
**Procedure start (a: integer);**
**Procedure stop (a, b: integer);**
**Procedure wait;**
**Function realtime: integer;**
**Procedure svc1 (var param: univ kern_svc1_block);**
**Procedure svc2;**
**Procedure svc7;**
**Procedure getmem;**
**Procedure breakpnt (ln: integer);**
**End (*kernel*);**

(*@ Added for network *)
(* Disk file id's and logical units *)

**Const**
**trace_output = 9;**
**Terminal = 10;**
**File_a_lu = 11;**
**File_b_lu = 12;**
**File_c_lu = 13;**
**File_d_lu = 14;**
**First_file_id = 'A';**
**Last_file_id = 'D';**
**Type**
**file_range = first_file_id .. last_file_id;**
**Const**
**file_lu = (11, 12, 13, 14): array [file_range] of integer;**

(* Constants and types for fixed length i/o *)
**Const**
**Message_length = 20;**
**Type**
**message_type = array [1..message_length] of char;**
(*$SVC1PFX$*)

(*$Types and constants for Message_io_class$*)

(*$SVC1 PREFIX$*)

(*$Record structure of the SVC1 parameter$*)

TYPE
  ERROR_SVC1_TYPE = RECORD
    DI: BYTE;
    DD: BYTE
  END;

  SVC1_BLOCK_TYPE =
    PACKED RECORD
      FUNC       : BYTE;
      LOG_UNIT   : BYTE;
      D_I_ERROR  : BYTE;
      D_D_ERROR  : BYTE;
      BUFFER_START : INTEGER;
      BUFFER_END  : INTEGER;
      RANDOM_ADRS : INTEGER;
      LENGTH_XFER : INTEGER;
      RESERVED   : INTEGER
    END;

(*$Function flags for communicating with SVC1$*)

CONST
  FUN_READ_SVC1   = #40;
  FUN_WRITE_SVC1  = #20;
  FUN_BINARY_SVC1 = #10;
  FUN_ASCII_SVC1  = #00;
  FUN_WAIT_SVC1   = #08;
  FUN_RANDOM_SVC1 = #04;
  FUN_SEQUENC_SVC1 = #00;
  FUN_PROCED_SVC1 = #02;
  FUN_FORMAT_SVC1 = #00;
  FUN_IMAGE_SVC1  = #01;
  FUN_REWIND_SVC1 = #00;
  FUN_BKSPAC_SVC1 = #00;
  FUN_FSPAC_SVC1  = #00;
  FUN_WRITE_FM_SVC1 = #88;
  FUN_SKIP_TO_FM_SVC1 = #84;
  FUN_BKSPAC_TO_FM_SVC1 = #82;

(*$END SVC1 PREFIX$*)
TYPE MESSAGE_IO_CLASS = CLASS;

PROGRAMMING NOTES:
* MESSAGE_IO_CLASS provides standard entry points for
  * interfacing with the SVC1 supervisory call. Allows
  * fixed record I/O and rewind capabilities to specified
  * logical units.
  *
Programmer: Ronald C. Albury
Date Written: 3/25/82
Computer: Interdata 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

CONST
  * FUN_-:__,__SVC1: Bit flags used to communicate with
    * the supervisory call.
  *
  * TYPE
    * SVC1_BLOCK_TYPE: Record structure used to pass
      * parameters to the supervisory I/O calls.
    * ERROR_SVC1_TYPE: Record structure of the status
      * bytes from the supervisory call.
    * MESSAGE_TYPE: Array of characters.
  *
INTERNAL

VAR
  * PARAM: The parameter block for the supervisory
    * calls.

VAR
  * PARAM: SVC1_BLOCK_TYPE;

PROCEDURE ENTRY READ

VAR
  * PAD: Loop variable to pad the text buffer with
    * blanks if less then Message_length bytes are
    * read in.

PARAMETERS
  * IO DEVICE: Logical unit for the input request.
  *
PROCEDURE ENTRY READ ( IO DEVICE: BYTE;
  VAR TEXT: UNIV MESSAGE_TYPE;
  VAR ERROR: ERROR_SVC1_TYPE );

VAR
  * PAD: 0..MESSAGE_LENGTH;

BEGIN

(* A-3 *)
(*Set the SVC1 parameters for a sequential ASCII read*)
PARAM.FUNC := FUN_ASCII_SVC1 + FUN_SEQUEN_SVC1
+ FUN_READ_SVC1;
(*Set the logical unit to read from*)
PARAM.LOG_UNIT := IO_DEVICE;
(*Set the address to store the read data*)
PARAM.BUFFER_START := ADDRESS (TEXT);
PARAM.BUFFER_END := ADDRESS (TEXT) + SIZE(TEXT) -1;
(*Execute an SVC1*)
SVC1 ( PARAM );
(*Pad out the buffer with blanks*)
FOR PAD := PARAM.LENGTH_XFER TO MESSAGE_LENGTH DO
  TEXT [PAD] := ' ';
{Endfor}
(*Set the status bytes*)
ERROR_DI := PARAM.D_I_ERROR;
ERROR_DD := PARAM.D_D_ERROR
(*End entry Read*)
END;

(****PROCEDURE ENTRY WRITE%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 * INTERNAL
  * VAR
   * MESSAGE: A local variable, necessary to make
   * ADDRESS function work correctly.
   * PARAMETERS
    * IN
     * IO_DEVICE: Logical unit for the output request.
     * TEXT: Buffer of characters to output.
    * OUT
     * ERROR: The status bytes of the I/O call.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PROCEDURE ENTRY WRITE ( IO_DEVICE: BYTE;
TEXT: UNIV MESSAGE_TYPE;
VAR ERROR: ERROR_SVC1_TYPE );
VAR
MESSAGE: MESSAGE_TYPE;
(*Begin entry Write*)
BEGIN
(*Set the SVC1 parameters for a sequential ASCII write*)
PARAM.FUNC := FUN_ASCII_SVC1 + FUN_SEQUEN_SVC1
+ FUN_WRITE_SVC1;
(*Set the logical unit to write to*)
PARAM.LOG_UNIT := IO_DEVICE;
(*Set the address of the data to be transferred*)
MESSAGE := TEXT; (*must be local variable for ADDRESS*)
PARAM.BUFFER_START := ADDRESS (MESSAGE);
PARAM.BUFFER_END := ADDRESS (MESSAGE) + SIZE(MESSAGE) -1;
(*Execute the SVC1*)
SVC1 ( PARAM );
(*Set the status bytes*)
(* A-4 *)
(### MSGIO ###)
(### Class to provide fixed record I/O ###)

ERROR_DD := PARAM.D_I_ERROR;
ERROR_DI := PARAM.D_D_ERROR
(*End entry Write*)
END;

(*PROCEDURE ENTRY REWIND*************************************************************************************
* PARAMETERS
* IN
*   IO_DEVICE: Logical unit for the rewind request.
* OUT
*   ERROR: The status bytes of the rewind call.
*************************************************************************************)
PROCEDURE ENTRY REWIND ( IO_DEVICE: BYTE;
                         VAR ERROR: ERROR_SVC1_TYPE );

VAR
PARAM: SVC1_BLOCK_TYPE;
(*Begin entry Rewind*)
BEGIN
(*Set the SVC1 parameters for rewind*)
   PARAM.FUNC := FUN_REWIND_SVC1;
(*Set the logical unit to be rewound*)
   PARAM.LOG_UNIT := IO_DEVICE;
(*Execute the SVC1*)
   SVC1 ( PARAM );
(*Set the status bytes*)
   ERROR_DI := PARAM.D_I_ERROR;
   ERROR_DD := PARAM.D_D_ERROR
(*End entry Rewind*)
END;

BEGIN
END;

(* A-5 * )
TYPE FIFO = CLASS ( LIMIT: INTEGER );

* MODIFIED PBH FIFO CLASS

* PROGRAMMER: PER BRINCH HANSEN
* MODIFIED BY: RONALD C. ALBURY
* DATE WRITTEN:
* COMPUTER: P.E. 8/32

* INTERNAL
* VAR
*   HEAD: Position of the oldest entry in the queue.
*   TAIL: Position of the newest entry in the queue.
*   LENGTH: Length of the queue.

* PARAMETERS
* IN
* LIMIT: Number of positions available in the queue

VAR
HEAD, TAIL, LENGTH: INTEGER;

FUNCTION ENTRY ARRIVAL: INTEGER;
BEGIN
  ARRIVAL := TAIL;
  TAIL := TAIL MOD LIMIT + 1;
  LENGTH := LENGTH + 1
END;

FUNCTION ENTRY DEPARTURE: INTEGER;
BEGIN
  DEPARTURE := HEAD;
  HEAD := HEAD MOD LIMIT + 1;
  LENGTH := LENGTH - 1
END;

(* New function entry EXAMINE *)
FUNCTION ENTRY EXAMINE: INTEGER;
BEGIN
 (* Set to FIFO head without changing the FIFO *)
  EXAMINE := HEAD
END;

(* New function entry SIZE *)
FUNCTION ENTRY SIZE: INTEGER;
BEGIN
 (* Set to the number of entries in the FIFO *)
  SIZE := LENGTH
END;
FUNCTION ENTRY EMPTY: BOOLEAN;
BEGIN
    EMPTY := (LENGTH = 0)
END;

(***New function entry OCCUPIED***)
FUNCTION ENTRY OCCUPIED: BOOLEAN;
BEGIN
    (*Set true if FIFO is occupied*)
    OCCUPIED := ( LENGTH <> 0 )
END;

(***New function entry FULL***)
FUNCTION ENTRY FULL: BOOLEAN;
BEGIN
    (*Set true if FIFO is full*)
    FULL := ( LENGTH = LIMIT )
END;

BEGIN (*FIFO INITIALIZATION*)
    HEAD := 1;
    TAIL := 1;
    LENGTH := 0
END;
(* Standard BRINCH HANSEN RESOURCE *)

(* RESOURCE *)

(* Resource manages a FIFO buffer *)

(* Maximum number of processes that will attempt to access the resource. *)

(* FIFO = A P.B.H. CLASS for managing a FIFO buffer *)

(* INTERNAL *)

(* VAR *)

(* FREE: A boolean variable that indicates if the resource is available. *)

(* Q: An array of Queue variables used as a fifo buffer for delaying processes. *)

(* NEXT: An instance of a P.B.H. FIFO class *)

(*_MONITOR = MONITOR; *)

VAR

FREE: BOOLEAN;

Q: ARRAY [1..MAXRESOURCE_USERS] OF QUEUE;

NEXT: FIFO;

PROCEDURE ENTRY REQUEST;
BEGIN
  IF ( FREE ) THEN
    FREE := FALSE
  ELSE
    (*@CAUTION@*)
    (* IF MAXRESOURCE_USERS IS TOO SMALL WE LOOSE A PROCESS HERE OR GET A DELAY QUEUE ERROR *)
    (DELAY ( Q [NEXT.ARRIVAL] ) ;)
    {ENDIF}
  END;

PROCEDURE ENTRY RELEASE;
BEGIN
  IF ( NEXT.EMPTY ) THEN
    FREE := TRUE
  ELSE
    CONTINUE ( Q [NEXT.DEPARTURE] )
  END;
END;

BEGIN (*MAIN BODY OF MONITOR*)

( = A-8 = )
FREE := TRUE;
INIT NEXT ( MAX_RESOURCE_USERS )
END;
TYPE MAIL_BOX_MONITOR = MONITOR;

MAIL_BOX_MONITOR is simply a means for one process to
receive messages from up to MAX_SENDER other processes.
It can store up to MAX_MAIL messages in it's FIFO
controlled MAIL_BUFFER.
If the receiver process attempts to pick up mail when
the buffer is empty, it is delayed until a sender process
deposits mail.
If a sender process attempts to deposit mail when the
buffer is full, it is delayed until the receiver process
picks up mail.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 3/25/82
LANGUAGE: CONCURRENT PASCAL ( BRINCH HANSEN [ K.S.U ] )
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

CONST
MAX_SENDERS = Maximum number of processes that
will send messages to the receiver.
MAX_MAIL = Maximum number of messages the monitor
can hold in it's buffer.

TYPE
FIFO = Modified Brinch Hansen FIFO class to
handle a FIFO buffer.
PACKET_TYPE = The record structure of the mail.

INTERNAL
VAR
RECEIVER: Queue variable to delay the receiver.
SENDER: Array of Queue variables used as a fifo
buffer for delaying senders.
DELAYED_SENDERS: Fifo to control SENDER buffer.
MAIL_BUFFER: Array of packets used as a fifo
buffer for storing mail.
NEXT_MAIL: Fifo to control MAIL_BUFFER.

VAR
DELAYED_SENDERS, NEXT_MAIL: FIFO;
MAIL_BUFFER: ARRAY [1..MAX_MAIL] OF PACKET_TYPE;
RECEIVER: QUEUE;
SENDER: ARRAY [1..MAX_SENDERS] OF QUEUE;

(# A-10 #)
(*PROCEDURE ENTRY GET*****************************************************)  
* PARAMETERS  
*  
* OUT  
*  
*  OUTGOING_MAIL: Receives the oldest entry  
* from the MAIL_BUFFER.  
*****************************************************************************)

PROCEDURE ENTRY GET (VAR OUTGOING_MAIL: PACKET_TYPE);
(*Begin entry GET*)

BEGIN  
(*If [there is no mail in the FIFO queue] then*)

IF (NEXT_MAIL.EMPTY) THEN  
(*Put the receiver to sleep*)

DELAY (RECEIVER);
(*Endif*)

(*Set outgoing mail to the oldest packet in the queue*)

OUTGOING_MAIL := MAIL_BUFFER [NEXT_MAIL.DEPARTURE];

(*If [there are senders sleeping] then*)

IF (DELAYED_SENDERS.OCCUPIED) THEN  
(*Wake up the oldest sleeper*)

CONTINUE (SENDER [DELAYED_SENDERS.DEPARTURE])
(*Endif*)

(*End entry GET*)

END;

(*PROCEDURE ENTRY DEPOSIT***********************************************)

* PARAMETERS  
*  
* IN  
*  
*  INCOMING_MAIL: A packet being deposited into the  
* MAIL_BUFFER by a sender process.  
*******************************************************************************)

PROCEDURE ENTRY DEPOSIT (INCOMING_MAIL: PACKET_TYPE);
(*Begin entry DEPOSIT*)

BEGIN  
(*If [all known senders are delayed] then*)

IF (DELAYED_SENDERS.FULL) THEN  
(*SHOULD NEVER HAPPEN UNLESS MAX_SENDER IS WRONG*)  
(*WE LOOSE THE MAIL*)
(*Else*)

ELSE

BEGIN  
(*If [mail queue is full] then*)

IF (NEXT_MAIL.FULL) THEN  
(*Put the sender to sleep*)

DELAY (SENDER [DELAYED_SENDERS.ARRIVAL]);
(*Endif*)

(*Store the mail in a FIFO queue*)

MAIL_BUFFER [NEXT_MAIL.ARRIVAL] := INCOMING_MAIL;

(*If [the receiver is sleeping] wake him up*)

CONTINUE (RECEIVER)

END
(*Endif*)

(*End entry DEPOSIT*)

(* A-11 *)
END;

BEGIN (*MONITOR INITIALIZATION*)

    INIT
    NEXT_MAIL (MAX_MAIL),
    DELAYED_SENDERS (MAX_SENDERS)

END; (*MAIL_BOX_MONITOR*)
TYPE WORKER_PROCESS = PROCESS (CONSOLE: RESOURCE_MONITOR;
FROM_NET: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR);

The WORKER_PROCESS is an application layer process that transfers remote files to the operator console.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

TYPE

MESSAGE_IO_CLASS = A class that uses supervisory calls to handle fixed record I/O to specified logical units.
PACKAGE_TYPE = Record structure of the network packets.
MESSAGE_TYPE = Array of characters.
ERROR_SVC1_TYPE = Record structure of the status bytes from the supervisory call.
MAIL_BOX_MONITOR = A monitor used for passing packets between processes.
RESOURCE_MONITOR = Allows only one process to access a resource at a time.

INTERNAL

VAR

OP: Used to write lines of the transferred file to the operator.
PACKAGE: A network packet this process uses to communicate with the network.
TEXT: Array of characters used to communicate the operator.
OP_STATUS: Receives the status bytes from the MESSAGE_IO_CLASS. Not used here, but necessary for the calls to OP.

PARAMETERS

CONSOLE: The RESOURCE_MONITOR used to reserve the console for exclusive I/O.
FROM_NET: The monitor used to receive packets from the network.
TO_NET: The monitor used to send packets to the network.

VAR

OP: MESSAGE_IO_CLASS;
PACKAGE: PACKAGE_TYPE;
TEXT: MESSAGE_TYPE;
OP_STATUS: ERROR_SVC1_TYPE;
("*Begin Worker process*")
BEGIN
("*Initialize the interface to the operator*")
INIT OP;
("*Cycle forever*")
CYCLE
("*Get the id for the file to be transferred*")
CONSOLE.REQUEST;
TEXT := 'ENTER FILE ID.  - ';
TEXT [18] := FIRST_FILE_ID;
OP.WRITE (TERMINAL, TEXT, OP_STATUS);
OP.READ (TERMINAL, TEXT, OP_STATUS);
CONSOLE.RELEASE;
("*Send the request to the server*")
PACKET.TEXT := TEXT;
TO_NET.DEPOSIT (PACKET);
("*Transfer the file to the console*")
CONSOLE.REQUEST;
("*Repeat until end of file*")
REPEAT
("*Get a line from the network*")
FROM_NET.GET (PACKET);
("*Output it to the console*")
OP.WRITE (TERMINAL, PACKET.TEXT, OP_STATUS);
("*End repeat*")
UNTIL (PACKET.TEXT [1] = '/')
& (PACKET.TEXT [2] = ' * ');
CONSOLE.RELEASE
("*End cycle*")
END
("*End Worker process*")
END;
TYPE SERVER_PROCESS = PROCESS (FROM_NET: MAIL_BOX_MONITOR;
   TO_NET: MAIL_BOX_MONITOR);

(* The server process is an application layer process that *
  does the disk I/O for a remote worker process. *)

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

  TYPE
    MESSAGE_IO_CLASS = A class that uses supervisory calls to handle fixed record I/O to specified logical units.
    PACKET_TYPE = Record structure of the network packets.
    MESSAGE_TYPE = Array of characters.
    ERROR_SVC1_TYPE = Record structure of the status bytes from the supervisory call.
    MAIL_BOX_MONITOR = A monitor used for passing packets between processes.

INTERNAL

  VAR
    DISK: Used to input lines of a disk file.
    PACKET: A network packet used to communicate with the network.
    TEXT: Array of characters used for the disk I/O.
    FILE_ID: The id of the file the worker process is requesting.
    VALID_FILE_IDS: A set of the valid id's this process can access.
    FILE_LU: An array of logical units that are subscripted with file id's. Used to look up the logical unit of a file.
    NEXT_LU: Used in initializing FILE_LU.
    INDEX: Used in initializing FILE_LU.
    OP_STATUS: Receives the status bytes form the MESSAGE_IO_CLASS.

PARAMETERS

  FROM_NET: The monitor used to receive packets from the network.
  TO_NET: The monitor used to send packets to the network.

VAR

  DISK: MESSAGE_IO_CLASS;
  PACKET: PACKET_TYPE;
  TEXT: MESSAGE_TYPE;
  FILE_ID: CHAR;
(*$ Server application process $*)

VALID_FILE_ID: SET OF CHAR;
FILE_LU: ARRAY [FILE_RANGE] OF BYTE;
NEXT_LU: BYTE;
INDEX: FILE_RANGE;
OP_STATUS: ERROR_SVC1_TYPE;

(*Begin Server process*)
BEGIN
(*Initialize the interface to the disk files*)
IN10 DISK;
(*Set up an array to reference logical unit numbers*)
(*by the character id's of the files*)
NEXT_LU := TERMINAL + 1;
(*For all file id's do*)
FOR INDEX := FIRST_FILE_ID TO LAST_FILE_ID DO
BEGIN
(*Remember that it is a valid id*)
VALID_FILE_ID := VALID_FILE_ID + [INDEX];
(*Set it's logical unit number*)
FILE_LU [INDEX] := NEXT_LU;
NEXT_LU := NEXT_LU + 1
END;
(*Endfor*)
(*Cycle forever*)
CYCLE
(*Get the request from the net*)
FROM_NET.GET (PACKET);
FILE_ID := PACKET.TEXT [1];
(*If [it is a valid file id] then*)
IF (FILE_ID IN VALID_FILE_ID) THEN
(*Transfer the file*)
BEGIN
(*Read in a line from the disk*)
DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS);
(*While not [end of file] do*)
WHILE (OP_STATUS.DI = 0) AND (OP_STATUS.DD = 0) DO
BEGIN
(*Send it out on the network*)
PACKET.TEXT := TEXT;
TO_NET.DEPOSIT (PACKET);
(*Read in a new line from the disk file*)
DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS)
END;
(*Endwhile*)
(*Rewind the disk file*)
DISK.REWIND (FILE_LU [FILE_ID], OP_STATUS);
(*Send an EOF packet out on the network*)
PACKET.TEXT := '/
TO_NET.DEPOSIT (PACKET)
END
(*Else (an invalid file id]*)
ELSE

(* A-16 *)
(#$Server0$#$)
(#$Server application process#$)

(#Send an error message#)
BEGIN
PACKET.TEXT := '/$ BAD FILE ID - $';
PACKET.TEXT[19] := FILE_ID;
TQ_NET.DEPOSIT (PACKET)
END
(#Endif#)
(#End cycle#)
END
(#End Server#)
END;
TYPE BLACKBOX_PROCESS = 
    PROCESS ( TO_APP: MAIL_BOX_MONITOR;
             EVENT: MAIL_BOX_MONITOR;
             NEXT_NODE: MAIL_BOX_MONITOR);

(* The BLACKBOX layer represents the hardware and software *
  necessary for two processes to communicate on a network
*)

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

* TYPE
  * PACKET_TYPE = Record structure of the network
  * packets.
  * MAIL_BOX_MONITOR = A monitor used for passing
  * packets between processes.

INTERNAL

VAR

  PACKET: A network packet that the layer processes

PARAMETERS

  TO_APP: The monitor used to send packets up toward
           the application layer.
  EVENT: The monitor this layer uses to receive
          packets.
  NEXT_NODE: The monitor used to send packets to
             the next node in the network.

VAR

  PACKET: PACKET_TYPE;
BEGIN

(*Cycle forever*)
CYCLE

(*Wait for a packet*)
EVENT.GET (PACKET);

(*If [an outgoing packet] then*)
IF ( PACKET.DIRECTION = OUTGOING ) THEN

(*Set it as an incoming packet*)
PACKET.DIRECTION := INCOMING;
(*Pass it on to the next node*)
NEXT_NODE.DEPOSIT (PACKET)
END

(*Else (an incoming packet]*)
ELSE

(*Pass it up to the application layer*)
TO_APP.DEPOSIT (PACKET)

(*Endif*)

(* A-18 *)
(*$$$ BLACKBOX $$*)
(*$$$ Undefined layers of the network $$*)

(*End cycle*)
END
(*End Blackbox*)
END;

(* A-19 *)
TYPE WORKER_PROCESS = PROCESS(CONSOLE: RESOURCE_MONITOR;
   FROM_NET: MAIL_BOX_MONITOR;
   TO_NET: MAIL_BOX_MONITOR);

(* The WORKER_PROCESS is an application layer process that *
  transfers remote files to the operator console. *)

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

TYPE

MESSAGE_IO_CLASS = A class that uses supervisory  
calls to handle fixed record I/O to specified  
logical units.

PACKET_TYPE = Record structure of the network  
packets.

MESSAGE_TYPE = Array of characters.

ERROR_SVC1_TYPE = Record structure of the status  
bytes from the supervisory call.

MAIL_BOX_MONITOR = A monitor used for passing  
packets between processes.

RESOURCE_MONITOR = Allows only one process to  
access a resource at a time.

INTERNAL

VAR

OP: Used to write lines of the transfered file  
to the operator.

PACKET: A network packet this process uses to  
communicate with the network.

TEXT: Array of characters used to communicate  
the operator.

OP_STATUS: Receives the status bytes from the  
MESSAGE_IO_CLASS. Not used here, but necessary  
for the calls to OP.

PARAMETERS

CONSOLE: The RESOURCE_MONITOR used to reserve the  
console for exclusive I/O.

FROM_NET: The monitor used to receive packets from  
the network.

TO_NET: The monitor used to send packets to the net.

VAR

OP: MESSAGE_IO_CLASS;
PACKET: PACKET_TYPE;
TEXT: MESSAGE_TYPE;
OP_STATUS: ERROR_SVC1_TYPE;

(* A-20 *)
(* Begin Worker process *)
BEGIN
(* Initialize the interface to the operator *)
INIT OP;
(* Cycle forever *)
CYCLE
(* Get the id for the file to be transferred *)
CONSOLE.REQUEST;
TEXT := 'ENTER FILE ID.  = ';
TEXT [18] := FIRST_FILE_ID;
OP.WRITE (TERMINAL, TEXT, OP_STATUS);
OP.READ (TERMINAL, TEXT, OP_STATUS);
CONSOLE.RELEASE;
(* Send the request to the server *)
PACKET.DIRECTION := OUTGOING; (***)
PACKET.TEXT := TEXT;
TO_NET.DEPOSIT (PACKET);
(* Transfer the file to the console *)
CONSOLE.REQUEST;
(* Repeat until end of file *)
REPEAT
(* Get a line from the network *)
FROM_NET.GET (PACKET);
(* Output it to the console *)
OP.WRITE (TERMINAL, PACKET.TEXT, OP_STATUS);
(* End repeat *)
UNTIL (PACKET.TEXT [1] = '/')
& (PACKET.TEXT [2] = '*');
CONSOLE.RELEASE
(* End cycle *)
END
(* End Worker process *)
END;
TYPE SERVER_PROCESS = PROCESS (FROM_NET: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR);

The server process is an application layer process that
does the disk I/O for a remote worker process.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

MESSAGE_IO_CLASS = A class that uses supervisory
calls to handle fixed record I/O to specified
logical units.
PACKET_TYPE = Record structure of the network
packets.
MESSAGE_TYPE = Array of characters.
ERROR_SVC1_TYPE = Record structure of the status
bytes from the supervisory call.
MAIL_BOX_MONITOR = A monitor used for passing
packets between processes.

INTERNAL

VAR

DISK: Used to input lines of a disk file.
PACKET: A network packet used to communicate with
the network.
TEXT: Array of characters used for the disk I/O.
FILE_ID: The id of the file the worker process is
requesting.
VALID_FILE_IDS: A set of the valid id's this
process can access.
FILE_LU: An array of logical units that are
subscripted with file id's. Used to look up the
logical unit of a file.
NEXT_LU: Used in initializing FILE_LU.
INDEX: Used in initializing FILE_LU.
OP_STATUS: Receives the status bytes form the
MESSAGE_IO_CLASS.

PARAMETERS

FROM_NET: The monitor used to receive packets from
the network.
TO_NET: The monitor used to send packets to the net.

VAR

DISK: MESSAGE_IO_CLASS;
PACKET: PACKET_TYPE;
TEXT: MESSAGE_TYPE;
FILE_ID: CHAR;
VALID_FILE_IDS: SET OF CHAR;
FILE_LU: ARRAY [FILE_RANGE] OF BYTE;
NEXT_LU: BYTE;
INDEX: FILE_RANGE;
OP_STATUS: ERROR_SVC1_TYPE;

(#Begin Server process#)
BEGIN
(*Initialize the interface to the disk files*)
INIT_DISK;
(*Set up an array to reference logical unit numbers*)
(*by the character id's of the files*)
NEXT_LU := TERMINAL + 1;
(*For all file id's do*)
FOR INDEX := FIRST_FILE_ID TO LAST_FILE_ID DO
   BEGIN
      (*Remember that it is a valid id*)
      VALID_FILE_IDS := VALID_FILE_IDS + [INDEX];
      (*Set it's logical unit number*)
      FILE_LU [INDEX] := NEXT_LU;
      NEXT_LU := NEXT_LU + 1;
   END;
(*Endfor*)
(*Cycle forever*)
CYCLE
(*Get the request from the net*)
FROM_NET.GET (PACKET);
FILE_ID := PACKET.TEXT [1];
(*If [it is a valid file id] then*)
IF (FILE_ID IN VALID_FILE_IDS) THEN
   (**Transfer the file**) 
   BEGIN
      (*Read in a line from the disk*)
      DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS);
      (*While not [end of file] do*)
      WHILE (OP_STATUS.DI = 0) AND (OP_STATUS.DD = 0) DO
         BEGIN
            (*Send it out on the network*)
            PACKET.DIRECTION := OUTGOING; (****)
            PACKET.TEXT := TEXT;
            TO_NET.DEPOSIT (PACKET);
            (*Read in a new line from the disk file*)
            DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS)
         END;
      (*Endwhile*)
      (*Rewind the disk file*)
      DISK.REWIND (FILE_LU [FILE_ID], OP_STATUS);
      (*Send an EOF packet out on the network*)
      PACKET.DIRECTION := OUTGOING; (****)
      PACKET.TEXT := '/
      TO_NET.DEPOSIT (PACKET)
   END
(* A-23 *)
ELSE (an invalid file id)
BEGIN
PACKET.DIRECTION := OUTGOING; (***)
PACKET.TEXT := '/* BAD FILE ID - ';
PACKET.TEXT [19] := FILE_ID;
TO_NET.DEPOSIT (PACKET)
END

Endif

End cycle

END

End Server

END;
(*$*$* CR2NL $*$*$*)

(*$*$* Presentation layer record delimiter conversion $*$*$*)

TYPE CR_NL_CLASS = CLASS;
(********************************************************************)
*   CR_NL_CLASS substitutes carriage return and new line             *
*   characters in a message.                                        *
********************************************************************)
* PROGRAMMER: RON ALBURY                                          *
* DATE WRITTEN: 4/5/82                                           *
* Copyright 1982 by Ronald C. Albury                             *
********************************************************************)
* EXTERNAL                                                      *
* CONST                                                        *
*    MESSAGE_LENGTH = Number of characters in the                *
*    packet text.                                               *
*    TYPE                                                       *
*    MESSAGE_TYPE = ARRAY [ 1..MESSAGE_LENGTH ]                 *
*    OF CHAR                                                   *
********************************************************************)
* INTERNAL                                                     *
* CONST                                                        *
*    NL = ASCII representation of a 'new line'.                 *
*    CR = ASCII representation of a 'carriage return'.          *
********************************************************************)

CONST
  CR = '(;13:);';
  NL = '(;10:);';

(*$*$*PROCEDURE ENTRY CHANGE$*$*)

* VARIABLES
  INDEX: Used to increment through the message.
  OLD_DELIM: The delimiter we wish to change.
  NEW_DELIM: The delimiter to change to.
********************************************************************)
* PARAMETERS
  IN
    HOST_FILE_FORMAT: The format the file needs to
    be converted to.
  OUT
    TEXT: The array of characters to be converted.
********************************************************************)

PROCEDURE ENTRY CHANGE ( VAR TEXT: MESSAGE_TYPE;
  HOST_FORM: FILE_FORMAT_TYPE );

VAR
  INDEX: 1..MESSAGE_LENGTH;
  OLD_DELIM, NEW_DELIM: CHAR;
(*Begin entry Change*)
BEGIN
(* Decide which characters need to be changed*)
  IF ( HOST_FORM = CR_DELIM ) THEN
    BEGIN
      OLD_DELIM := NL;
      NEW_DELIM := CR
    END

(* A-25 *)
BEGIN (*CLASS_INITIALIZATION*)
END;

ELSE
BEGIN
OLD_DELIM := CR;
NEW_DELIM := NL
END;
{ENDIF}

(*Change all incorrect characters*)
FOR INDEX := 1 TO MESSAGE_LENGTH DO
  IF ( TEXT [INDEX] = OLD_DELIM ) THEN
    TEXT [INDEX] := NEW_DELIM
  {ENDIF}
{ENDFOR}

(*End entry Change*)
END;

BEGIN (*CLASS_INITIALIZATION*)
END;

(* A-26 *)
TYPE CRIPR_CLASS = CLASS;

{*** VIGENERE SUBSTITUTION CIPHER ***}

(**************************************************************************
  * CRIPR_CLASS uses cryptographic methods to provide
  * security in data transfer.
  **************************************************************************
  * PROGRAMMER: RON ALBURY
  * DATE WRITTEN: 4/2/82
  * Copyright 1982 by Ronald C. Albury
  **************************************************************************
  * EXTERNAL
  *  
  * CONST
  *    MESSAGE_LENGTH = The length of the message array.
  *    TYPE
  *      MESSAGE_TYPE = Array of characters.
  **************************************************************************
  * INTERNAL
  *  
  * CONST
  *    FIRST_CHAR = The first character in the ASCII
  *      character set (null).
  *    LAST_CHAR = The last character in the ASCII
  *      character set.
  *    TYPE
  *      MSG_SYMBOLS = Subrange of acceptable symbols in
  *        a message.
  *      CRIPR_TABLE = A two dimensional array for all
  *        acceptable symbols in a message.
  *    VAR
  *      ROW_ORD, COL_ORD: Integers used to calculate the
  *        characters in the CRIPR_TABLEs during
  *        initialization.
  *      ROW_INDEX, COL_INDEX: MSG_SYMBOLS to increment
  *        through the CRIPR_TABLEs at initialization.
  *      ENCRYPT, DECRYPT: CRIPR_TABLEs used to look up
  *        character substitutions for encryption and
  *        decryption.
  *      SPAN: Used with a MOD function to 'fold' the
  *        MSG_SYMBOLS around during initialization of the
  *        CRIPR_TABLEs.
  *      ORD_FIRST_CHAR, ORD_LAST_CHAR: Used to set SPAN.
  *      MSG_SYM_SET: Set of acceptable symbols in a
  *        message.
  **************************************************************************

CONST
  FIRST_CHAR = '(:0:)';
  LAST_CHAR = '(:127:)';

TYPE
  MSG_SYMBOLS = FIRST_CHAR..LAST_CHAR;
  CRIPR_TABLE = ARRAY [MSG_SYMBOLS, MSG_SYMBOLS] OF CHAR;

VAR
  ROW_ORD, COL_ORD: INTEGER;
ROW_INDEX, COL_INDEX: MSG_SYMBOLS;
ENCRIPT, DECRYPT: CRIPT_TABLE;
ORD_FIRST_CHAR, ORD_LAST_CHAR, SPAN: 0..128;
MSG_SYM_SET: SET OF MSG_SYMBOLS;

(***PROCEDURE ENTRY ENCODE*******************************
   VARIABLES
   * MSG_INDEX: Index to increment through the message.
   * KEY_INDEX: Index to increment through the key.
   * PARAMETERS
   * IN
   * KEY: The array of characters used as the key for
   * encrypting the message.
   * OUT
   * MESSAGE: The array of characters to be encrypted.
   ******************************************)
PROCEDURE ENTRY ENCODE ( VAR MESSAGE: MESSAGE_TYPE;
                           KEY: MESSAGE_TYPE );

VAR
   MSG_INDEX: 1..MESSAGE_LENGTH;
   KEY_INDEX: 1..MESSAGE_LENGTH;

(*Begin entry Encode*)
BEGIN
(*For all the characters in a message do*)
FOR MSG_INDEX := 1 TO MESSAGE_LENGTH DO
BEGIN
(*Calculate which letter in the key to use*)
(*In case the key isn't the same length as a message*)
   KEY_INDEX := ( (MSG_INDEX-1) MOD MESSAGE_LENGTH ) + 1;
(*If [unable to translate this character then]*)
   IF NOT ( MESSAGE [MSG_INDEX] IN MSG_SYM_SET ) THEN
      (*Arbitrarily encode it as Last_char*)
      MESSAGE [MSG_INDEX] := LAST_CHAR
(*Else (a good character]*)
   ELSE
      (*Look up the new value in the Encrypt table*)
      MESSAGE [MSG_INDEX] :=
         ENCRYPT [KEY [KEY_INDEX], MESSAGE [MSG_INDEX]]
   (*Endif*)
END
(*Endfor*)
(*End entry Encode*)
END;

(***PROCEDURE ENTRY DECODE*******************************
   VARIABLES
   * MSG_INDEX: Index to increment through the message.
   * KEY_INDEX: index to increment through the key.
   * PARAMETERS
   * IN
   * KEY: The array of characters used as the key for
   * encrypting the message.
   ******************************************)
PROCEDURE ENTRY DECRYPT ( VAR MESSAGE: MESSAGE_TYPE;
                           KEY: MESSAGE_TYPE );

VAR
   MSG_INDEX: 1..MESSAGE_LENGTH;
   KEY_INDEX: 1..MESSAGE_LENGTH;

(*Begin entry Decrypt*)
BEGIN
(*For all the characters in a message do*)
FOR MSG_INDEX := 1 TO MESSAGE_LENGTH DO
BEGIN
(*Calculate which letter in the key to use*)
(*In case the key isn't the same length as a message*)
   KEY_INDEX := ( (MSG_INDEX-1) MOD MESSAGE_LENGTH ) + 1;
(*If [unable to translate this character then]*)
   IF NOT ( MESSAGE [MSG_INDEX] IN MSG_SYM_SET ) THEN
      (*Arbitrarily encode it as Last_char*)
      MESSAGE [MSG_INDEX] := LAST_CHAR
(*Else (a good character]*)
   ELSE
      (*Look up the new value in the Decrypt table*)
      MESSAGE [MSG_INDEX] :=
         DECRYPT [KEY [KEY_INDEX], MESSAGE [MSG_INDEX]]
   (*Endif*)
END
(*Endfor*)
(*End entry Decrypt*)
END;
(* $$$ CRIPTV $$$ *)
(* $$$ Presentation layer data encryption $$$ *)

* decripting the message. *

* OUT *
* MESSAGE: The array of characters to be decripted. *
* ************************************************************

PROCEDURE ENTRY DECODE ( VAR MESSAGE: MESSAGE_TYPE;
                         KEY: MESSAGE_TYPE);

VAR
  MSG_INDEX: 1..MESSAGE_LENGTH;
  KEY_INDEX: 1..MESSAGE_LENGTH;

(*Begin entry Decode*)
BEGIN

(*For all the characters in a message do*)
FOR MSG_INDEX := 1 TO MESSAGE_LENGTH DO
  BEGIN
    (*Calculate which letter in the key to use (in case*)
    (*the key isn't the same length as the message]*)
    KEY_INDEX := ( (MSG_INDEX-1) MOD MESSAGE_LENGTH )
    + 1;
    (*If [unable to translate this character] then]*)
    IF NOT ( MESSAGE [MSG_INDEX ] IN MSG_SYM_SET ) THEN
      (*Arbitrarily decode it as Last_char*)
      MESSAGE [MSG_INDEX] := LAST_CHAR
    (*Else (a good character]*)
    ELSE
      (*Look up the new value in the Decript table]*)
      MESSAGE [MSG_INDEX] :=
        DECRYPT [KEY [KEY_INDEX], MESSAGE [MSG_INDEX]]
    (*Endif*)
  END
(*Endfor*)
(*End entry Decode*)
END;

(*The method of initializing the Encript and Decript*)
(*tables allows for maximum flexibility if*)
(*you decide to change the set of acceptable message symbols*)
(*A section of the initialized encript table contains:*)
(*@ ::::::::::@*)
(*@ ABCD...@*)
(*@ BCD...@*)
(*@ CDEFG...@*)
(*@ EFGHI...@*)
(*@ ::::::::::@*)

(*Begin Cript_class initialization*)
BEGIN
  MSG_SYM_SET := [ ];
  ORD_FIRST_CHAR := ORD( FIRST_CHAR );
  ORD_LAST_CHAR := ORD( LAST_CHAR );
  SPAN := ORD_LAST_CHAR - ORD_FIRST_CHAR + 1;
  FOR ROW_INDEX := FIRST_CHAR TO LAST_CHAR DO
    BEGIN
      ROW_ORD := ORD(ROW_INDEX);
  END
  END;

(* A-29 *)
MSG_SYM_SET := MSG_SYM_SET + [ROW_INDEX];
FOR COL_INDEX := FIRST_CHAR TO LAST_CHAR DO
    BEGIN
        COL_ORD := ORD(COL_INDEX);
        ENCRPT [COL_INDEX, ROW_INDEX] :=
            CHR ( ( (ROW_ORD+COL_ORD) MOD SPAN )
                 + ORD_FIRST_CHAR );
        DECRPT [COL_INDEX, ROW_INDEX] :=
            CHR ( ( (ROW_ORD+SPAN-COL_ORD) MOD SPAN )
                 + ORD_FIRST_CHAR );
    END;
END
{ENDFOR EACH COLUMN}
END
{ENDFOR EACH ROW}
(*End Cript_class initialization*)
END;
TYPE PRESENT_PROCESS = PROCESS (TO_APP: MAIL_BOX_MONITOR;
EVENT: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR;
HOST_FORM: FILE_FORMAT_TYPE);

The PRESENTATION layer handles such tasks as encryption
and file format modification, before the packets are
presented to the application layer.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/29/82
Copyright 1982 by Ronald C. Albury

EXTERNAL

TYPE
PACKET_TYPE: Record structure of the network
packets.
MAIL_BOX_MONITOR: A monitor used for passing
packets between processes.
FILE_FORMAT_TYPE: If the file uses carriage return
or new line as a delimiter.
CRPT_CLASS: A class which translates messages
into or out of a cipher.
CR_NL_CLASS: A class which changes the format of
messages between carriage return and new line
delimiters.
MESSAGE_TYPE: Array of characters.

INTERNAL

VAR
CIPHER: Handles encryption for the layer.
FORMAT: Handles changing the format of messages
for the layer.
PACKET: A network packet that the layer processes.
KEY: The cipher key used by CIPHER.

PARAMETERS
TO_APP: The monitor used to send packets to the
application layer.
EVENT: The monitor this layer uses to receive
packets.
TO_NET: The monitor used to send packets down to the
network.
HOST_FORM: The file format this host uses.

VAR
CIPHER: CRPT_CLASS;
FORMAT: CR_NL_CLASS;
PACKET: PACKET_TYPE;
KEY: MESSAGE_TYPE;
(### PRESENT ###)
(### Process to simulate the Presentation layer ###)

(*Begin Present_process*)
BEGIN
(*Initialize the encryption and format routines*)
INIT CIPHER, FORMAT;
(*Set the encryption key*)
KEY := 'TEMPORARY KEY &l)
(*Cycle forever*)
CYCLE
(*Wait for a packet*)
EVENT.GET (PACKET);
(*If [packet is heading out] then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
BEGIN
(*If [the security level is secret] then*)
IF (PACKET.SECURITY = SECRET) THEN
(*Encode the text*)
CIPHER.ENCODE (PACKET.TEXT, KEY);
(Endif*)
(*Identify the file format of the text*)
PACKET.FILE_FORMAT := HOST_FORM;
(*Send the packet out on the network*)
TO_NET.DEPOSIT (PACKET)
END
(*Else (packet on it's way to application]*)
ELSE
BEGIN
(*If [the packet is encrypted] then*)
IF (PACKET.SECURITY = SECRET) THEN
BEGIN
(*Decode it*)
CIPHER.DECODE (PACKET.TEXT, KEY);
PACKET.SECURITY := PUBLIC
END;
(Endif*)
(*If [wrong record delimiter] then*)
IF (PACKET.FILE_FORMAT <> HOST_FORM) THEN
BEGIN
(*Modify the format to match the host*)
FORMAT.CHANGE (PACKET.TEXT, HOST_FORM);
PACKET.FILE_FORMAT := HOST_FORM
END;
(Endif*)
(*Pass the packet up to the application layer*)
TO_APP.DEPOSIT (PACKET)
END
(End cycle*)
END
(*End Present_process*)
END;

(* A-32 *)
TYPE WORKER_PROCESS = PROCESS(CONSOLE: RESOURCE_MONITOR;
FROM_NET: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR);

(* The WORKER_PROCESS is an application layer process that
  transfers remote files to the operator console.
*)
PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

TYPE
MESSAGE_IO_CLASS = A class that uses supervisory
calls to handle fixed record I/O to specified
logical units.
PACKET_TYPE = Record structure of the network
packets.
MESSAGE_TYPE = Array of characters.
ERROR_SVC1_TYPE = Record structure of the status
bytes from the supervisory call.
MAIL_BOX_MONITOR = A monitor used for passing
packets between processes.
RESOURCE_MONITOR = Allows only one process to
access a resource at a time.

INTERNAL

VAR
OP: Used to write lines of the transferred file
to the operator.
PACKET: A network packet this process uses to
communicate with the network.
TEXT: Array of characters used to communicate
the operator.
OP_STATUS: Receives the status bytes from the
MESSAGE_IO_CLASS. Not used here, but necessary
for the calls to OP.

PARAMETERS
CONSOLE: The RESOURCE_MONITOR used to reserve the
console for exclusive I/O.
FROM_NET: The monitor used to recieve packets from
the network.
TO_NET: The monitor used to send packets to the net

VAR
OP: MESSAGE_IO_CLASS;
PACKET: PACKET_TYPE;
TEXT: MESSAGE_TYPE;
OP_STATUS: ERROR_SVC1_TYPE;
(*Begin Worker process*)
BEGIN
(*Initialize the interface to the operator*)
INIT OP;
(*Cycle forever*)
CYCLE
(*Get the id for the file to be transferred*)
CONSOLE.REQUEST;
TEXT := 'ENTER FILE ID. - ';
TEXT [18] := FIRST_FILE_ID;
OP.WRITE (TERMINAL, TEXT, OP_STATUS);
OP.READ (TERMINAL, TEXT, OP_STATUS);
CONSOLE.RELEASE;
(*Send the request to the server*)
PACKET.DIRECTION := OUTGOING;
PACKET.SECURITY := PUBLIC; (***)
PACKET.TEXT := TEXT;
TO_NET.DEPOSIT (PACKET);
(*Transfer the file to the console*)
CONSOLE.REQUEST;
(*Repeat until end of file*)
REPEAT
 (*Get a line from the network*)
FROM_NET.GET (PACKET);
 (*Output it to the console*)
OP.WRITE (TERMINAL, PACKET.TEXT, OP_STATUS);
(*End repeat*)
UNTIL (PACKET.TEXT [1] = '/')
& (PACKET.TEXT [2] = '#');
CONSOLE.RELEASE
(*End cycle*)
END
(*End Worker process*)
END;
TYPE SERVER_PROCESS = PROCESS (FROM_NET: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR);

******************************************************************************
* The server process is an application layer process that
* does the disk I/O for a remote worker process.
******************************************************************************

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

********************************************************************************
* EXTERNAL
* TYPE
* MESSAGE_IO_CLASS = A class that uses supervisory
* calls to handle fixed record I/O to specified
* logical units.
* PACKET_TYPE = Record structure of the network
* packets.
* MESSAGE_TYPE = Array of characters.
* ERROR_SVC_TYPE = Record structure of the status
* bytes from the supervisory call.
* MAIL_BOX_MONITOR = A monitor used for passing
* packets between processes.

********************************************************************************
* INTERNAL
* VAR
* DISK: Used to input lines of a disk file.
* PACKET: A network packet used to communicate with
* the network.
* TEXT: Array of characters used for the disk I/O.
* FILE_ID: The id of the file the worker process is
* requesting.
* VALID_FILE_IDS: A set of the valid id's this
* process can access.
* FILE_LU: An array of logical units that are
* subscripted with file id's. Used to look up the
* logical unit of a file.
* NEXT_LU: Used in initializing FILE_LU.
* INDEX: Used in initializing FILE_LU.
* OP_STATUS: Receives the status bytes form the
* MESSAGE_IO_CLASS.

********************************************************************************
* PARAMETERS
* FROM_NET: The monitor used to recieve packets from
* the network.
* TO_NET: The monitor used to send packets to the net.
********************************************************************************

VAR
DISK: MESSAGE_IO_CLASS;
PACKEt: PACKET_TYPE;
TEXT: MESSAGE_TYPE;
FILE_ID: CHAR;

(* A-35 *)
VALID_FILE_IDS: SET OF CHAR;
FILE_LU: ARRAY [FILE_RANGE] OF BYTE;
NEXT_LU: BYTE;
INDEX: FILE_RANGE;
OP_STATUS: ERROR_SVC1_TYPE;

(*Begin Server process*)
BEGIN

(*Initialize the interface to the disk files*)
INIT DISK;

(*Set up an array to reference logical unit numbers*)
(*by the character id's of the files*)
NEXT_LU := TERMINAL + 1;

(*For all file id's do*)
FOR INDEX := FIRST_FILE_ID TO LAST_FILE_ID DO
BEGIN

(*Remember that it is a valid id*)
VALID_FILE_IDS := VALID_FILE_IDS + [INDEX];

(*Set it's logical unit number*)
FILE_LU [INDEX] := NEXT_LU;
NEXT_LU := NEXT_LU + 1
END;

(*Endfor*)

(*Cycle forever*)
CYCLE

(*Get the request from the net*)
FROM_NET.GET (PACKET);
FILE_ID := PACKET.TEXT [1];

(*If [it is a valid file id] then*)
IF (FILE_ID IN VALID_FILE_IDS) THEN

(*Transfer the file*)
BEGIN

(*Read in a line from the disk*)
DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS);

(*While not [end of file] do*)
WHILE (OP_STATUS.DI = 0) AND (OP_STATUS.DD = 0) DO
BEGIN

(*Send it out on the network*)
PACKET.DIRECTION := OUTGOING;
PACKET.SECURITY := SECRET; (****)
PACKET.TEXT := TEXT;
TO_NET.DEPOSIT (PACKET);

(*Read in a new line from the disk file*)
DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS)
END;

(*Endwhile*)

(*Rewind the disk file*)
DISK.REWIND (FILE_LU [FILE_ID], OP_STATUS);

(*Send an EOF packet out on the network*)
PACKET.DIRECTION := OUTGOING;
PACKET.SECURITY := SECRET; (****)
PACKET.TEXT := '/

(* A-36 *)
(### SERVER2 ###)
(### Net2 Server application process ###)

TO_NET.DEPOSIT (PACKET)
END
(Else (an invalid file id))
ELSE
(Send an error message)
BEGIN
PACKET.DIRECTION := OUTGOING;
PACKET.SECURITY := PUBLIC; (###)
PACKET.TEXT := '/# BAD FILE ID = ';
PACKET.TEXT [19] := FILE_ID;
TO_NET.DEPOSIT (PACKET)
END
(End)
(End cycle)
END
(End Server)
END;
TYPE MAIL_BOX_MONITOR = MONITOR;
(**********************************************************************
 * MAIL_BOX_MONITOR is simply a means for one process to * 
 * receive prioritized messages from up to MAX_SENDER other * 
 * processes. * 
 * It can store up to MAX_MAIL messages in each of it's * 
 * FIFO controlled MAIL_BUFFERS. * 
 * If the receiver process attempts to pick up mail when * 
 * the buffers are empty, it is delayed until a sender * 
 * process deposits mail. * 
 * If a sender process attempts to deposit mail when * 
 * that priority's buffer is full, it is delayed until the * 
 * receiver process picks up mail. * 
 **********************************************************************
 
 PROGRAMMER: RONALD C. ALBURY
 
 DATE WRITTEN: 3/25/82
 
 COMPUTER: INTERDATA 8/32
 
 COPYRIGHT 1982 BY RONALD C. ALBURY

**********************************************************************
* EXTERNAL *
* CONST *
* MAX_SENDERS = Maximum number of processes that *
* will send messages to the receiver. *
* MAX_MAIL = Maximum number of messages the monitor *
* can hold in one priority's buffer. *
* TYPE *
* FIFO = Modified Brinch Hansen FIFO class to *
* handle a FIFO buffer. *
* PACKET_TYPE = The record structure of the mail. *
* PRIORITY_TYPE = Enumerations of the various *
* priorities a packet can have. *

**********************************************************************
* INTERNAL *
* CONST - NONE *
* TYPE _ NONE *
* VAR *
* RECEIVER: Queue variable to delay the receiver. *
* SENDER: Array of Queue variables used as a fifo *
* buffer for delaying senders. *
* DELAYED_SENDERS: Fifos to control SENDER buffers. *
* MAIL_BUFFER: Array of packets used as a *
* fifo / priority buffer for storing mail. *
* NEXT_MAIL: Fifos to control MAIL_BUFFER. *
* INDEX: Used to initialize FIFO's. *

**********************************************************************
VAR
DELEYED_SENDERS: ARRAY [PRIORITY_TYPE] OF FIFO;
NEXT_MAIL: ARRAY [PRIORITY_TYPE] OF FIFO;
MAIL_BUFFER: ARRAY [1..MAX_MAIL, PRIORITY_TYPE] OF PACKET_TYPE;
RECEIVER: QUEUE;
SENDER: ARRAY [1..MAX_SENDERS, PRIORITY_TYPE] OF QUEUE;

( # A-38 #)
INDEX: PRIORITY_TYPE;

(*PROCEDURE ENTRY GET*********************************************************************)
* INTERNAL
  * VAR
    * NEXT: Used to simplify code. Receives the index
    * of the next entry in the MAIL_BUFFER to go.
    * ALL_EMPTY: Boolean variable to show if all
    * of the various priorities buffers' are empty.
    * PRI: Loop variable used to set ALL_EMPTY and
    * NEXT.
  *
*----------------------------------------------------------------------*
* PARAMETERS
  *
  * OUT
    * OUTGOING_MAIL: Receives the oldest/highest
    * priority entry in the MAIL_BUFFER.
  *----------------------------------------------------------------------*
PROCEDURE ENTRY GET ( VAR OUTGOING_MAIL: PACKET_TYPE );
VAR
  * NEXT := 1..MAX_MAIL;
  * ALL_EMPTY: BOOLEAN;
  * PRI: PRIORITY_TYPE;
(*Begin entry GET*)
BEGIN
  (*Check if we have mail*)
  ALL_EMPTY := TRUE; (***)
  FOR PRI := LOW_PRI TO HIGH_PRI DO
    ALL_EMPTY := ALL_EMPTY AND NEXT_MAIL [PRI].EMPTY;
  {Endfor}
  (*If [there is no mail] then*)
  IF ( ALL_EMPTY ) THEN
    (*Put the receiver to sleep*)
    DELAY ( RECEIVER );
  (*Endif*)
  (*Set outgoing mail to the oldest packet in the queue*)
  (*with the highest priority*)
  (*We are guaranteed to have mail at this point.*)
  PRI := HIGH_PRI; (***)
  WHILE (NEXT_MAIL [PRI].EMPTY) DO
    PRI := PRED (PRI );
  {Endwhile}
  NEXT := NEXT_MAIL [PRI].DEPARTURE;
  OUTGOING_MAIL := MAIL_BUFFER [NEXT, PRI];
  (*If [there are senders sleeping] then*)
  IF ( DELAYED_SENDERS [PRI].OCCUPIED ) THEN
    (*Wake up the oldest sleeper*)
    CONTINUE (SENDER [DELAYED_SENDERS [PRI].DEPARTURE, PRI]);
  (*Endif*)
(*End entry GET*)
END;
(* A-39 *)
(### MAILBOX3 ###)
(### Prioritized communication mailbox ###)

(###PROCEDURE ENTRY DEPOSIT###)

* INTERNAL
* VAR
  * PRI: Used to simplify code. Receives the
  * priority of the incoming mail.

* PARAMETERS
  * IN
  * INCOMING_MAIL: A packet being deposited into the
  * MAIL_BUFFER by a sender process.

PROCEDURE ENTRY DEPOSIT ( INCOMING_MAIL: PACKET_TYPE );
VAR
  PRI: PRIORITY_TYPE;
BEGIN
  PRI := INCOMING_MAIL.PRIORITY;
  (*If [all known senders are delayed] then*)
  IF ( DELAYED_SENDERS [PRI].FULL ) THEN
    (*Should never happen unless MAX_SENDER is wrong*)
    (*We loose the mail*)
  (*Else*)
  ELSE
    (*If [mail queue is full] then*)
    IF ( NEXT_MAIL [PRI].FULL ) THEN
      (*Put the sender to sleep*)
      DELAY ( SENDER [DELAYED_SENDERS [PRI].ARRIVAL, PRI]);
    (*Endif*)
    (*Store the mail in a FIFO queue*)
    MAIL_BUFFER [NEXT_MAIL [PRI].ARRIVAL, PRI ] := INCOMING_MAIL:
    (*If [the receiver is sleeping] wake him up*)
    CONTINUE ( RECEIVER )
  END
  (*Endif*)
(*End entry DEPOSIT*)
END;

BEGIN (*MONITOR INITIALIZATION*)
FOR INDEX := LOW_PRI TO HIGH_PRI DO
  INIT NEXT_MAIL [INDEX] (MAX_MAIL),
  DELAYED_SENDERS [INDEX] (MAX_SENDERS)
[*Endfor]
END; (*MAIL_BOX_MONITOR*)

(* A-40 *)
TYPE NET_IO_CLASS = CLASS (FROM_NET: MAIL_BOX_MONITOR;
               TO_NET: MAIL_BOX_MONITOR);

(* NET_IO_CLASS provides standard entry points for
  interfacing with the network layers. Allows creation
  and destruction of sessions, and data transferal. *)

*** Programmed: Ronald C. Albury
*** Date Written: 10/08/82
*** Computer: Interdata 8/32
*** Copyright 1982 by Ronald C. Albury

EXTERNAL
  TYPE
  MAIL_BOX_MONITOR: Interprocess communication
  mailboxes.
  PACKET_TYPE: Record structure of the network
  packets.
  MESSAGE_TYPE: Array of characters.
  PKT_STATUS_TYPE: A field of the Packet_type
  record, for error flags.
  CHAIN_TYPE: The subrange of session commands
  dealing with chaining messages.

INTERNAL
  VAR
  PACKET: The network packet that is assembled
           and passed to the lower layers.

VAR PACKET: PACKET_TYPE;

PROCEDURE ENTRY NET_LISTEN
PARAMETERS
OUT
STATUS: Error flags indicating the status of the
        listen.

PROCEDURE ENTRY_NET_LISTEN (VAR STATUS: PKT_STATUS_TYPE);
(*Begin entry Net_listen*)
BEGIN
  (*Set the network parameters for a listen*)
  PACKET.TEXT := ' ';
  PACKET.SECURITY := PUBLIC;
  PACKET.DIRECTION := OUTGOING;
  PACKET.PRIORITY := HIGH_PRI;
  PACKET.SESSION_CMD := LISTEN;
  (*Issue the listen to the network and wait for*)
  (*a response*)
  TO_NET.DEPOSIT (PACKET);
  FROM_NET.GET (PACKET);
  (*Set the status flags*)

(* A-41 *)
STATUS := PACKET.STATUS
(*End entry Net_listen*)
END;

(****PROCEDURE ENTRY MAKE_SESSION********************
 * PARAMETERS
 * IN
 * PASSWORD: Array of characters containing the
 * the password to be sent to the destination
 * process when the session request is issued.
 * OUT
 * STATUS: Error flags indicating the status of the
 * session request.
******************************************************************************)
PROCEDURE ENTRY MAKE_SESSION ( PASSWORD: MESSAGE_TYPE;
VAR STATUS: PKT_STATUS_TYPE);
(*Begin entry Make_session*)
BEGIN
(*Set the network parameters for a session request*)
PACKET.TEXT := PASSWORD;
PACKET.SECURITY := PUBLIC;
PACKET.DIRECTION := OUTGOING;
PACKET.STATUS := [];
PACKET.SESSION_CMD := ESTABLISH;
PACKET.PRIORITY := HIGH_PRI;
(*Issue the session request to the network and wait*)
(*for a response*)
TO_NET.DEPOSIT (PACKET);
FROM_NET.GET (PACKET);
(*Set the status flags*)
STATUS := PACKET.STATUS
(*End entry Make_session*)
END;

(****PROCEDURE ENTRY CLEAR_SESSION*******************************
 * PARAMETERS
 * OUT
 * STATUS: Error flags indicating the status of the
 * clear.
******************************************************************************)
PROCEDURE ENTRY CLEAR_SESSION (VAR STATUS: PKT_STATUS_TYPE);
(*Begin entry Clear_session*)
BEGIN
(*Set the network parameters for a clear*)
PACKET.TEXT := ' ';
PACKET.SECURITY := PUBLIC;
PACKET.DIRECTION := OUTGOING;
PACKET.STATUS := [];
PACKET.SESSION_CMD := BREAK;
(*Make sure it arrives after outstanding messages*)
PACKET.PRIORITY := LOW_PRI;
(*Issue the clear to the network and wait for a*)
(* A-42 *)
(*$*$ NETIO $*$*)

(*$*$ Standard entries to the network $*$*)

(*response*)
TO_NET.DEPOSIT (PACKET);
FROM_NET.GET (PACKET);
(*Set the status flags*)
STATUS := PACKET.STATUS
(*End entry Clear_session*)
END;

(*$*$PROCEDURE ENTRY NET_READ______________________________

* PARAMETERS
  *
  * OUT
  *
  * TEXT: Array of characters to receive the message
    * from the network.
  *
  * STATUS: Error flags indicating the status of the
    * read.

***************************************************************************)

PROCEDURE ENTRY NET_READ (VAR TEXT: MESSAGE_TYPE;
VAR STATUS: PKT.STATUS_TYPE);

(*Begin entry Net_read*)
BEGIN
(*Get the packet from the network*)
FROM_NET.GET (PACKET);
(*Extract the Text and Status flags from the packet*)
TEXT := PACKET.TEXT;
STATUS := PACKET.STATUS
(*End entry Net_read*)
END;

(*$*$PROCEDURE ENTRY NET_WRITE______________________________

* PARAMETERS
  *
  * IN
  *
  * TEXT: Array of characters to send on the network.
  *
  * XFER: The session layer chain command for this
    * data transfer.
  *
  * SECURITY: The security level desired for this
    * data item.
  *
  * OUT
  *
  * STATUS: Error flags indicating the status of the
    * write.

***************************************************************************)

PROCEDURE ENTRY NET_WRITE (TEXT: MESSAGE_TYPE;
XFER: CHAIN_TYPE;
SECURITY: SECURITY_TYPE;
VAR STATUS: PKT.STATUS_TYPE);

(*Begin entry Net_write*)
BEGIN
(*Set the network parameters for a write.*)
PACKET.TEXT := TEXT;
PACKET.SECURITY := SECURITY;
PACKET.DIRECTION := OUTGOING;
PACKET.STATUS := [];
PACKET.SESSION_CMD := XFER;

(* A-43 *)
PACKET.PRIORITY := MED_PRI;
(*Issue the write to the net and wait for a response*)
TO_NET.DEPOSIT (PACKET);
FROM_NET.GET (PACKET);
(*Set the status flags*)
STATUS := PACKET.STATUS
(*End entry Net_write*)
END;

BEGIN
END;

(* A-44 *)
TYPE ERROR_CLASS = CLASS (CONSOLE: RESOURCE_MONITOR);

* Error_class interprets and displays network errors to
  the terminal.

Programmer: Ronald C. Albury
Date Written: 10/02/82
Copyright 1982 by Ronald C. Albury

EXTERNAL

CONST

First_Error: The network error with the lowest
  ordinal value.
Last_Error: The network error with the highest
  ordinal value.

TYPE

Resource_Monitor: Standard Brinch Hansen Resource
to control access to the terminal.
Message_IO_Class: A class that provides fixed
  record I/O to specified logical units.
Message_Type: Array of characters.
_pkt_Status_Type: Set of possible errors that can
  be encountered in the network.
Status_Type: Enumerations of network errors.
Error_Svc1_Type: Record structure of the status
  bytes returned by Message_IO_Class.

INTERNAL

VAR

Op: An instance of Message_IO_Class to handle
  I/O to the terminal.

PARAMETERS

Console: A resource monitor to assure no other
  other process is using the terminal while the
  error messages are being displayed.

VAR

OP: MESSAGE_IO_CLASS;

(***Procedure Entry Report**************************

INTERNAL

VAR

STATUS_INDEX: Loop variable for testing which
  errors are in the status word.
OP_STATUS: Receives the status bytes from the
  terminal I/O.

PARAMETERS

IN

LOCATION: A string of characters indicating where
  the class is being called from.

(* A-45 *)
(*$%%%% ERROR $%%%%*)
(*$%%%% Class for reporting network errors $%%%%*)

* STATUS: The status word to be checked for error *
* flags. *
******************************************************************************
PROCEDURE ENTRY REPORT (LOCATION: MESSAGE_TYPE;
 STATUS: PKT_STATUS_TYPE);

VAR STATUS_INDEX: STATUS_TYPE;
 OP_STATUS: ERROR_SVC1_TYPE;

(*Begin entry Report*)
BEGIN
 (*If [there are error flags in the status word] then*)
 IF (STATUS <> []) THEN
 BEGIN
 (*Request the console and display Location*)
  CONSOLE.REQUEST;
  OP.WRITE (TERMINAL, LOCATION, OP_STATUS);
 (*Display all errors in the status word*)
  FOR STATUS_INDEX := FIRST_ERROR TO LAST_ERROR DO
   IF (STATUS_INDEX IN STATUS) THEN
    OP.WRITE (TERMINAL, ERROR_MSG [STATUS_INDEX],
     OP_STATUS);
  {Endif}
 {Endfor}
 CONSOLE.RELEASE
 END
 (*ENDIF*)
(*End entry Report*)
END;

BEGIN
 INIT OP
END;
TYPE SESSION_PROCESS = PROCESS (TO_APP: MAIL_BOX_MONITOR;
EVENT: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR;
HOST_ID: HOST_ID_TYPE);

The session layer handles the initial set up of a
session between two hosts. It also is capable of
chaining a group of related messages together to
make sure that, in the event of network failure,
the reciever is not in the middle of a transmission.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 4/2/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

CONST
PASSWORD: A structured constant containing the
passwords of the hosts on the network.
MAX_SESSION_WAIT: The maximum number of pending
requests the host is allowed to queue up.
MAX_CHAIN: The maximum number of packets the
layer can chain before delivering them to the
application layer.

TYPE
PACKET_TYPE: Record structure of the network
packet.
MAIL_BOX_MONITOR: A monitor used for passing
packets between processes.
HOST_ID_TYPE: Enumeration of hosts in network.
MESSAGE_TYPE: Array of characters.
FIFO: Modified Brinch Hansen FIFO class to
handle a fifo buffer.

INTERNAL

TYPE
SESSION_STATE_TYPE: Enumeration of the states
a session layer can be in.

VAR
PASS_WORD: Contains this host's password. Used
to check if incoming requests have access
rights to this host.
PACKET: The network packet this process uses to
communicate with.
WAIT_BUFF: A fifo controlled buffer used to
store waiting request packets.
WAITING_REQ: A FIFO class to control WAIT_BUFF.
CHAIN_BUFF: An array of packets used to store
chained packets so they can be delivered to
application layer as a group.
CHAIN_PTR: Integer used to control CHAIN_BUFF.
(* $$$$  SESSION  $$$$*)
(* $$$$  Process to simulate Session layer $$$$*)

* CHAIN_INDEX: Integer used to increment through
  the CHAIN_BUFF when delivering the packets .
* STATE: The current state of the session layer.

*:PARAMETERS*
* TO_APP: The monitor used to send packets to the
  application layer.
* EVENT: The monitor this layer uses to receive
  packets.
* TO_NET: The monitor used to send packets to the
  network.
* HOST_ID: The id given to this host when the network
  is brought up.

*:----------------------------------------------------------------------------------*

TYPE
  SESSION_STATE_TYPE = (NO_SESSION, LISTENING, REQUESTING,
                        IN_SESSION);

VAR
  PASS_WORD: MESSAGE_TYPE;
  PACKET: PACKET_TYPE;
  CHAIN_INDEX: INTEGER;
  CHAIN_PTR: INTEGER;
  CHAIN_BUFF: ARRAY [1..MAX_CHAIN] OF PACKET_TYPE;
  STATE: SESSION_STATE_TYPE;

(*Begin Session_process*)
BEGIN
(*Initialize the host's password*)
  PASS_WORD := PASSWORD [HOST_ID];
(*Initialize the state to No_session*)
  STATE := NO_SESSION;
(*Cycle forever*)
CYCLE
  EVENT.GET (PACKET);
(*Process the packet based on current state*)
  CASE STATE OF
    (*When [State = No_session]*)
      NO_SESSION:
      (*If [packet is from application layer] then*)
      IF (PACKET.DIRECTION = OUTGOING) THEN
      CASE PACKET.SESSION_CMD OF
        (*When [Cmd = Listen]*)
        LISTEN:
          (*Go into Listening state*)
          STATE := LISTENING;
        (*When [Cmd = Establish]*)
        ESTABLISH:
          BEGIN
            (*Go into Requesting state*)
            STATE := REQUESTING;
            (*Issue the request*)
            (* A-48 *)
PACKET.SESSION_CMD := REQUEST;
TO_NET.DEPOSIT (PACKET)
END;
(*Otherwise*)
ELSE:
BEGIN
(*Notify application layer of illegal*)
(*command*)
PACKET.STATUS := PACKET.STATUS
+ [NO_LOCAL_SESSION];
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END
END(*CASE*)
(*Else (packet from the net)*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Break]*)
BREAK:
(*Do nothing*)
STATE := NOSESSION;
(*Otherwise*)
(*Such as a request when not listening,*)
(*for data transfer when not in session*)
(*Should be prevented by protocol*)
END;(*CASE*)
(*Endif*)
(*When [State = requesting]*)
REQUESTING:
(*If [Packet is from Application layer] then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
BEGIN
(*Application process should be blocked,*
(*so this can not happen.*)
(*Notify Application layer of error*)
PACKET.DIRECTION := INCOMING;
PACKET.STATUS := PACKET.STATUS
+ [NO_LOCAL_SESSION];
TO_APP.DEPOSIT (PACKET)
END
(*Else (packet from the net]*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Start]*)
START:
BEGIN
(*Go into In_Session state*)
STATE := IN_SESSION;
(*Send up any piggy backed data*)
TO_APP.DEPOSIT (PACKET)
END;
(*When [Cmd = Break]*)
(* A-49 *)
BEGIN
(*Go into No_Session state*)
STATE := NO_SESSION;
PACKET.STATUS := PACKET.STATUS + [SESSION_ENDING];
(*Send up any piggy backed data*)
TO_APP.DEPOSIT (PACKET)
END;
(*Otherwise*)
(*Should be prevented by protocol*)
END (*Case*);
(*Endif*)
(*When [State = Listening]*)
LISTENING:
(*If [packet from application layer] then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
BEGIN
(*Application layer should be blocked so*)
(*this can not happen*)
(*Notify application layer of error*)
PACKET.STATUS := PACKET.STATUS +
[NO_LOCAL_SESSION];
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END
(*Else (packet from network]*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Request]*)
REQUEST:
BEGIN
(*If [it has the right password] then*)
IF (PACKET.TEXT = PASS_WORD) THEN
BEGIN
(*Set up packet for a favorable reply*)
PACKET.SESSION_CMD := START;
(*Go to In_Session state*)
STATE := IN_SESSION
TO_APP.DEPOSIT (PACKET)
END
(*Else (bad password]*)
ELSE
BEGIN
PACKET.SESSION_CMD := BREAK;
(*Set up packet for a refusal*)
PACKET.STATUS := PACKET.STATUS + [BAD_PASSWORD]
END;
(*ENDIF*)
(*Send a return packet*)
PACKET.DIRECTION := OUTGOING;
(* A-50 *)
(**$** SESSION **$**)
(**$** Process to simulate Session layer **$**)

TO_NET.DEPOSIT (PACKET)
END;

(*Otherwise*)
ELSE:
BEGIN
(*Should be prevented by protocol*)
(*Notify source of error*)
PACKET.STATUS := PACKET.STATUS + [NO_REMOTE_SESSION];
PACKET.SESSION_CMD := BREAK;
PACKET.DIRECTION := OUTGOING;
TO_NET.DEPOSIT (PACKET)
END
END;(*CASE*)

(*Endif*)

(*When [State = In_session]*)
IN_SESSION:
(*If [packet is from application layer] then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
CASE PACKET.SESSION_CMD OF
(*When [Cmd is a data transfer type]*)
CHAIN, END_CHAIN, ABORT_CHAIN, IMMEDIATE:
BEGIN
(*Send data*)
TO_NET.DEPOSIT (PACKET)
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END;

(*When [Cmd = Break]*)
BREAK:
BEGIN
(*Go to No_Session state*)
STATE := NO_SESSION;
(*Send a Break to partner*)
TO_NET.DEPOSIT (PACKET);
PACKET.STATUS := PACKET.STATUS + [SESSIONENDING];
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END;

(*Otherwise*)
ELSE:
BEGIN
(*Notify Application layer it has made a*)
(*mistake*)
PACKET.STATUS := PACKET.STATUS + [LOCAL_IN_SESSION];
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END

(* A-51 *)
(*$SESSION$*)
(*$Process to simulate Session layer$*)

END(*CASE*)
(*Else (packet from the network)*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Break]*)
BREAK:
BEGIN
(*Go to No_Session state*)
STATE := NO_SESSION;
(*Send up any messages left in the chain*)
(*buffer*)
FOR CHAIN_INDEX := 1 TO CHAIN_PTR DO
   TO_APP.DEPOSIT
      (CHAIN_BUFF [CHAIN_INDEX]);
{ENDIFOR}
CHAIN_PTR := 0;
PACKET.STATUS := PACKET.STATUS + [SESSION_ENDING];
(*Notify Application of session ending*)
TO_APP.DEPOSIT (PACKET)
END;
(*When [Cmd = Request]*)
REQUEST:
BEGIN
(*Send back a busy signal*)
PACKET.STATUS := PACKET.STATUS + [BUSY];
PACKET.SESSION_CMD := BREAK;
PACKET.DIRECTION := OUTGOING;
TO_NET.DEPOSIT (PACKET)
END;
(*When [Cmd = Immediate]*)
IMMEDIATE:
(*Immediately pass it to Application*)
TO_APP.DEPOSIT (PACKET);
(*When [Cmd = Chain]*)
CHAIN:
BEGIN
(*Store the message in the Chain buffer*)
CHAIN_PTR := CHAIN_PTR + 1;
(*NOTE-possible Chain_Index range error*)
CHAIN_BUFF [CHAIN_PTR] := PACKET
END;
(*When [Cmd = End_Chain]*)
END_CHAIN:
BEGIN
(*Pass up all messages stored in the*)
(*Chain buffer*)
FOR CHAIN_INDEX := 1 TO CHAIN_PTR DO
   TO_APP.DEPOSIT
      (CHAIN_BUFF [CHAIN_INDEX]);
{ENDIFOR}

(* A-52 *)
(**$$ SESSION $$**)
(**$$ Process to simulate Session layer $$**)

CHAIN_PTR := 0;
TO_APP.DEPOSIT (PACKET)
END;
(*When [Cmd = Abort_Chain]*)
ABORT_CHAIN:
(*Empty the Chain buffer*)
CHAIN_PTR := 0;
(*Otherwise*)
ELSE:
(*Should be prevented by protocol*)
BEGIN
(*Disconnect everyone and start over*)
PACKET.STATUS := PACKET.STATUS + [REMOTE_IN_SESSION];
PACKET.SESSION_CMD := BREAK;
PACKET.DIRECTION := OUTGOING;
TO_NET.DEPOSIT (PACKET)
END
END(*CASE*);
(*End*)
END(*CASE*)
(*End cycle*)
END
(*End Session_process*)
END;

(* A-53 *)
TYPE WORKER_PROCESS = PROCESS(CONSOLE: RESOURCE_MONITOR;
FROM_NET: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR);

THE WORKER_PROCESS is an application layer process that
transfers remote files to the operator console.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

CONST

ERROR_MSG = Structured constant containing text
explanations of possible packet errors.

TYPE

MESSAGE_IO_CLASS = A class that uses supervisory
calls to handle fixed record I/O to specified
logical units.

PACKET_TYPE = Record structure of the network
packets.

MESSAGE_TYPE = Array of characters.

ERROR_SVC1_TYPE = Record structure of the status
bytes from the supervisory call.

MAIL_BOX_MONITOR = A monitor used for passing
packets between processes.

RESOURCE_MONITOR = Allows only one process to
access a resource at a time.

INTERNAL

VAR

STATUS_INDEX: Used to report packet errors.
OP: Used to write lines of the transferred file
to the operator.

PACKET: A network packet this process uses to
communicate with the network.

TEXT: Array of characters used to communicate
the operator.

OP_STATUS: Receives the status bytes from the
MESSAGE_IO_CLASS. Not used here, but necessary
for the calls to OP.

PARAMETERS

CONSOLE: The RESOURCE_MONITOR used to reserve the
console for exclusive I/O.

FROM_NET: The monitor used to recieve packets from
the network.

TO_NET: The monitor used to send packets to the net

VAR

OP: MESSAGE_IO_CLASS;

(* A-54 *)
ERROR: ERROR_CLASS;
OS: NET_IO_CLASS;
NET_STATUS: PKT_STATUS_TYPE;
TEXT: MESSAGE_TYPE;
OP_STATUS: ERROR_SVC1_TYPE; (****)

(*Begin Worker process*)
BEGIN
(*Initialize the interface to the operator*)
  INIT OP;
  INIT ERROR (CONSOLE);
  INIT OS (FROM_NET, TO_NET);
(*Cycle forever*)
CYCLE
  CONSOLE.REQUEST;
  OP.WRITE (TERMINAL, 'ENTER SERVICE PASSWORD', OP_STATUS);
  OP.READ (TERMINAL, TEXT, OP_STATUS);
  CONSOLE.RELEASE;
(**Request a session**) (****)
  OS.MAKE_SESSION (TEXT, NET_STATUS);
  ERROR.REPORT ('WORKER 10', NET_STATUS);
(*If [we connect] then*)
  IF (NET_STATUS = [1]) THEN (****)
  BEGIN
  REPEAT
    TEXT := 'ENTER FILE ID. = ';
    TEXT [18] := FIRST_FILE_ID;
    CONSOLE.REQUEST;
    OP.WRITE (TERMINAL, TEXT, OP_STATUS);
    OP.READ (TERMINAL, TEXT, OP_STATUS);
    CONSOLE.RELEASE;
(*Send the request to the server*)
    OS.NET_WRITE (TEXT, IMMEDIATE, PUBLIC, NET_STATUS);
    ERROR.REPORT ('WORKER 20', NET_STATUS);
(*Transfer the file to the console*)
    CONSOLE.REQUEST;
(*Repeat until end of file*)
  REPEAT
    (*Get a line from the network*)
    OS.NET_READ (TEXT, NET_STATUS);
    (*Output it to the console*)
    OP.WRITE (TERMINAL, TEXT, OP_STATUS);
(*End repeat*)
  OR (SESSION_ENDING IN NET_STATUS);
  CONSOLE.RELEASE;
  ERROR.REPORT ('WORKER 30', NET_STATUS);
  CONSOLE.REQUEST;
    OP.WRITE (TERMINAL, 'MORE FILES Y/N ', OP_STATUS);
    OP.READ (TERMINAL, TEXT, OP_STATUS);

(* A-55 *)
(### Worker3 ###)
(### Net3 Worker application process ###)

CONSOLE.RELEASE
UNTIL (TEXT [1] = 'N');
OS.CLEAR_SESSION (NET_STATUS);
ERROR.REPORT ('Worker 40 ', NET_STATUS)
END

(Endif*)
(End cycle*)
END

(End Worker process*)
END;
TYPE SERVER_PROCESS = PROCESS (CONSOLE: RESOURCE_MONITOR;
                   FROM_NET: MAIL_BOX_MONITOR;
                   TO_NET: MAIL_BOX_MONITOR);

The server process is an application layer process that
does the disk I/O for a remote worker process.

PROGRAMMER: ROM ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL

TYPE
MESSAGE_IO_CLASS = A class that uses supervisory
calls to handle fixed record I/O to specified
logical units.
PACKET_TYPE = Record structure of the network
packets.
MESSAGE_TYPE = Array of characters.
ERROR_SVC1_TYPE = Record structure of the status
bytes from the supervisory call.
MAIL_BOX_MONITOR = A monitor used for passing
packets between processes.

INTERNAL

VAR

DISK: Used to input lines of a disk file.
 PACKET: A network packet used to communicate with
         the network.
 TEXT: Array of characters used for the disk I/O.
 FILE_ID: The id of the file the worker process is
         requesting.
 VALID_FILE_IDS: A set of the valid id's this
                    process can access.
 FILE_LU: An array of logical units that are
          subscripted with file id's. Used to look up the
          logical unit of a file.
 NEXT_LU: Used in initializing FILE_LU.
 INDEX: Used in initializing FILE_LU.
 OP_STATUS: Receives the status bytes form the
            MESSAGE_IO_CLASS.

PARAMETERS
 FROM_NET: The monitor used to receive packets from
 the network.
 TO_NET: The monitor used to send packets to the net.

VAR
 DISK: MESSAGE_IO_CLASS;
 OS: NET_IO_CLASS;
 NET_STATUS: PKT_STATUS_TYPE;
ERROR: ERROR_CLASS;
TEXT: MESSAGE_TYPE;
FILE_ID: CHAR;
VALID_FILE_IDS: SET OF CHAR;
FILE_LU: ARRAY [FILE_RANGE] OF BYTE;
NEXT_LU: BYTE;
INDEX: FILE_RANGE;
OP_STATUS: ERROR_SVC1_TYPE;

(*Begin Server process*)
BEGIN
(*Initialize the interface to the disk files*)
INIT DISK;
INIT OS (FROM_NET, TO_NET);
INIT ERROR (CONSOLE);
(*Set up an array to reference logical unit numbers*)
(*by the character id's of the files*)
NEXT_LU := TERMINAL + 1;
(*For all file id's do*)
FOR INDEX := FIRST_FILE_ID TO LAST_FILE_ID DO
BEGIN
(*Remember that it is a valid id*)
VALID_FILE_IDS := VALID_FILE_IDS + [INDEX];
(*Set it's logical unit number*)
FILE_LU [INDEX] := NEXT_LU;
NEXT_LU := NEXT_LU + 1
END;
(*Endfor*)
(*Cycle forever*)
CYCLE
(*Put your ears up*)
OS_NET_LISTEN (NET_STATUS);
ERROR_REPORT ('SERVER 10 ', NET_STATUS);
(*Get the request from the net*)
OS_NET_READ (TEXT, NET_STATUS);
ERROR_REPORT ('SERVER 15 ', NET_STATUS);
FILE_ID := TEXT [1];
WHILE NOT (SESSION_ENDING IN NET_STATUS) DO
BEGIN
(*If [it is a valid file id] then*)
IF (FILE_ID IN VALID_FILE_IDS) THEN
(*Transfer the file*)
BEGIN
(*Read in a line from the disk*)
DISK_READ (FILE_LU [FILE_ID], TEXT, OP_STATUS);
(*While not [end of file] do*)
WHILE (OP_STATUS.DI = 0) AND (OP_STATUS.DD = 0) DO
BEGIN
(*Send it out on the network*)
OS_NET_WRITE (TEXT, IMMEDIATE, SECRET,
NET_STATUS);
ERROR_REPORT ('SERVER 20 ',

(* A-58 *)
NET_STATUS);
(*Read in a new line from the disk file*)
   DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS)
   END;
(*Endwhile*)
(*Rewind the disk file*)
   DISK.REWIND (FILE_LU [FILE_ID], OP_STATUS);
(*Send an EOF packet out on the network*)
   OS.NET_WRITE ('/
   IMMEDIATE, PUBLIC, NET_STATUS);
   ERROR.REPORT ('SERVER 30
   ',NET_STATUS);
   END
(*Else (an invalid file id)*)
ELSE
(*Send an error message*)
   BEGIN
   TEXT := '/
   BAD FILE ID - '
   TEXT [19] := FILE_ID;
   OS.NET_WRITE (TEXT,IMMEDIATE,PUBLIC,NET_STATUS);
   ERROR.REPORT ('SERVER 40
   ',NET_STATUS);
   END;
(*Endif*)
OS.NET_READ (TEXT, NET_STATUS);
FILE_ID := TEXT [1];
ERROR.REPORT ('SERVER 50
   ', NET_STATUS)
END
(*Endwhile*)
(*End cycle*)
END
(*End Server*)
END;
(*$LOCNET$*)

(*$Process to simulate a Local Area Network$*)

TYPE NETWORK_PROCESS = PROCESS ( TO_APP: MAIL_BOX_MONITOR;
EVENT: MAIL_BOX_MONITOR;
NEXT_NODE: MAIL_BOX_MONITOR;
NODE_ID: HOST_ID_TYPE);

(* The network layer represents the hardware and software
necessary to transmit packets across a local area network. *)

PROGRAMMER: RON ALBURY

DATE WRITTEN: 6/28/82

COPYRIGHT 1982 by Ronald C. Albury

EXTERNAL

TYPE

PACKET_TYPE = Record structure of the network packets.
MAIL_BOX_MONITOR = A monitor used for passing packets between processes.*

INTERNAL

VAR

PACKET: A network packet that the layer processes

PARAMETERS

TO_APP: The monitor used to send packets up toward the application layer.
EVENT: The monitor this layer uses to recieve packets.
NEXT_NODE: The monitor used to send packets to the next node in the network.
NODE_ID: Tells the process which node it is serving.*

VAR

PACKET: PACKET_TYPE;

BEGIN

(*Cycle forever*)

CYCLE

(*Wait for a packet*)

EVENT.GET (PACKET);

(*If [an outgoing packet] then*)

IF ( PACKET.DIRECTION = OUTGOING ) THEN

BEGIN

(*Set it as an incoming packet*)

PACKET.DIRECTION := INCOMING;

(*Pass it on to the next node*)

NEXT_NODE.DEPOSIT (PACKET)

END

(*Else (an incoming packet]*)

ELSE

(* A-60 *)
(*$$$ LOCNET $$*)
(*$$$ Process to simulate a Local Area Network $$*)

(*If [this is the packets destination] then*)
IF (PACKET.DESTINATION = NODE_ID) THEN
(*Pass it up to the application layer*)
TO_APP.DEPOSIT (PACKET)
(*Else it is bound for another node*)
ELSE
(*If [it hasn't made it around the ring] then*)
IF (PACKET.SOURCE <> NODE_ID) THEN
(*Send it on*)
NEXT_NODE.DEPOSIT (PACKET)
(*Else destination node must be down*)
ELSE
BEGIN
(*Notify host*)
PACKET.DIRECTION := INCOMING;
PACKET.STATUS := PACKET.STATUS + [DESTINATION_NODE_DOWN];
PACKET.TRANS_CMD := DISCONNECT;
TO_APP.DEPOSIT (PACKET)
END
(*Endif made it around ring*)
(*Endif belongs at this node*)
(*Endif packet direction*)
(*End cycle*)
END
(*End Network process*)
END;

(* A-61 *)
TYPE NET_IO_CLASS = CLASS (FROM_NET: MAIL_BOX_MONITOR;
  TO_NET: MAIL_BOX_MONITOR);

**-----------------------------------**
** NET_IO_CLASS provides standard entry points for **
** interfacing with the network layers. Allows creation **
** and destruction of sessions, and data transferal. **
**-----------------------------------**

Programmer: Ronald C. Albury
Date Written: 10/08/82
Computer: Interdata 8/32
Copyright 1982 by Ronald C. Albury

**-----------------------------------**

EXTERNAL

**-----------------------------------**

TYPE

MAIL_BOX_MONITOR: Interprocess communication
mailbox.

PACKET_TYPE: Record structure of the network
packets.

MESSAGE_TYPE: Array of characters.

PKT_STATUS_TYPE: A field of the Packet_type
record, for error flags.

CHAIN_TYPE: The subrange of session commands
dealing with chaining messages.

**-----------------------------------**

INTERNAL

VAR

PACKET: The network packet that is assembled
and passed to the lower layers.

**-----------------------------------**

VAR

PACKET: PACKET_TYPE;

**-----------------------------------**

PROCEDURE ENTRY NET_LISTEN

PARAMETERS

OUT

STATUS: Error flags indicating the status of the
listen.

**-----------------------------------**

PROCEDURE ENTRY NET_LISTEN (VAR STATUS: PKT_STATUS_TYPE);

(*Begin entry Net_listen*)
BEGIN

(*Set the network parameters for a listen*)

PACKET.TEXT := ';
PACKET.SECURITY := PUBLIC;
PACKET.DIRECTION := OUTGOING;
PACKET.PRIORITY := MED_PRI;
PACKET.SESSION_CMD := LISTEN;
PACKET.NAME := 'APPLI CMD LISTEN ';

(*Issue the listen to the network and wait for*)
(*a response*)

TO_NET.DEPOSIT (PACKET);
FROM_NET.GET (PACKET);

(* A-62 *)
(* Set the status flags *)
STATUS := PACKET.STATUS
(* End entry Net_listen *)
END;

(*PROCEDURE ENTRY MAKE_SESSION******************************************
* PARAMETERS
* IN
* PASSWORD: Array of characters containing the
* the password to be sent to the destination
* process when the session request is issued.
* NAME: Array of characters containing the common
* name of the destination process.
* OUT
* STATUS: Error flags indicating the status of the
* session request.
******************************************)
PROCEDURE ENTRY MAKE_SESSION (PASSWORD: MESSAGE_TYPE;
NAME: MESSAGE_TYPE;
VAR STATUS: PKT_STATUS_TYPE);

(*Begin entry Make_session*)
BEGIN
(*Set the network parameters for a session request*)
PACKET.NAME := NAME; (***)
PACKET.TEXT := PASSWORD;
PACKETSECURITY := PUBLIC;
PACKET.DIRECTION := OUTGOING;
PACKET.STATUS := [];
PACKET.SESSION_CMD := ESTABLISH;
PACKET.PRIORITY := LOW_PRI;
(*Issue the session request to the network and wait*)
(*for a response*)
TO_NET.DEPOSIT (PACKET);
FROM_NET.GET (PACKET);
(*Set the status flags*)
STATUS := PACKET.STATUS
(*End entry Make_session*)
END;

(*PROCEDURE ENTRY CLEAR_SESSION****************************************
* PARAMETERS
* OUT
* STATUS: Error flags indicating the status of the clear.
****************************************
PROCEDURE ENTRY CLEAR_SESSION (VAR STATUS: PKT_STATUS_TYPE);
(*Begin entry Clear_session*)
BEGIN
(*Set the network parameters for a clear*)
PACKET.TEXT := '\0';
PACKETSECURITY := PUBLIC;
PACKET.DIRECTION := OUTGOING;
(* A-63 *)
PACKET.STATUS := []; 
PACKET.SESSION_CMD := BREAK; 
PACKET.PRIORITY := MED_PRI; 
PACKET.NAME := 'APPLI CMD CLEAR SESS'; (****) 
(#Issue the clear to the network and wait for a#) 
(#response#) 
TO_NET.DEPOSIT (PACKET); 
FROM_NET.GET (PACKET); 
(#Set the status flags#) 
STATUS := PACKET.STATUS 
(#End entry Clear_session#) 
END;

(***PROCEDURE ENTRY NET_READ*******************************)

PARAMETERS

OUT

TEXT: Array of characters to receive the message 
    from the network.
STATUS: Error flags indicating the status of the 
    read.

(*******************************************************************************)

PROCEDURE ENTRY NET_READ (VAR TEXT: MESSAGE_TYPE;
VAR STATUS: PKT_STATUS_TYPE);
(#Begin entry Net_read#) 
BEGIN 
(#Get the packet from the network#) 
FROM_NET.GET (PACKET); 
(#Extract the Text and Status flags from the packet#) 
TEXT := PACKET.TEXT; 
STATUS := PACKET.STATUS 
(#End entry Net_read#) 
END;

(***PROCEDURE ENTRY NET_WRITE*******************************)

PARAMETERS

IN

TEXT: Array of characters to send on the network. 
XFER: The session layer chain command for this 
data transfer. 
SECURITY: The security level desired for this 
data item. 

OUT

STATUS: Error flags indicating the status of the 
write.

(*******************************************************************************)

PROCEDURE ENTRY NET_WRITE (TEXT: MESSAGE_TYPE;
XFER: CHAIN_TYPE;
SECURITY: SECURITY_TYPE;
VAR STATUS: PKT_STATUS_TYPE); 
(#Begin entry Net_write#) 
BEGIN 
(#Set the network parameters for a write.#) 

(# A-64 #)
(*$NETIO4 $*$*)
(*$Revised Network Entries $*$*)

```plaintext
PACKET.TEXT := TEXT;
PACKET.SECURITY := SECURITY;
PACKET.DIRECTION := OUTGOING;
PACKET.STATUS := [];
PACKET.SESSION_CMD := XFER;
PACKET.PRIORITY := MED_PRI;
PACKET.NAME := 'APPL CMD WRITE';  (*$$*)
(*Issue the write to the net and wait for a response*)
TO_NET.DEPOSIT (PACKET);
FROM_NET.GET (PACKET);
(*Set the status flags*)
STATUS := PACKET.STATUS
(*End entry Net_write*)
END;

BEGIN
END;
```

(* A-65 *)
TYPE SESSION_PROCESS = PROCESS (TO_APP: MAIL_BOX_MONITOR;
EVENT: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR;
HOST_ID: HOST_ID_TYPE);

The session layer handles the initial set up of a
session between two hosts. It also is capable of
chaining a group of related messages together to
make sure that, in the event of network failure,
the receiver is not in the middle of a transmission.

- PROGRAMMER: RON ALBURY
- DATE WRITTEN: 4/2/82
- COMPUTER: INTERDATA 8/32
- Copyright 1982 by Ronald C. Albury

EXTERNAL
CONST
  PASSWORD: A structured constant containing the
    passwords of the hosts on the network.
  MAX_SESSION_WAIT: The maximum number of pending
    requests the host is allowed to queue up.
  MAX_CHAIN: The maximum number of packets the
    layer can chain before delivering them to the
    application layer.

TYPE
  PACKET_TYPE: Record structure of the network
    packet.
  MAIL_BOX_MONITOR: A monitor used for passing
    packets between processes.
  HOST_ID_TYPE: Enumeration of hosts in network.
  MESSAGE_TYPE: Array of characters.
  FIFO: Modified Brinch Hansen FIFO class to
    handle a fifo buffer.

INTERNAL

TYPE
  SESSION_STATE_TYPE: Enumeration of the states
    a session layer can be in.

VAR
  PASS_WORD: Contains this host's password. Used
    to check if incoming requests have access
    rights to this host.
  PACKET: The network packet this process uses to
    communicate with.
  WAIT_BUFF: A fifo controlled buffer used to
    store waiting request packets.
  WAITING_REQ: A FIFO class to control WAIT_BUFF.
  CHAIN_BUFF: An array of packets used to store
    chained packets so they can be delivered to
    application layer as a group.
  CHAIN_PTR: Integer used to control CHAIN_BUFF.
CHAIN_INDEX: Integer used to increment through the CHAIN_BUFF when delivering the packets.
STATE: The current state of the session layer.

PARAMETERS
* TO_APP: The monitor used to send packets to the application layer.
* EVENT: The monitor this layer uses to receive packets.
* TO_NET: The monitor used to send packets to the network.
* HOST_ID: The id given to this host when the network is brought up.

TYPE
SESSION_STATE_TYPE = (NO_SESSION, LISTENING, REQUESTING, IN_SESSION);

VAR
PASS_WORD: MESSAGE_TYPE;
PACKAGE: PACKET_TYPE;
CHAIN_INDEX: INTEGER;
CHAIN_PTR: INTEGER;
CHAIN_BUFF: ARRAY [1..MAX_CHAIN] OF PACKET_TYPE;
STATE: SESSION_STATE_TYPE;

(*Begin Session_process*)
BEGIN
(*Initialize the host's password*)
PASS_WORD := PASSWORD [HOST_ID];
(*Initialize the state to No_session*)
STATE := NO_SESSION;
(*Cycle forever*)
CYCLE
EVENT.GET (PACKAGE);
(*Process the packet based on current state*)
CASE STATE OF
(*When [State = No_session]*)
NO_SESSION:
(*If [packet is from application layer] then*)
IF (PACKAGE.DIRECTION = OUTGOING) THEN
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Listen]*)
LISTEN:
(*Go into Listening state*)
STATE := LISTENING;
(*When [Cmd = Establish]*)
ESTABLISH:
BEGIN
(*Go into Requesting state*)
STATE := REQUESTING;
(*Piggy back a session request on a*)

(* A-67 *)
(*$*$ $SESSION4 $*$*)
(*$*$ Revised Session layer $*$*)

(*Transport Connect command*)
PACKET.TRANS_CMD := CONNECT;  (*#*)
PACKET.SESSION_CMD := REQUEST;
TO_NET.DEPOSIT (PACKET)
END;
(*Otherwise*)
ELSE:
BEGIN
(*Notify application layer of illegal*)
(*command*)
PACKET.STATUS := PACKET.STATUS
+ [NO_LOCAL_SESSION];
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END
END(*CASE*)
(*Else (packet from the net]*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Break]*)
BREAK:
(*Do nothing*)
STATE := NO_SESSION;
(*Otherwise*)
(*Such as a request when not listening,*)
(*for data transfer when not in session*)
(*Should be prevented by protocol*)
END(*CASE*)
(*Endif*)
(*When [State = requesting]*)
REQUESTING:
(*If [Packet is from Application layer] Then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
BEGIN
(*Application process should be blocked,*)
(*so this can not happen.*)
(*Notify Application layer of error*)
PACKET.DIRECTION := INCOMING;
PACKET.STATUS := PACKET.STATUS
+ [NO_LOCAL_SESSION];
TO_APP.DEPOSIT (PACKET)
END
(*Else (packet from the net]*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Start]*)
START:
BEGIN
(*Assume it came from Transport layer*)
(*On a Data_xfer.*)
(*Go into In_Session state*)
STATE := IN_SESSION;

(* A-68 *)
(*Sending Session data*)
TO_APP.DEPOSIT (PACKET)
END;
(*When [Cmd = Break]*)
BREAK:
BEGIN
(*Assume it came from Transport layer on*)
(*Disconnect*)
(*Go into No_Session state*)
STATE := NO_SESSION;
PACKET.STATUS := PACKET.STATUS + [SESSION_ENDING];
(*Send up any piggy backed data*)
TO_APP.DEPOSIT (PACKET)
END;
(*Otherwise*)
(*Should be prevented by protocol*)
END(Case);
(*Endif*)
(*When [State = Listening]*)
LISTENING:
(*If [packet from application layer] then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
BEGIN
(*Application layer should be blocked so*)
(*this can not happen*)
(*Notify application layer of error*)
PACKET.STATUS := PACKET.STATUS + [NO_LOCAL_SESSION];
PACKET.DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END
(*Else (packet from network]*)
ELSE
CASE PACKET.SESSION_CMD OF
(*When [Cmd = Request]*)
REQUEST:
BEGIN
(*If [it has the right password] then*)
IF (PACKET.TEXT = PASS_WORD) THEN
BEGIN
(*Set up packet for a favorable reply*)
PACKET.SESSION_CMD := START;
PACKET.TRAN_CMD := DATA_XFER; (***)
PACKET.NAME := 'SESSION STARTS';
(*Go to In_Session state*)
STATE := IN_SESSION
(*Do not unblock Application process*)
(*Let the Transport Ack do that*)
END
(*Else (bad password]*)
ELSE
(*)
BEGIN
    PACKET.SESSION_CMD := BREAK;
    (* Set up packet for a refusal *)
    PACKET.TRANSACTION_CMD := DISCONNECT; (* *)
    PACKET.NAME := 'BAD PASSWORD';
    PACKET.STATUS := PACKET.STATUS
        + [BAD_PASSWORD]
END;
(*ENDIF*)
(*Send a return packet*)
    PACKET.DIRECTION := OUTGOING;
    TO_NET.DEPOSIT (PACKET)
END;
(*Otherwise*)
ELSE:
BEGIN
(* Should be prevented by protocol *)
(* Notify source of error *)
    PACKET.STATUS := PACKET.STATUS
        + [NO_REMOTE_SESSION];
    PACKET.SESSION_CMD := BREAK;
    PACKET.TRANSACTION_CMD := DISCONNECT; (* *)
    PACKET.DIRECTION := OUTGOING;
    TO_NET.DEPOSIT (PACKET)
END
(*End;*)
(*End if*)
(*When [State = In_session]*)
IN_SESSION:
(*If [packet is from application layer] then*)
    IF (PACKET.DIRECTION = OUTGOING) THEN
    CASE PACKET.SESSION_CMD OF
(*When [Cmd is a data transfer type]*)
        CHAIN, END_CHAIN, ABORT_CHAIN, IMMEDIATE:
        BEGIN
            (* Set packet as a data transfer *)
            PACKET.TRANSACTION_CMD := DATA_XFER; (* *)
            (* Send data *)
            TO_NET.DEPOSIT (PACKET)
(*This is taken over *)
            PACKET.DIRECTION := INCOMING;(* *)
(*by transport layer *)
            TO_APP.DEPOSIT (PACKET) (* *)
        END;
(*When [Cmd = Break]*)
BREAK:
    BEGIN
(* Go to No_Session state *)
        STATE := NO_SESSION;
(* Piggy back break on disconnect*)
        PACKET.TRANSACTION_CMD := DISCONNECT; (* *)
        PACKET.NAME := 'SESSION ENDING';
(* Send a Break to partner *)
        TO_NET.DEPOSIT (PACKET);
(* A-70 *)
PACKET_STATUS := PACKET_STATUS
("Notify Application layer of status")
("of Break")
+ [SESSION_ENDING];
PACKET_DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END;
("Otherwise")
ELSE:
BEGIN
("Notify Application layer it has made a")
("mistake")
PACKET_STATUS := PACKET_STATUS
+ [LOCAL_IN_SESSION];
PACKET_DIRECTION := INCOMING;
TO_APP.DEPOSIT (PACKET)
END
END("CASE")
("Else (packet from the network")
ELSE
CASE PACKET_SESSION_CMD OF
("When [Cmd = Break"]
BREAK:
("Assume transport connection broken")
BEGIN
("Go to No_Session state")
STATE := NO_SESSION;
("Send up any messages left in the chain")
("buffer")
FOR CHAIN_INDEX := 1 TO CHAIN_PTR DO
   TO_APP.DEPOSIT
      (CHAIN_BUFF [CHAIN_INDEX]);
ENDFOR
CHAIN_PTR := 0;
PACKET_STATUS := PACKET_STATUS
+ [SESSION_ENDING];
("Notify Application of session ending")
TO_APP.DEPOSIT (PACKET)
END;
("When [Cmd = Request"]
REQUEST:
BEGIN
("Should be prevented by Transport layer")
("Send back a busy signal")
PACKET_STATUS := PACKET_STATUS
+ [BUSY];
PACKET_SESSION_CMD := BREAK;
PACKET_DIRECTION := OUTGOING;
TO_NET.DEPOSIT (PACKET)
END;
("When [Cmd = Immediate"]
IMMEDIATE:

(* A-71 *)
(*$$ SESSION4 $$*)
(*$$ Revised Session layer $$*)

(*Immediately pass it to Application*)
TO_APP.DEPPOSIT (PACKET);
(*When [Cmd = Chain]*)
CHAIN:
BEGIN
(*Store the message in the Chain buffer*)
CHAIN_PTR := CHAIN_PTR + 1;
(*Note possible Chain_Index range error*)
CHAIN_BUFF [CHAIN_PTR] := PACKET
END;
(*When [Cmd = End_Chain]*)
END_CHAIN:
BEGIN
(*Pass up all messages stored in the*)
(*Chain buffer*)
FOR CHAIN_INDEX := 1 TO CHAIN_PTR DO
   TO_APP.DEPPOSIT
   (CHAIN_BUFF [CHAIN_INDEX]);
ENDFOR
CHAIN_PTR := 0;
TO_APP.DEPPOSIT (PACKET)
END;
(*When [Cmd = Abort_Chain]*)
ABORT_CHAIN:
(*Empty the Chain buffer*)
CHAIN_PTR := 0;
(*Otherwise*)
ELSE:
(*Should be prevented by Transport layer*)
BEGIN
(*Disconnect everyone and start over*)
PACKET.STATUS := PACKET.STATUS + [REMOTE_IN_SESSION];
PACKET.SESSION_CMD := BREAK;
PACKET.DIRECTION := OUTGOING;
TO_NET.DEPPOSIT (PACKET)
END

END(*CASE*);
(*endif*)
END(*CASE*)
(*End cycle*)
END
(*End Session_process*)
END;

(* A-72 *)
TYPE CLOCK_MONITOR = MONITOR;
(***************************************************************************
 * The CLOCK_MONITOR is the interface between the Transport layer and the Timeout_Process. *
 * ***************************************************************************)
 *
 * Programmer: Ronald C. Albury
 * Date Written: 10/10/82
 * Copyright 1982 by Ronald C. Albury
 *
 * EXTERNAL
 * TYPE
 * TRANSPORT_COMMANDS: Enumerations of all possible commands the Transport layer will respond to.
 *
 * INTERNAL
 *
 * VAR
 *   LIMIT: Stores the number of seconds the Timeout process should wait before sending a packet to the Transport process.
 *   ELAPSED: The number of seconds that have elapsed since the Transport process has initiated the timer.
 *   TIMER: Queue variable to delay the Timeout process when it is not being used.
 *   STOP_TIMER: A boolean flag indicating if the Transport process wants the Timeout process turned off.
 *   TRANS_EVT: Stores the transport event to be delivered in the event of a time out.
 *
***************************************************************************)

VAR
   LIMIT, ELAPSED: INTEGER;
   TIMER: QUEUE;
   STOP_TIMER: BOOLEAN;
   TRANS_EVENT: TRANSPORT_COMMANDS;

(**Procedure Entry Start***************************************************************************
 * PARAMETERS
 * IN
 *   MAX_TICK: The number of seconds before a time out is delivered.
 *   T_EVT: The transport event to be delivered if there is a time out.
 *
***************************************************************************)

PROCEDURE ENTRY START (MAX_TICK: INTEGER;
                        T_EVT: TRANSPORT_COMMANDS);

(*Begin entry Start*)
BEGIN
  (*Reset the clock to 0*)
  ELAPSED := 0;
  (*Remember the time limit and transport event*)
  TRANS_EVENT := T_EVT;

(*)
/**
  ** C L O C K  
  ** Monitor for controlling time-outs **/
/**

LIMIT := MAX_TICK;
(*Turn the timer on*)
STOP_TIMER := FALSE;
CONTINUE (TIMER)
(*End entry Start*)
END;

(*--------------------------*)
PROCEDURE ENTRY STOP;
(*Begin entry Stop*)
BEGIN
(*Set the STOP_TIMER flag*)
STOP_TIMER := TRUE
(*End entry Stop*)
END;

(*--------------------------*)
PROCEDURE Entry Tick

* PARAMETERS
* OUT
* TICK_NUMBER: Receives the current number of ticks
* since the time-out timer was started.
* MAX_TICK: Receives the current time limit for the
* time-out.
* T_EVT: Receives the transport event to be sent
* if the time-out occurs.

(*--------------------------*)
PROCEDURE ENTRY TICK (VAR TICK_NUMBER: INTEGER;
VAR MAX_TICK: INTEGER;
VAR T_EVT: TRANSPORT_COMMANDS);

(*Begin entry Tick*)
BEGIN
(*If [the STOP_TIMER flag is set] then*)
IF (STOP_TIMER) THEN
(*Delay the timer*)
DELAY (TIMER);
(*Endif*)
(*Increment the number of ticks since the time-out started*)
ELAPSED := ELAPSED + 1;
(*Set the output parameters*)
TICK_NUMBER := ELAPSED;
T_EVT := TRANSPORT_EVENT;
MAX_TICK := LIMIT
(*End entry Tick*)
END;

BEGIN
STOP_TIMER := TRUE
END;

(* A-74 *)
TYPE TIMEOUT_PROCESS = PROCESS (CLOCK: CLOCK_MONITOR; 
TRANSPORT: MAIL_BOX_MONITOR); 

The Timeout_Process delivers a high priority network packet to the Transport layer if it is not turned off within a specified time. The Transport layer, through the clock monitor, is able to set both the transport control message to be delivered in the packet, and the length of time the timer will run.

PROGRAMMER: Ronald C. Albury 
DATE WRITTEN: 10/7/82 
Copyright 1982 by Ronald C. Albury

EXTERNAL 

TYPE 

MAIL_BOX_MONITOR: A monitor used for passing network packets between processes. 
PACKET_TYPE: Record structure of the network packets.

INTERNAL 

VAR 

TICK_NUMBER: The number the Clock_Monitor has been entered since the time-out has started. 
MAX_TICK: The number of ticks to wait before sending the time-out packet. 
TIME_PKT: The packet used for delivering the time-out to the transport layer.

PARAMETERS 

CLOCK: The monitor to get the time-out commands from. 
TRANSPORT: The monitor to deliver the time-out packets to.

VAR 

TICK_NUMBER, MAX_TICK: INTEGER; 
TIME_PKT: PACKET_TYPE;

(*Begin Timeout_Process*)

BEGIN 

(*Initialize the Time-out packet*)

TIME_PKT.PRIORITY := HIGH_PRI; 
TIME_PKT.DIRECTION := INCOMING; 
TIME_PKT.NAME := 'THIS IS A TIME OUT'; 
TIME_PKT.STATUS := [];

(*Cycle forever*)

CYCLE 

(*Go into a wait state for 1 second*)

WAIT;

(* A-75 *)
(*Enter the Clock_Monitor for an update from the*)
(*Transport layer.*)
  CLOCK.TICK (TICK_NUMBER, MAX_TICK, TIME_PKT.TRANS_CMD);
(*IF [TIME IS UP EXACTLY] THEN*)
(*@To avoid synchronization problems, allow Tick_number*)
(*@to exceed Max_tick*)
  IF (TICK_NUMBER = MAX_TICK) THEN
    BEGIN
      (*Send the time-out packet*)
      TRANSPORT.DEPosit (TIME_PKT)
    END
  (*Endif*)
END (*Cycle*)
(*End Timeout_Process*)
END;
TYPE TRANSPORT_PROCESS = PROCESS (TO_APP: MAIL_BOX_MONITOR;
EVENT: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR;
TIMER: CLOCK_MONITOR;
THIS_NODE: HOST_ID_TYPE);

******************************************************************************
* The transport layer handles the node's communication with the network. It maps
* hosts requests onto network addresses, and manages all connections. It also
* must assure the delivery of the messages to its host in the order they were sent, despite possible losses in
* the network.
******************************************************************************
* PROGRAMMER: Ronald C. Albury
* DATE WRITTEN: 10/7/82
* COPYRIGHT 1982 by Ronald C. Albury
******************************************************************************
* EXTERNAL
* TYPE
* PACKET_TYPE: Record structure of the network packet.
* MAIL_BOX_MONITOR: A monitor used for passing packets between processes.
* HOST_ID_TYPE: Enumerations of hosts in network.
* CLOCK_MONITOR: Communication interface with the Timeout_Process.
******************************************************************************
* INTERNAL
* TYPE
* TRANS_STATE_TYPE: Enumerations of the states the transport layer can be in.
* IPC_TABLE_TYPE: Record structure used for connection management.
* VAR
* PACKET: The network packet this process uses to communicate with.
* STATE: The current state of the transport layer.
* IPC: Table and buffer used for connection management.
******************************************************************************
TYPE
TRANS_STATE_TYPE = (NOT_CONNECTED, CONNECTING,
IN_CONNECTION);
IPC_TABLE_TYPE = RECORD
  LOCAL_ADDRS: HOST_ID_TYPE;
  REMOTE_ADDRS: HOST_ID_TYPE;
  LOCAL_CONNUM: INTEGER;
  REMOTE_CONNUM: INTEGER;
  DATA_SEQ: DATA_WINDOW_TYPE;
  ACK_SEQ: DATA_WINDOW_TYPE;
  WAITING_FOR_ACK: BOOLEAN;
  BUFFER: PACKET_TYPE

(# A-77 #)
(*$*$ $TRANS$ $*$*$

(*$*$ $Process$ $to$ $simulate$ $Transport$ $layer$ $*$*$

END;

VAR
  PACKET: PACKET_TYPE;
  STATE: TRANS_STATE_TYPE;
  IPC: IPC_TABLE_TYPE;
  WAITING_FOR_ACK: BOOLEAN;

(*Internal Procedure Send_Net*)
PROCEDURE SEND_NET (VAR PACKET: PACKET_TYPE);
(*Begin Send_Net*)
BEGIN
  (*Set the packet direction, connection numbers,*
  (*and addresses*)
  PACKET.DIRECTION := OUTGOING;
  PACKET.S_CONNUM := IPC.LOCAL_CONNUM;
  PACKET.D_CONNUM := IPC.REMOTE_CONNUM;
  PACKET.SOURCE := IPC.LOCAL_ADDR;
  PACKET.DESTINATION := IPC.REMOTE_ADDR;
  TO_NET.DEPOSIT (PACKET)
(*End internal procedure Send_Net*)
END;

(*Begin Transport_Process*)
BEGIN
  (*Initialize the state to NOT_CONNECTED*)
  STATE := NOT_CONNECTED;
  (*Initialize IPC table*)
  IPC.DATA_SEQ := 0;
  IPC.ACK_SEQ := 0;
  IPC.LOCAL_CONNUM := 0;
  IPC.WAITING_FOR_ACK := FALSE;
  (*Cycle forever*)
  CYCLE
    (*Get a packet*)
    EVENT.GET (PACKET);
    (*Process the packet based on current state*)
    CASE STATE OF
      (*When [state = Not Connected]*)
      NOT_CONNECTED:
        (*If [packet is from application layer] then*)
        IF (PACKET.DIRECTION = OUTGOING) THEN
          CASE PACKET.TRANS_CMD OF
            (*When [Cmd = Connect]*)
            CONNECT:
              BEGIN
                (*In an open system, the address of the*)
                (*remote process would have to be looked*)
                (*up in a directory. It would consist*)
                (*of a node id, and a process id.*)
                (*Get address of destination host*)
                IF (PACKET.NAME = DIRECTORY [HOST_S_1])
                  THEN

(* A-78 * )
(*$$ TRAN $$*)

(*$$ Process to simulate Transport layer $$*)

IPC.REMOTE_ADDRS := HOST_S_1
ELSE IF (PACKET.NAME = DIRECTORY
[HOST_S_2]) THEN
  IPC.REMOTE_ADDRS := HOST_S_2
("If [no record of that host] then")
ELSE
BEGIN
("Send an error message to the")
("application program")
PACKET.DIRECTION := INCOMMING;
PACKET.STATUS := PACKET.STATUS
+ [NO_SUCH_HOST];
PACKET.SESSION_CMD := BREAK;
PACKET.NAME := 'ERROR MSG NO HOST';
TO_APP.DEPOSIT (PACKET)
END;
("Endif")
("If [we got the address] then")
IF NOT (NO_SUCH_HOST IN PACKET.STATUS)
THEN
BEGIN
("Go to Connecting state")
STATE := CONNECTING;
("Set up the IPC table")
IPC.BUFFER := PACKET;
IPC.LOCAL_ADDR := THIS_NODE;
IPC.LOCAL_CONNUM :=
  IPC.LOCAL_CONNUM + 1;
IPC.REMOTE_CONNUM := 0;
("Send a connection inquiry to the")
("destination")
PACKET.TRANSACTION := INQUIRE;
PACKET.NAME := 'TRANS INQUIRE';
SEND_NET (PACKET);
("Set the time-out timer in case the")
("inquiry is lost on the net")
TIMER.START (4, T_Q_ACCEPT)
END
("Endif")
END;
("When [Cmd =Disconnect]")
DISCONNECT:
("Disregard it")
("Should be prevented by session layer")
BEGIN
STATE := NOT_CONNECTED
END;
("Otherwise")
ELSE:
("Should be prevented by session")
("Resync on a disconnect")
BEGIN

(* A-79 *)
(*Send an error message to the session*)
(*and application processes*)
  PACKET.DIRECTION := INCOMING;
  PACKET.SESSION_CMD := BREAK;
  PACKET.STATUS := PACKET.STATUS + [INQ_LOCAL_CONNECTION];
  PACKET.NAME := 'DISASTER';
  TO_APP.DEPOSIT (PACKET)
END

CASE PACKET.TRANS_CMD OF
  (*When [Cmd = Inquire]*)
  INQUIRE:
    BEGIN
      (*Go into In_Connection state*)
      STATE := IN_CONNECTION;
      (*Set up the IPC table*)
      IPC.REMOTE_ADDR := PACKET.SOURCE;
      IPC.LOCAL_ADDR := THIS_NODE;
      IPC.LOCAL_CONNNUM := IPC.LOCAL_CONNNUM + 1;
      IPC.REMOTE_CONNNUM := PACKET.S_CONNNUM;
      (*Pass up piggy-backed messages*)
      PACKET.NAME := 'GOT AN INQUIRE';
      TO_APP.DEPOSIT (PACKET);
      (*Send an acceptance to the source of the inquiry*)
      PACKET.TRANS_CMD := ACCEPT;
      PACKET.NAME := 'ACCEPTANCE';
      SEND_NET (PACKET)
    END;
  (*When [Cmd = Disconnect]*)
  DISCONNECT:
    (*Disregard it*)
    (*Should be prevented by session layer*)
    BEGIN
      STATE := NOT_CONNECTED
    END;
  (*When [Cmd = Time_out]*)
  T_Q_ACCEPT, T_Q_DATA_ACK:
    (*Disregard it*)
    (*May be caused by possible delays in*)
    (*Receiving the time-outs*)
    BEGIN
      STATE := NOT_CONNECTED
    END;
  (*Otherwise*)
  ELSE:
    BEGIN
      (*Send an error message and a disconnect*)
      (*to the source of the problem*)
    END

(* A-80 *)
(*** TRANS ***)
(*** Process to simulate Transport layer ***)

(Should be prevented by protocol)
PACKET.DIRECTION := OUTGOING;
PACKET.TRANS_CMD := DISCONNECT;
PACKET.STATUS := PACKET.STATUS
  + [NO_REMOTE_CONNECTION];
PACKET.DESTINATION := PACKET.SOURCE;
PACKET.SOURCE := THIS_NODE;
PACKET.NAME := 'DISASTER';
TQ_NET.DEPOSIT (PACKET)
END

(Endif)

(When [State = Connecting])
CONNECTING:

(If [packet is from the application layer] then)
IF (PACKET.DIRECTION = OUTGOING) THEN
(Should be prevented by session)
(Resync on a disconnect)
BEGIN
PACKET.DIRECTION := INCOMING;
PACKET.SESSION_CMD := BREAK;
PACKET.STATUS := PACKET.STATUS
  + [NO_LOCAL_CONNECTION];
PACKET.NAME := 'DISASTER';
TQ_APP.DEPOSIT (PACKET)
END

(Else from net)
ELSE
CASE PACKET.TRANS_CMD OF
(When [Cmd = Accept])
ACCEPT:
BEGIN
(If [accept matches our request] then)
IF (PACKET.SOURCE = IPC.REMOTE_ADDR)
  & (PACKET.D_CONNUM = IPC.LOCAL_CONNUM)
THEN
BEGIN
(Turn off the time-out)
TIMER.STOP;
(Go to In_Connection state)
STATE := IN_CONNECTION;
(Complete the IPC table)
IPC.REMOTE_CONNUM := PACKET.S_CONNUM
END
(Else a bad accept)
ELSE
(Should be prevented by session)
(unless a node crashes)
(Resync on disconnect)
BEGIN
PACKET.DIRECTION := OUTGOING;
PACKET.SESSION_CMD := BREAK;

(*) A-81 *)
(*$TRAN$*$)

(*$Process to simulate Transport layer*$*)

PACKET.TRANS_CMD := DISCONNECT;
PACKET.STATUS := PACKET.STATUS + [NO_REMOTE_CONNECTION];
PACKET.S_CONN := IPC.LOCAL_CONN;
PACKET.DESTINATION := PACKET.SOURCE;
PACKET.SOURCE := THIS_NODE;
PACKET.NAME := 'DISASTER';
TO_NET.DEPOSIT (PACKET)
END
(*Endif*)
END;
(*When [Cmd = Disconnect]*)

DISCONNECT:
BEGIN
(*It only needs to be from the right*)
(*address too be taken seriously*)
(*If [packet is from right address] then*)
IF (PACKET.SOURCE = IPC.REMOTE_ADDR)
THEN
BEGIN
(*Turn off the time-out*)
TIMER.STOP;
(*Notify the session and application*)
(*layers*)
PACKET.NAME := 'RECEIVED DISCONNECT';
PACKET.SESSION_CMD := BREAK;
TO_APP.DEPOSIT (PACKET);
(*Go to Not_connected state*)
STATE := NOT_CONNECTED
END
(*Else*)
ELSE
(*Disregard it*)
BEGIN
STATE := CONNECTING
END
(*Endif*)
END;
(*When [Cmd = Time-out Accept]*)
T_Q_ACCEPT:
BEGIN
(*Send another inquiry*)
PACKET := IPC.BUFFER;
PACKET.TRANS_CMD := INQUIRE;
PACKET.NAME := 'ANOTHER INQUIRY';
SEND_NET (PACKET);
(*Start the timer again*)
TIMER.START (*, T_Q_ACCEPT)
END;
(*When [Cmd = Time-out Data Ack]*)
T_Q_DATA_ACK:
(*Disregard it*)
BEGIN
STATE := CONNECTING
END;
(*Otherwise*)
ELSE:
(*Resync on disconnect*)
BEGIN
PACKET.DIRECTION := OUTGOING;
PACKET.TRANS_CMD := DISCONNECT;
PACKET.SESSION_CMD := BREAK;
PACKET.STATUS := PACKET.STATUS
+ [NO_REMOTE_CONNECTION];
PACKET.S_CONNUN := IPC.LOCAL_CONNUN;
PACKET.DESTINATION := PACKET.SOURCE;
PACKET.SOURCE := THIS_NODE;
PACKET.NAME := 'DISASTER';
TO_NET.DEPOSIT (PACKET)
END
END(*CASE*);
(*Endif*)
(*When [State = In_Connection]*)
IN_CONNECTION:
(*If [packet is from the application layer] then*)
IF (PACKET.DIRECTION = OUTGOING) THEN
CASE PACKET.TRANS_CMD OF
(*When [Cmd = Disconnect]*)
DISCONNECT:
BEGIN
(*Go to Not_Connected state*)
STATE := NOT_CONNECTED;
(*Turn off the time-out*)
TIMER.STOP;
(*Re-set the IPC table*)
IPC.DATA_SEQ := 0;
IPC.ACK_SEQ := 0;
IPC.WAITING_FOR_ACK := FALSE;
(*There is currently nothing in this*)
(*@protocol to insure the disconnect*)
(*@command reaches the destination*)
(*Send the disconnect*)
PACKET.NAME := 'DISCONNECTING YOU';
SEND_NET (PACKET)
END;
(*When [Cmd = Data transfer]*)
DATA_XFER:
BEGIN
(*Set the sequence number of the packet*)
(*@for end to end protocol*)
PACKET.DATA_SEQ := IPC.DATA_SEQ;
IPC.WAITING_FOR_ACK := TRUE;
(*Send the packet and start a time-out*)
PACKET.NAME := 'DATA TRANSFER';

(* A-83 *)
IPC.BUFFER := PACKET;
SEND_NET (PACKET);
TIMER.START (4, T_Q_DATA_ACK)
END;
(*Otherwise*)
ELSE:
(*Resync on disconnect*)
BEGIN
TIMER.STOP;
PACKET.SESSION_CMD := BREAK;
PACKET.STATUS := PACKET.STATUS + [CONNECTION_BROKEN];
PACKET.DIRECTION := INCOMING;
PACKET.NAME := 'DISASTER';
TO_APP.DEPOSIT (PACKET);
PACKET.TRANS_CMD := DISCONNECT;
SEND_NET (PACKET);
STATE := NOT_CONNECTED
END
END(*CASE*)
(*Else from net*)
ELSE
CASE PACKET.TRANS_CMD OF
(*When [Cmd = Disconnect]*)
DISCONNECT:
(*It need only come from the right*)
(*address*)
BEGIN
(*If [packet is from right address] then*)
IF (PACKET.SOURCE = IPC.REMOTE_ADDR)
THEN
BEGIN
(*Turn off the time-out*)
TIMER.STOP;
(*Go to Not_Connected state*)
STATE := NOT_CONNECTED;
(*Re-set the IPC table*)
IPC.DATA_SEQ := 0;
IPC.ACK_SEQ := 0;
IPC.WAITING_FOR_ACK := FALSE;
(*Pass up piggy-backed messages*)
PACKET.NAME := 'WAS_DISCONNECTED';
TO_APP.DEPOSIT (PACKET)
END
(*Else*)
ELSE
(*Disregard it*)
BEGIN
STATE := IN_CONNECTION
END
(*Endif*)
END

(* A-84 *)
(*$* TRAN $*#)
(*$* Process to simulate Transport layer $*#)

(*When [Cmd = Inquire]*
INQUIRE:
(*This protocol can currently handle*)
(*only one connection at a time*)
BEGIN
(*If [not re-transmission from peer]*)
(then*)
IF (PACKET.S_CONNUM <> IPC.REMOTE_CONNUM)
OR (PACKET.SOURCE <> IPC.REMOTE_ADDR)
THEN
BEGIN
(*Disconnect them*)
PACKET.DIRECTION := OUTGOING;
PACKET.TRANS_CMD := DISCONNECT;
PACKET.DESTINATION := PACKET.SOURCE;
PACKET.SOURCE := THIS_NODE;
PACKET.NAME := 'CAN NOT TALK TO YOU ';
PACKET.STATUS := PACKET.STATUS
+ [BUSY];
TO_NET.DEPOSIT (PACKET)
END
(*Else*)
ELSE
BEGIN
(*Assumes the last thing sent*)
(*was an Accept*)
PACKET.TRANS_CMD := ACCEPT;
(*Re-transmit the last packet sent*)
PACKET.NAME := 'ANOTHER ACCEPT ';
SEND_NET (PACKET)
END
(*Endif*)
END;
(*When [Cmd = Data_Xfer]*)
DATA_XFER:
BEGIN
(*If [Packet is from partner] then*)
IF (PACKET.SOURCE = IPC.REMOTE_ADDR)
& (PACKET.S_CONNUM = IPC.REMOTE_CONNUM)
& (PACKET.D_CONNUM = IPC.LOCAL_CONNUM)
THEN
BEGIN
(*If [Packet has correct sequence]then*)
IF (PACKET.DATA_SEQ = IPC.ACK_SEQ) AND
(NOT IPC.WAITING_FOR_ACK) THEN
BEGIN
(*Increment the Ack sequence num*)
IPC.ACK_SEQ := (IPC.ACK_SEQ+1)
MOD 2;
(*Send the packet up*)
PACKET.NAME :=
'DATA RECEIVED ';

(* A-85 *)
(*$**$ TRANS $**$*)
(*$**$ Process to simulate Transport layer $**$*)

TO_APP.DEPOSIT (PACKET)
END;
(*Endif*)
IF (NOT IPC.WAITING_FOR_ACK) OR
(PACKET.DATA_SEQ <> IPC.ACK_SEQ) THEN
BEGIN
(*Send back an acknowledgment*)
PACKET.TRANS_CMD := DATA_ACK;
PACKET.STATUS := [];
PACKET.TEXT :=
PACKET.NAME :=
'DATA ACK';
PACKET.PRIORITY := HIGH_PRI;
PACKET.SECURITY := PUBLIC;
SEND_NET (PACKET)
END
(*ENDIF*)
ELSE
(*Else an illegal packet*)
ELSE
(*Wipe out connection that is sending*)
(*data illegally so it won't keep*)
(*timing out*)
BEGIN
PACKET.DIRECTION := OUTGOING;
PACKET.TRANS_CMD := DISCONNECT;
PACKET.DESTINATION := PACKET.SOURCE;
PACKET.SOURCE := THIS_NODE;
PACKET.NAME := 'CANT TALK TO YOU';
TO_NET.DEPOSIT (PACKET)
END
(*Endif*)
END;
(*When [Cmd = Data Acknoledgment]*)
DATA_ACK:
BEGIN
(*If [packet is from partner] then*)
IF (PACKET.SOURCE = IPC.REMOTE_ADDR)
& (PACKET.S_CONNUM = IPC.REMOTE_CONNUM)
& (PACKET.D_CONNUM = IPC.LOCAL_CONNUM)
& (PACKET.DATA_SEQ = IPC.DATA_SEQ) THEN
BEGIN
(*Turn off time-out*)
TIMER.STOP;
(* Increment the IPC data sequence no.*)
IPC.DATA_SEQ := (IPC.DATA_SEQ+1)
MOD 2;
IPC_WAITING_FOR_ACK := FALSE;
(*Unblock the application process*)
PACKET.SESSION_CMD := IMMEDIATE;
PACKET.NAME := 'RECEIVED AN ACK';
(* A-86 *)
(### TRANS ###)
(### Process to simulate Transport layer ###)

TQ_APP.DEPOSIT (PACKET)
END
(End)

END;

(When [Cmd = Time out data ack])
T_Q_DATA_ACK:
BEGIN

(Retransmit the last packet)
PACKET := IPC.BUFFER;
PACKET.NAME := 'DIDNT GET ACK RESEND';
SEND_NET (PACKET);

(Re-start the time-out)
TIMER.START (4, T_Q_DATA_ACK)
END;

(Otherwise)
ELSE:
BEGIN

(DISREGARD IT)
STATE := IN_CONNECTION
END

END(###CASE###)
(End)
END(###CASE###)
(End cycle)
END

(End Transport Process)
END;

(* A-87 *)
TYPE BAD_NODE_PROCESS = PROCESS (EVENT: MAIL_BOX_MONITOR;
   NEXT_NODE: MAIL_BOX_MONITOR);

(* BAD_NODE_PROCESS simulates a faulty node in a local area network, resulting in network packets being lost. *)

* Programmer: Ronald C. Albury
* Date Written: 10/11/82
* Copyright 1982 by Ronald C. Albury

* EXTERNAL *
* TYPE *
* MAIL_BOX_MONITOR: A monitor used for passing network packets between processes.
* PACKET_TYPE: Record structure of the network packet.

* INTERNAL *
* CONST *
* LOSS_FREQ: The frequency with which this node looses packets.

* VAR *
* PACKET: The network used by this process to communicate
* SKIP: An integer used to determine when a packet should be lost.

* PARAMETERS *
* EVENT: The monitor used by this process to receive packets from other nodes.
* NEXT_NODE: The monitor used by this process to send packets to other nodes.

CONST
   LOSS_FREQ = 10;
VAR
   PACKET: PACKET_TYPE;
   SKIP: INTEGER;

(*Begin Bad_Node process*)
BEGIN
   SKIP := 0;
   (*Cycle forever*)
   CYCLE
      (*Get a packet*)
      EVENT.GET (PACKET);
      SKIP := (SKIP+1) MOD LOSS_FREQ;
      (*If [this one is not lost] then*)
      IF (SKIP <> 0) THEN
         BEGIN
            (*Send it to the next node*)
            NEXT_NODE.DEPOSIT (PACKET)

(* A-88 *)
(*$*$ BADNODE $*$*$*)
(*$*$ Process to simulate data loss in the network $*$*$*)

END
(ELSE BREAKPNT (80)*
 (*Endif*)
 (*End cycle*)
 END
 (*End Bad_Node process*)
 END;

(* A-89 *)
TYPE WORKER_PROCESS = PROCESS(CONSOLE: RESOURCE_MONITOR;
    FROM_NET: MAIL_BOX_MONITOR;
    TO_NET: MAIL_BOX_MONITOR);

* The WORKER_PROCESS is an application layer process that
  transfers remote files to the operator console.

PROGRAMMER: RON ALBURY
DATE WRITTEN: 6/28/82
COMPUTER: INTERDATA 8/32
Copyright 1982 by Ronald C. Albury

EXTERNAL
CONST
   ERROR_MSG = Structured constant containing text
                 explanations of possible packet errors.

TYPE
   MESSAGE_TO_CLASS = A class that uses supervisory
                      calls to handle fixed record I/O to specified
                      logical units.
   PACKET_TYPE = Record structure of the network
                    packets.
   MESSAGE_TYPE = Array of characters.
   ERROR_SVC1_TYPE = Record structure of the status
                     bytes from the supervisory call.
   Mail_BOX_MONITOR = A monitor used for passing
                       packets between processes.
   RESOURCE_MONITOR = Allows only one process to
                       access a resource at a time.

INTERNAL
VAR
   STATUS_INDEX: Used to report packet errors.
   OP: Used to write lines of the transferred file
        to the operator.
   PACKET: A network packet this process uses to
           communicate with the network.
   TEXT: Array of characters used to communicate
         the operator.
   OP_STATUS: Receives the status bytes from the
              MESSAGE_TO_CLASS. Not used here, but necessary
              for the calls to OP.

PARAMETERS
CONSOLE: The RESOURCE_MONITOR used to reserve the
          console for exclusive I/O.
FROM_NET: The monitor used to recieve packets from
          the network.
TO_NET: The monitor used to send packets to the net

VAR
   OP: MESSAGE_TO_CLASS;

(*) A-90 *)
ERROR: ERROR_CLASS;
OS: NET_ID_CLASS;
NET_STATUS: Pkt_STATUS_TYPE;
TEXT: MESSAGE_TYPE;
NAME: MESSAGE_TYPE;
OP_STATUS: ERROR_SVC1_TYPE;

(*Begin Worker process*)
BEGIN
  (*Initialize the interface to the operator*)
  INIT OP;
  (*Initialize network interfaces*)
  INIT ERROR (CONSOLE);
  INIT OS (FROM_NET, TO_NET);
  (*Cycle forever*)
  CYCLE
    (*Read in session request*)
    CONSOLE.REQUEST;
    OP.WRITE (TERMINAL, 'ENTER SERVER HOST ', OP_STATUS);
    OP.READ (TERMINAL, NAME, OP_STATUS);
    OP.WRITE (TERMINAL, 'ENTER SERVER PASSWORD', OP_STATUS);
    OP.READ (TERMINAL, TEXT, OP_STATUS);
    CONSOLE.RELEASE;
    (*Request a session from the network*)
    OS.MAKE_SESSION (TEXT, NAME, NET_STATUS); (****)
    ERROR.REPORT ('WORKER 10 ', NET_STATUS);
    (*If we connect then*)
    IF (NET_STATUS = []) THEN
      BEGIN
        (*Repeat for all files desired*)
        REPEAT
          (*Read in the file request*)
          TEXT := 'ENTER FILE ID. - ';
          TEXT [18] := FIRST_FILE_ID;
          CONSOLE.REQUEST;
          OP.WRITE (TERMINAL, TEXT, OP_STATUS);
          OP.READ (TERMINAL, TEXT, OP_STATUS);
          CONSOLE.RELEASE;
          (*Send the request to the server*)
          OS.NET_WRITE (TEXT,IMMEDIATE,PUBLIC,NET_STATUS);
          ERROR.REPORT (' WORKER 20 ',NET_STATUS);
          (*Transfer the file to the console*)
          CONSOLE.REQUEST;
          (*Repeat until end of file*)
          REPEAT
            (*Get a line from the network*)
            OS.NET_READ (TEXT, NET_STATUS);
            (*Output it to the console*)
            OP.WRITE (TERMINAL, TEXT, OP_STATUS);
            (*End repeat*)

        (* A-91 *)
OR (SESSION_ENDING IN NET_STATUS);
CONSOLE.RELEASE;
ERROR.REPORT ('WORKER 30', NET_STATUS);
("Determine if more files are desired")
CONSOLE.REQUEST;
OP.WRITE (TERMINAL, 'MORE FILES Y/N', OP_STATUS);
OP.READ (TERMINAL, TEXT, OP_STATUS);
CONSOLE.RELEASE
("End repeat")
UNTIL (TEXT [1] = 'N');
("End that session")
OS.CLEAR_SESSION (NET_STATUS);
ERROR.REPORT ('WORKER 40', NET_STATUS)
END
("Endif")
("End cycle")
END
("End Worker process")
END;
TYPE SERVER_PROCESS = PROCESS (CONSOLE: RESOURCE_MONITOR;
FROM_NET: MAIL_BOX_MONITOR;
TO_NET: MAIL_BOX_MONITOR;
FIRST_NODE_FILE: CHAR;
LAST_NODE_FILE: CHAR);

(* The server process is an application layer process that
  does the disk I/O for a remote worker process. *)

* PROGRAMMER: RON ALBURY
* DATE WRITTEN: 6/28/82
* COMPUTER: INTERDATA 8/32

EXTERNAL

CONST
FILE_LU: An array of logical units that are
       subscripted with file id's. Used to look up
       logical unit of a requested file.

TYPE
MESSAGE_IO_CLASS = A class that uses supervisory
       calls to handle fixed record I/O to specified
       logical units.
PACKET_TYPE = Record structure of the network
       packets.
MESSAGE_TYPE = Array of characters.
ERROR_SVC1_TYPE = Record structure of the status
       bytes from the supervisory call.
MAIL_BOX_MONITOR = A monitor used for passing
       packets between processes.

INTERNAL

VAR
DISK: Used to input lines of a disk file.
PACKET: A network packet used to communicate with
       the network.
TEXT: Array of characters used for the disk I/O.
FILE_ID: The id of the file the worker process is
       requesting.
VALID_FILE_IDS: A set of the valid id's this
       process can access.
INDEX: Used in initializing Valid_File_Id's
OP_STATUS: Recieves the status bytes from the
MESSAGE_IO_CLASS.

PARAMETERS
FROM_NET: The monitor used to recieve packets from
       the network.
TO_NET: The monitor used to send packets to the net.

VAR
DISK: MESSAGE_IO_CLASS;
OS: NET_IO_CLASS;

(* A-93 *)
NET_STATUS: PKT_STATUS_TYPE;
ERROR: ERROR_CLASS;
TEXT: MESSAGE_TYPE;
FILE_ID: CHAR;
VALID_FILE_IDS: SET OF CHAR;
INDEX: FILE_RANGE;
OP_STATUS: ERROR_SVC1_TYPE;

(*Begin Server process*)
BEGIN
  (*Initialize the interface to the disk files*)
  INIT DISK;
  INIT OS (FROM_NET, TO_NET);
  INIT ERROR (CONSOLE);
  (**Build the set of all valid file id's at this node.*)
  (**For (all files at this node) do**) FOR INDEX := FIRST_NODE_FILE TO LAST_NODE_FILE DO (***)
  BEGIN
    (**Remember that it is a valid id**) VALID_FILE_IDS := VALID_FILE_IDS + [INDEX]
  END;
  (**Endfor**) (**Cycle forever**) CYCLE
  (**Put your ears up**) OS.NET_LISTEN (NET_STATUS);
  ERROR.REPORT ('SERVER 10 ', NET_STATUS);
  (**Get the request from the net**) OS.NET_READ (TEXT, NET_STATUS);
  ERROR.REPORT ('SERVER 15 ', NET_STATUS);
  FILE_ID := TEXT [1];
  (**While (we still have a session) do**) WHILE NOT(SESSIONENDING IN NET_STATUS) DO BEGIN
    (**If [it is a valid file id] then**) IF (FILE_ID IN VALID_FILE_IDS) THEN (**Transfer the file**) BEGIN
      (**Read in a line from the disk**) DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS);
      (**While not [end of file] do**) WHILE (OP_STATUS.DI = 0) AND (OP_STATUS.DD = 0) DO BEGIN
        (**Send it out on the network**) OS.NET_WRITE (TEXT, IMMEDIATE, SECRET,
          NET_STATUS);
        ERROR.REPORT ('SERVER 20 ', NET_STATUS);
        (**Read in a new line from the disk file**) DISK.READ (FILE_LU [FILE_ID], TEXT, OP_STATUS)
      END;
      (**Endwhile**) (**Endfor**)
    END;
  END;
  (**Endfor**)
END;

(* A-94 *)
(*Rewind the disk file*)

DISK.REWIND (FILE_LU [FILE_ID], GP_STATUS);
(*Send an EOF packet out on the network*)
OS.NET_WRITE ("/*
IMMEDIATE, PUBLIC, NET_STATUS);
ERROR.REPORT ("SERVER 30
", NET_STATUS);
END

(*Else (an invalid file id]*)
ELSE

(*Send an error message*)
BEGIN
TEXT := "/* BAD FILE ID - ";
TEXT [19] := FILE_ID;
OS.NET_WRITE (TEXT, IMMEDIATE, PUBLIC, NET_STATUS);
ERROR.REPORT ("SERVER 40
", NET_STATUS);
END;

(*Endif*)
(*Get the next request from the net*)
OS.NET_READ (TEXT, NET_STATUS);
FILE_ID := TEXT [1];
ERROR.REPORT ("SERVER 50
", NET_STATUS)
END

(*Endwhile*)
(*End cycle*)
END

(*End Server*)
END;
Appendix B

STERLING'S

PROTOCOLS
Introduction

This appendix presents the protocol specifications of the various layers of STERLING, describing them as finite state automata with variables. The behaviors of these automata, interacting with identical machines representing the peer layers, specifies the layers' protocols.

To aid the student with de-bugging modifications to STERLING, packet fields not essential to the automation are included in the specifications. It must be noted, however, that the inclusion of these fields defeats some of the advantages of layering, in regards to the documentation.
Layer: BLACKBOX

Initial State: BLOCKED

Packet Received (from host):
  Direction: OUTGOING
  Text: <any message>

Packet Delivered (to next node):
  Direction: INCOMING
  Text: <unchanged>

Resulting State: BLOCKED

Explanation:

The Blackbox receives a packet from the host, changes its direction flag, then sends it to the next node of the network.
Layer: BLACKBOX

Initial State: BLOCKED

Packet Received (from prior node):
   Direction: INCOMING
   Text: <any message>

Packet Delivered (to host):
   Direction: <unchanged>
   Text: <unchanged>

Resulting State: BLOCKED

Explanation:
The Blackbox receives a packet from the network and passes it up to its host.
Layer: PRESENTATION

Initial State: BLOCKED

Packet Received (from host):
  Direction: OUTGOING
  Security: SECRET
  Text: <message with host delimiter>

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Text: <encrypted message with host delimiter>

Resulting State: BLOCKED

Explanation:
The Presentation layer receives a packet from the Application layer with a request for encrypted transmission. It encrypts the message, sets the file format field to indicate which record delimiter is used, and sends the packet to the net. The file_format field does not have a valid value until the field is set by the Presentation layer.
Layer: PRESENTATION

Initial State: BLOCKED

Packet Received (from host):
  Direction: OUTGOING
  Security: PUBLIC
  Text: <message with host delimiter>

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Text: <unchanged>

Resulting State: BLOCKED

Explanation:
The Presentation layer receives a transmission packet from the Application layer. It sets the file format field to indicate which record delimiter is used, and sends the packet to the network. The file_format field does not have a valid value until the field is set by the Presentation layer.
Layer: PRESENTATION

Initial State: BLOKED

Packet Received (from network):
   Direction: INCOMING
   Security: SECRET
   File Format: CR_DELIM or NL_DELIM (source delimiter)
   Text: <encrypted message with source delimiter>

Packet Delivered (to host):
   Direction: <unchanged>
   Security: PUBLIC
   File Format: CR_DELIM or NL_DELIM (host delimiter)
   Text: <message with host delimiter>

Resulting State: BLOKED

Explanation:

The Presentation layer receives a packet that is labeled secret from the network. It decrypts the message, makes sure the record delimiter is compatible with its host, and sends the message to the Application layer.
Layer: PRESENTATION

Initial State: BLOCKED

Packet Received (from network):
Direction: INCOMING
Security: PUBLIC
File Format: CR_DELIM or NL_DELIM (source delimiter)
Text: <message with source delimiter>

Packet Delivered (to host):
Direction: <unchanged>
Security: <unchanged>
File Format: CR_DELIM or NL_DELIM (host delimiter)
Text: <message with host delimiter>

Resulting State: BLOCKED

Explanation:
The Presentation layer receives a packet from the network. It makes sure the record delimiter is compatible with its host, and sends the message to the Application layer.
Layer: APPLICATION

Initial State: ACTIVE

Network Call: NET_LISTEN

Packet Delivered (to network):
  Direction: OUTGOING
  Security: PUBLIC
  Priority: HIGH_PRI
  Session_Cmd: LISTEN
  Text: '

Resulting State: BLOCKED

Explanation:
The Application layer issues a packet commanding the Session layer to listen for a session request from the network.
Layer: APPLICATION

Initial State: ACTIVE

Network Call: MAKE_SESSION

Packet Delivered (to network):
  Direction: OUTGOING
  Security: PUBLIC
  Priority: HIGH_PRI
  Session_Cmd: ESTABLISH
  Status: []
  Text: <destination password>

Resulting State: BLOCKED

Explanation:

The Application layer issues a packet commanding the Session layer to request a session on the network.
Layer: APPLICATION

Initial State: ACTIVE

Network Call: CLEAR_SESSION

Packet Delivered (to network):
  Direction: OUTGOING
  Security: PUBLIC
  Priority: LOW_PRI
  Session_Cmd: BREAK
  Status: []
  Text: '

Resulting State: BLOCKED

Explanation:

The Application layer issues a packet commanding the Session layer to terminate its session.
Layer: APPLICATION

Initial State: ACTIVE

Network Call: NET_READ

Resulting State: BLOCKED

Explanation:

The Application layer is attempting to receive a packet from its session.
Layer: APPLICATION

Initial State: ACTIVE

Network Call: NET_WRITE

Packet Delivered (to network):
- Direction: OUTGOING
- Security: SECRET or PUBLIC (application program's option)
- Priority: MED_PRI
- Session_Cmd: IMMEDIATE, CHAIN, END_CHAIN, or ABORT_CHAIN (application program's option)
- Status: []
- Text: <message with host delimiter>

Resulting State: BLOCKED

Explanation:

The Application layer is attempting to send a message on the network.
Layer: SESSION

Initial State: NO SESSION

Packet Received (from host):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: LISTEN
  Status: []
  Text:

Resulting State: LISTENING

Explanation:

The Session layer receives a packet from the Application layer asking it to listen for a session request. The Session layer goes into a listening state and waits for a request to arrive.
Layer: SESSION

Initial State: NO SESSION

Packet Received (from host):
   Direction: OUTGOING
   Security: PUBLIC
   File Format: CR_DELIM or NL_DELIM (host delimiter)
   Priority: HIGH_PRI
   Session_Cmd: ESTABLISH
   Status: []
   Text: <destination password>

Packet Delivered (to network):
   Direction: <unchanged>
   Security: <unchanged>
   File Format: <unchanged>
   Priority: <unchanged>
   Session_Cmd: REQUEST
   Status: <unchanged>
   Text: <unchanged>

Resulting State: REQUESTING

Explanation:
The Session layer receives a packet from the Application
layer asking it to establish a session with a remote site. The Session layer goes into the requesting state and sends a request onto the network.
Layer: SESSION

Initial State: REQUESTING

Packet Received (from network):

  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: START
  Status: []
  Text: <source password>

Packet Delivered (to host):

  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Status: <unchanged>
  Text: <unchanged>

Resulting State: IN_SESSION

Explanation:

The Session layer receives a start signal from its peer
session. The Session layer sends this packet to its
Application layer, unblocking it, and goes to an in-session
state. The file format and text have these values because
the packet was echoed by the source's Session layer.
Layer: SESSION

Initial State: REQUESTING

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: BREAK
  Status: + [BAD_PASSWORD]
  Text: <incorrect password>

Packet Delivered (to host):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Status: + [BAD_PASSWORD, SESSION_ENDING]
  Text: <incorrect password>

Resulting State: NO_SESSION

Explanation:

The Session layer receives a packet from its peer
rejecting its request for a session. It resumes a no-session state and sends the rejection packet to its Application layer to unblock it. The file_format and text have these values because the packet was echoed by the source's Session layer.
Layer: SESSION

Initial State: LISTENING

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Status: []
  Text: <host password>

Packet Delivered (to host):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: START
  Status: <unchanged>
  Text: <unchanged>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
Priority: <unchanged>
Session_Cmd: START
Status: <unchanged>
Text: <unchanged>

Resulting State: IN_SESSION

Explanation:
The Session layer, while waiting for a session request, receives one with a valid password. It sends back a session start packet to its peer session, and a copy of the packet to its Application layer to unblock it. The Session layer is now in-session.
Layer: SESSION

Initial State: LISTENING

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Status: []
  Text: <incorrect password>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: BREAK
  Status: + [BAD_PASSWORD]
  Text: <unchanged>

Resulting State: LISTENING

Explanation:

The Session layer, while waiting for a session request,
receives one with an invalid password. It sends back a rejection packet to the originator of the request and remains in a listening state.
Layer: SESSION

Initial State: IN SESSION

Packet Received (from host):
  Direction: OUTGOING
  Security: SECRET or PUBLIC (application program's option)
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: MED_PRI
  Session_Cmd: IMMEDIATE, CHAIN, END_CHAIN, or ABORT_CHAIN
  Status: []
  Text: <any message>

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Status: <unchanged>
  Text: <unchanged>

Packet Delivered (to host):
  Direction: INCOMING
  Security: <unchanged>
  File_Format: <unchanged>
Priority: <unchanged>
Session_Cmd: <unchanged>
Status: <unchanged>
Text: <unchanged>

Resulting State: IN_SESSION

Explanation:

The Session layer receives a data packet from the Application layer. It transparently passes the packet to the network and sends a copy back to the Application layer indicating a successful transfer.
Layer: SESSION

Initial State: IN_SESSION

Packet Received (from host):
  Direction: OUTGOING
  Security: SECRET or PUBLIC (application program's option)
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: LOW_PRI
  Session_Cmd: BREAK
  Status: []
  Text: '

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Status: <unchanged>
  Text: <unchanged>

Packet Delivered (to host):
  Direction: INCOMING
  Security: <unchanged>
  File_Format: <unchanged>
Priority: <unchanged>
Session_Cmd: <unchanged>
Status: + [SESSION_ENDING]
Text: <unchanged>

Resulting State: NO_SESSION

Explanation:

The Session layer receives a command from the Application layer to terminate the session. It sends the termination command on the network to its peer session and sends a copy of the packet to the Application layer to indicate a successful termination.
Layer: SESSION

Initial State: IN SESSION

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: LOW_PRI
  Session_Cmd: BREAK
  Status: []
  Text: ' ' 

Packets Delivered (to host from chain buffer):
  Direction: INCOMING
  Security: SECRET or PUBLIC (sender's option)
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: MED_PRI
  Session_Cmd: CHAIN
  Status: []
  Text: <any message>

Packet Delivered (to host):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
Priority: <unchanged>
Status: + [SESSION_ENDING]
Text: <unchanged>

Explanation:

The Session layer receives a packet from the network commanding it to terminate the session. It sends all messages it had stored in the chain buffer to the Application layer, then follows them with the termination packet.
Layer: SESSION

Initial State: IN_SESSION

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Status: []
  Text: <host password>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: BREAK
  Status: + [BUSY]
  Text: <unchanged>

Resulting State: IN_SESSION

Explanation:

The Session layer is already involved in a session when
it receives a request for a session from another process.
It refuses the request and remains in its original session.
Layer: SESSION

Initial State: IN_SESSION

Packet Received (from network):
  Direction: INCOMING
  Security: SECRET or PUBLIC (sender's option)
  File_Format: CR_DELIM or ML_DELIM (source delimiter)
  Priority: MED_PRI
  Session_Cmd: IMMEDIATE
  Status: []
  Text: <any message>

Packet Delivered (to host):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Status: <unchanged>
  Text: <unchanged>

Resulting State: IN_SESSION

Explanation:
  The layer receives a data transfer packet from the
network with an IMMEDIATE command and transparently forwards it to the Application layer.
Layer: SESSION

Initial State: IN_SESSION

Packet Received (from network):
  Direction: INCOMING
  Security: SECRET or PUBLIC (sender's option)
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: MED_PRI
  Session_Cmd: CHAIN
  Status: []
  Text: <any message>

Resulting State: IN_SESSION

Explanation:

The layer receives a data packet with a command that it
be chained, and it quarantines the packet in a buffer.
Layer: SESSION

Initial State: IN SESSION

Packet Received (from network):

Direction: INCOMING
Security: SECRET or PUBLIC (sender's option)
File_Format: CR_DELIM or NL_DELIM (source delimiter)
Priority: MED_PRI
Session_Cmd: END_CHAIN
Status: []
Text: <any message>

Packets Delivered (to host from chain buffer):

Direction: INCOMING
Security: SECRET or PUBLIC (sender's option)
File_Format: CR_DELIM or NL_DELIM (source delimiter)
Priority: MED_PRI
Session_Cmd: CHAIN
Status: []
Text: <any message>

Packet Delivered (to host):

Direction: <unchanged>
Security: <unchanged>
File_Format: <unchanged>
Priority: <unchanged>
Status: <unchanged>
Text: <unchanged>

Resulting State: IN_SESSION

Explanation:
The layer receives a data transfer packet with an END_CHAIN command. It sends all messages in the chain buffer to the Application layer and follows them with the new packet.
Layer: SESSION

Initial State: IN_SESSION

Packet Received (from network):
   Direction: INCOMING
   Security: SECRET or PUBLIC (sender's option)
   File_Format: CR_DELIM or NL_DELIM (source delimiter)
   Priority: MED_PRI
   Session_Cmd: ABORT_CHAIN
   Status: []
   Text: <any message>

Resulting State: IN_SESSION

Explanation:
   The layer receives a packet from the network with an
   ABORT_CHAIN command. It emptys the chain buffer and
   discards the packet.
Layer: APPLICATION (revised)

Initial State: ACTIVE

Network Call: MAKE_SESSION

Packet Delivered (to network):
  Direction: OUTGOING
  Security: PUBLIC
  Priority: HIGH_PRI
  Session_Cmd: ESTABLISH
  Name: <destination name>
  Status: []
  Text: <destination password>

Resulting State: BLOCKED

Explanation:
The Application layer issues a packet commanding the Session layer to request a session on the network. The name of the destination is included to allow the Transport layer to look up the destination address.
Layer: SESSION (revised)

Initial State: NO SESSION

Packet Received (from host):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: ESTABLISH
  Name: <destination name>
  Status: []
  Text: <destination password>

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: REQUEST
  Name: <unchanged>
  Trana_Cmd: CONNECT
  Status: <unchanged>
  Text: <unchanged>

Resulting State: REQUESTING
Explanation:

The Session layer receives a packet from the Application layer asking it to establish a session with a remote site. The Session layer goes into the requesting state and commands the Transport layer to make a connection and deliver the request across the network. The trans_cmd field does not have a valid value until it is set by the Session layer.
Layer: SESSION (revised)

Initial State: LISTENING

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: 'GOT AN INQUIRE'
  Status: []
  Text: <host password>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: START
  Name: 'SESSION STARTS'
  Trans_Cmd: DATA_XFER
  Status: <unchanged>
  Text: <unchanged>

Resulting State: IN_SESSION
Explaination:

The Session layer, while waiting for a session request, receives one with a valid password. It commands the Transport layer to deliver a session start packet to its peer session. The Application layer will be unblocked by the Transport layer's ACK. The Session layer is now in-session. The trans_omd field does not have a valid value until it is set by the Session layer.
Layer: SESSION (revised)

Initial State: LISTENING

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: 'GOT AN INQUIRE'
  Status: []
  Text: <incorrect password>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: BREAK
  Name: 'BAD PASSWORD'
  Trans_Cmd: DISCONNECT
  Status: + [BAD_PASSWORD]
  Text: <unchanged>

Resulting State: LISTENING
Explanation:

The Session layer, while waiting for a session request, receives one with an invalid password. It commands the Transport layer to deliver a rejection packet to its peer session and remains in a listening state.
Layer: SESSION (revised)

Initial State: IN_SESSION

Packet Received (from host):
  Direction: OUTGOING
  Security: SECRET or PUBLIC (application program's option)
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: MED_PRI
  Session_Cmd: IMMEDIATE, CHAIN, END_CHAIN, or ABORT_CHAIN
  Name: 'APPLI CMD WRITE'
  Status: []
  Text: <any message>

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: <unchanged>
  Trans_Cmd: DATA_XFER
  Status: <unchanged>
  Text: <unchanged>

Resulting State: IN_SESSION
Explanation:

The Session layer receives a data packet from the Application layer. It transparently passes the packet to the Transport layer as a data transfer. The Transport layer's ACK will unblock the Application layer.
Layer: SESSION (revised)

Initial State: IN SESSION

Packet Received (from host):
  Direction: OUTGOING
  Security: SECRET or PUBLIC (application program's option)
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: LOW_PRI
  Session_Cmd: BREAK
  Name: 'APPLI CMD CLEAR SESS'
  Status: []
  Text: ''

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Name: 'SESSION ENDING'
  Session_Cmd: BREAK
  Trans_Cmd: DISCONNECT
  Status: <unchanged>
  Text: <unchanged>

Packet Delivered (to host):
Direction: INCOMING
Security: <unchanged>
File_Format: <unchanged>
Priority: <unchanged>
Name: 'SESSION ENDING'
Status: + [SESSION_ENDING]
Text: <unchanged>

Resulting State: NO_SESSION

Explanation:

The Session layer receives a command from the Application layer to terminate the session. It sends the termination command on the network to its peer session piggy-backed on a Transport disconnect and sends a copy of the packet to the Application layer to indicate a successful termination.
Layer: TRANSPORT

Initial State: NOT CONNECTED

Packet Received (from host):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: <destination name>
  Trans_Cmd: CONNECT
  Status: []
  Text: <destination password>

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: 'TRANS INQUIRE'
  Trans_Cmd: INQUIRE
  S_Connum: <local connection number>
  D_Connum: 0
  Source: <host address>
Destination: <remote address>
Status: <unchanged>
Text: <unchanged>

Resulting State: CONNECTING

Explanation:

The Transport layer receives a command to establish a connection with a remote site. It sets up a connection management data structure and issues the inquiry to the remote site. The destination's connection number is not yet known, so the d_connum field has no meaning.
Layer: TRANSPORT

Initial State: NOT_CONNECTED

Packet Received (from host):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: <illegal name>
  Trans_Cmd: CONNECT
  Status: []
  Text: <destination password>

Packet Delivered (to host):
  Direction: INCOMING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: BREAK
  Name: 'ERROR MSG NO HOST'
  Status: + [NO_SUCH_HOST]
  Text: <unchanged>

Resulting State: NOT_CONNECTED
Explanation:

The Transport layer receives a command to establish a connection with a remote site. It is unable to find an address for the destination, so it sends back an error packet.
Layer: TRANSPORT

Initial State: NOT CONNECTED

Packet Received (from network):

Direction: INCOMING
Security: PUBLIC
File_Format: CR_DELIM or NL_DELIM (source delimiter)
Priority: HIGH_PRI
Session_Cmd: REQUEST
Name: 'TRANS INQUIRE '
      or 'ANOTHER INQUIRY '
Trans_Cmd: INQUIRE
S_Connum: <source connection number>
D_Connum: 0
Source: <remote address>
Destination: <host address>
Status: []
Text: <host password>

Packet Delivered (to network):

Direction: OUTGOING
Security: <unchanged>
File_Format: <unchanged>
Priority: <unchanged>
Session_Cmd: <unchanged>
Name: 'ACCEPTANCE
Trans_Cmd: ACCEPT
S_Connum: <local connection number>
D_Connum: <remote connection number>
Source: <host address>
Destination: <remote address>
Status: <unchanged>
Text: <unchanged>

Packet Delivered (to host):
Direction: INCOMING
Security: <unchanged>
File_Format: <unchanged>
Priority: <unchanged>
Session_Cmd: <unchanged>
Name: 'GOT AN INQUIRE
Status: <unchanged>
Text: <unchanged>

Resulting State: IN_CONNECTION

Explanation:
The Transport layer receives a connection inquiry. It agrees to the connection and sends packets to the host and to the source of the inquiry.
Layer: TRANSPORT

Initial State: CONNECTING

Packet Received (from net):

Direction: INCOMING
Security: PUBLIC
File_Format: <host delimiter>
Priority: HIGH_PRI
Session_Cmd: REQUEST
Name: 'ACCEPTANCE'
Trans_Cmd: ACCEPT
S_Connum: <remote connection number>
D_Connum: <host connection number>
Source: <remote address>
Destination: <host address>
Status: []
Text: <source password>

Resulting State: IN_CONNECTION

Explanation:

The Transport layer receives a packet accepting its connection inquiry. It completes the connection management data structure and goes to the in-connection state. The file_format, session_cmd, and text fields have these values
because the packet was echoed by the source's Transport layer.
Layer: TRANSPORT

Initial State: CONNECTING

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: 'CANT NOT TALK TO YOU'
  Trans_Cmd: DISCONNECT
  S_ConnNum: <remote connection number>
  D_ConnNum: <host connection number>
  Source: <remote address>
  Destination: <host address>
  Status: []
  Text: <source password>

Packet Delivered (to host):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: BREAK
  Name: 'RECEIVED DISCONNECT'
Status: <unchanged>

Text: <unchanged>

Resulting State: NOT_CONNECTED

Explanation:

The Transport layer attempts to make a connection with a peer layer that is already connected. It passes the rejection packet to the host and resumes a not-connected state. The file_format and text have these values because the packet was echoed by the source's Transport layer.
Layer: TRANSPORT

Initial State: CONNECTING

Packet Received (from time-out):
  Direction: INCOMING
  Priority: HIGH_PRI
  Name: 'THIS IS A TIME OUT'
  Trans_Cmd: T_Q_ACCEPT
  Status: []

Packet Delivered (to net):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: <unchanged>
  Session_Cmd: REQUEST
  Name: 'ANOTHER INQUIRY'
  Trans_Cmd: INQUIRE
  S_Connum: <host connection number>
  D_Connum: 0
  Source: <host address>
  Destination: <remote address>
  Status: <unchanged>
  Text: <destination password>
Resulting State: CONNECTING

Explanation:
The Transport layer is timed out while waiting for a response from its peer layer, so it sends another connection inquiry.
Layer: TRANSPORT

Initial State: IN CONNECTION

Packet Received (from host):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: LOW_PRI
  Session_Cmd: BREAK
  Name: 'SESSION ENDING'
  Trans_Cmd: DISCONNECT
  Status: []
  Text: '

Packet Delivered (to network):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: 'DISCONNECTING YOU'
  Trans_Cmd: DISCONNECT
  S_Connum: <host connection number>
  D_Connum: <remote connection number>
  Source: <host address>
Destination: <remote address>
Status: <unchanged>
Text: <unchanged>

Resulting State: Not_Connected

Explanation:
The Transport layer receives a disconnect packet from its host and breaks the connection.
Layer: TRANSPORT

Initial State: IN_CONNECTION

Packet Received (from host):
  Direction: OUTGOING
  Security: SECRET or PUBLIC (application program's option)
  File_Format: CR_DELIM or NL_DELIM (host delimiter)
  Priority: HIGH_PRI or MED_PRI
  Session_Cmd: START, IMMEDIATE, CHAIN, END_CHAIN, or ABORT_CHAIN
  Name: 'APPLI CMD WRITE' or 'SESSION STARTS'
  Trans_Cmd: DATA_XFER
  Status: []
  Text: <any message>

Packet Delivered (to net):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: 'DATA TRANSFER'
  Trans_Cmd: <unchanged>
  S_Connum: <host connection number>
  D_Connum: <remote connection number>
Source: <host address>
Destination: <remote address>
Data_Seq: <current sequence number>
Status: <unchanged>
Text: <unchanged>

Resulting State: IN_CONNECTION

Explanation:
The Transport layer receives a data transfer packet from its host and sends it on the network connection.
Layer: TRANSPORT

Initial State: IN CONNECTION

Packet Received (from network):

  Direction: INCOMING
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: LOW_PRI
  Session_Cmd: BREAK
  Name: 'DISCONNECTING YOU'
  Trans_Cmd: DISCONNECT
  S_Connum: <remote connection number>
  D_Connum: <host connection number>
  Source: <remote address>
  Destination: <host address>
  Status: []
  Text: '

Packet Delivered (to host):

  Direction: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: 'WAS DISCONNECTED'
  Status: <unchanged>
  Text: <unchanged>
Resulting State: NOT_CONNECTED

Explanation:

The Transport layer is disconnected by its peer layer. It passes the disconnect packet to the host and goes to a not-connected state.
Layer: TRANSPORT

Initial State: IN CONNECTION

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: 'TRANS INQUIRE' or 'ANOTHER INQUIRY'
  Trans_Cmd: INQUIRE
  S_Connum: <incorrect remote connection number>
  D_Connum: 0
  Source: <incorrect remote address>
  Destination: <host address>
  Status: []
  Text: <host password>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: 'CAN NOT TALK TO YOU'

B-67
Trans_Cmd: DISCONNECT
S_Connum: <unchanged>
D_Connum: <unchanged>
Source: <host address>
Destination: <originating address>
Status: + [BUSY]
Text: <unchanged>

Resulting State: IN_CONNECTION

Explanation:
The Transport layer receives an inquiry while it is already connected to a different process and sends back a busy signal.
Layer: TRANSPORT

Initial State: IN CONNECTION

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI
  Session_Cmd: REQUEST
  Name: 'ANOTHER INQUIRY'
  Trans_Cmd: INQUIRE
  S_Connum: <current remote connection number>
  D_Connum: 0
  Source: <current remote address>
  Destination: <host address>
  Status: []
  Text: <host password>

Packet Delivered (to network):
  Direction: OUTGOING
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
  Session_Cmd: <unchanged>
  Name: 'ANOTHER ACCEPT'
- N E T 4 -

Trans_Cmd: ACCEPT
S_Connum: <host connection number>
D_Connum: <remote connection number>
Source: <host address>
Destination: <remote address>
Status: <unchanged>
Text: <unchanged>

Resulting State: IN_CONNECTION

Explanation:

The Transport layer's peer process timed out waiting for an accept and issued another inquiry. The Transport layer returns another accept and remains in-connection.
Layer: TRANSPORT

Initial State: IN CONNECTION (waiting-for-ack)

Packet Received (from network):
- Direction: INCOMING
- Security: PUBLIC or SECRET (sender's option)
- File_Format: CR_DELIM or NL_DELIM (source delimiter)
- Priority: HIGH_PRI or MED_PRI
- Session_Cmd: START, IMMEDIATE, CHAIN, END_CHAIN, or ABORT_CHAIN
- Name: 'DATA TRANSFER' or 'DIDNT GET ACK RESEND'
- Trans_Cmd: DATA_XFER
- S_Connum: <remote connection number>
- D_Connum: <host connection number>
- Source: <remote address>
- Destination: <host address>
- Data_Seq: <any>
- Status: []
- Text: <any message>

Explanation:

The Transport layer is waiting for an acknowledgment of its last data transfer but, instead, receives a data transfer from its peer layer. The Transport layer cannot pass the incoming data to its host, because the Application
layer is blocked waiting for the acknowledgment, so it discards the new packet. NOTE: Waiting for the acknowledgment of a data transmission could be implemented as another state for the Transport layer, but for historical reasons it was implemented with a Boolean variable.
Layer: TRANSPORT

Initial State: IN CONNECTION (Not waiting-for-ack)

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC or SECRET (sender's option)
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI or MED_PRI
  Session_Cmd: START, IMMEDIATE, CHAIN, END_CHAIN, or ABORT_CHAIN
  Name: 'DATA TRANSFER' or 'DIDNT GET ACK REXMIT'
  Trans_Cmd: DATA_XFER
  S_Connum: <remote connection number>
  D_Connum: <host connection number>
  Source: <remote address>
  Destination: <host address>
  Data_Seq: <correct>
  Status: []
  Text: <any message>

Packet Delivered (to host):
  Direction: <unchanged>
  Security: <unchanged>
  File_Format: <unchanged>
  Priority: <unchanged>
Session_Cmd: <unchanged>
Name: 'DATA RECEIVED'
Status: <unchanged>
Text: <unchanged>

Packet Delivered (to net):
Direction: OUTGOING
Security: PUBLIC
File_Format: <unchanged>
Priority: HIGH_PRI
Session_Cmd: <unchanged>
Name: 'DATA ACK'
Trans_Cmd: DATA_ACK
S_Connum: <host connection number>
D_Connum: <destination connection number>
Source: <host address>
Destination: <remote address>
Data_Seq: <unchanged>
Status: []
Text: '

Resulting State: IN_CONNECTION

Explanation:
The Transport layer receives a data packet from the network, passes the packet to its host, and returns an
acknowledgment.
Layer: TRANSPORT

Initial State: IN CONNECTION

Packet Received (from network):
  Direction: INCOMING
  Security: PUBLIC or SECRET (sender's option)
  File_Format: CR_DELIM or NL_DELIM (source delimiter)
  Priority: HIGH_PRI or MED_PRI
  Session_Cmd: START, IMMEDIATE, CHAIN, END_CHAIN, or
      ABORT_CHAIN
  Name: 'DIDNT GET ACK RESEND'
  Trans_Cmd: DATA XFER
  S_Connum: <remote connection number>
  D_Connum: <host connection number>
  Source: <remote address>
  Destination: <host address>
  Data_Seq: <incorrect>
  Status: []
  Text: <any message>

Packet Delivered (to net):
  Direction: OUTGOING
  Security: PUBLIC
  File_Format: <unchanged>
  Priority: HIGH_PRI
Session_Cmd: <unchanged>
Name: 'DATA ACK'
Trans_Cmd: DATA_ACK
S_Connum: <host connection number>
D_Connum: <destination connection number>
Source: <host address>
Destination: <remote address>
Data_Seq: <unchanged>
Status: []
Text: ''

Explanation:

The Transport layer's peer process timed out waiting for an acknowledgment of a data transfer and sent the packet again. The Transport layer had already received the original data packet and had recognized the new packet as a duplicate because of the incorrect sequence number, so it re-transmitted an acknowledgment of the packet.
Layer: TRANSPORT

Initial State: IN_CONNECTION

Packet Received (from network):

Direction: INCOMING
Security: PUBLIC
File_Format: CR_DELIM or NL_DELIM (host delimiter)
Priority: HIGH_PRI
Session_Cmd: START, IMMEDIATE, CHAIN, END_CHAIN, or
          ABORT_CHAIN (echo of data-xfer)
Name: 'DATA ACK'

Trans_Cmd: DATA ACK
S_Connum: <remote connection number>
D_Connum: <host connection number>
Source: <remote address>
Destination: <host address>
Data_Seq: <correct>
Status: []
Text: '

Packet Delivered (to host):

Direction: <unchanged>
Security: <unchanged>
File_Format: <unchanged>
Priority: <unchanged>
Session_Cmd: IMMEDIATE

Name: 'RECEIVED AN ACK'

Status: <unchanged>

Text: <unchanged>

Resulting State: IN_CONNECTION

Explanation:

The Transport layer receives an acknowledgment of its last data transfer and passes it to its host to unblock the application layer.
Layer: TRANSPORT

Initial State: IN CONNECTION

Packet Received (from time-out):
  Direction: INCOMING
  Priority: HIGH_PRI
  Name: 'THIS IS A TIME OUT'
  Trans_Cmd: T O DATA ACK
  Status: []

Packet Delivered (to net):
  Direction: OUTGOING
  Security: <repeat of last x-mit>
  File_Format: <repeat of last x-mit>
  Priority: <repeat of last x-mit>
  Session_Cmd: <repeat of last x-mit>
  Name: 'DIDNT GET ACK REXMIT'
  Trans_Cmd: <repeat of last x-mit>
  S_Connun: <host connection number>
  D_Connun: <remote connection number>
  Source: <host address>
  Destination: <remote address>
  Data_Seq: <current sequence number>
  Status: <repeat of last x-mit>
  Text: <repeat of last x-mit>
Resulting State: IN_CONNECTION

Explanation:
The Transport layer times out without receiving an acknowledgment of its last data transfer, so it re-transmits the packet.
STERLING: A PEDAGOGICAL IMPLEMENTATION
OF THE
ISO MODEL FOR OPEN SYSTEM INTERCONNECTION

by

Ronald Curtis Albury

B.S. Rochester Institute of Technology 1976

-------------------

AN ABSTRACT OF A MASTER'S REPORT

Submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1983
This report describes the design and implementation of programs which demonstrate some of the functions of a computer network. The programs are patterned on the International Standards Organization's (ISO) model for computer interconnection and are intended to be used as an aid in teaching that model to graduate level students. The programs are fully documented and designed to be easily understood and portable between machines. The report discusses the principles the ISO used in deriving its model and how those principles relate to the implementation's design. The report restricts itself to the upper layers of the ISO model and concludes with the discussion of the transport layer.