CASE STUDY I

FINAL REPORT DEVELOPED FOR
LARGE SCALE SOFTWARE SYSTEM DESIGN
OF THE
AN/TYC-39 STORE AND FORWARD
MESSAGE SWITCH
USING
THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME IV OF IV

GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101
CASE STUDY I

FINAL REPORT DEVELOPED FOR
LARGE SCALE SOFTWARE SYSTEM DESIGN
OF THE
AN/TYC-39 STORE AND FORWARD
MESSAGE SWITCH
USING
THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME IV OF IV

GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101
**Ada Capability Study: Design of the Message Switching System AN/TYC-39 Using the Ada Programming Language**

### Abstract

An Ada oriented framework for the design and documentation of the U.S. Army TYC-39 store and forward message switch (military software) system is presented. This document package contains a Requirements, Design, Ada Integrated Methodology, and Final Report section. A methodology to use Ada in specifying requirements, design, and the implementation of a system was developed. This methodology was used to redesign the TYC-39 message switch system. A selected software module was programmed after the redesign.

### Document Analysis a. Descriptors

- Ada Programming Language
- Software Design with Ada
- Designing with Ada

b. Identifiers/Open-Ended Terms

- Message Switch
- Military Software
- Program Design Language

c. COSATI Field/Group

18. Availability Statement:


19. Security Class (This Report) UNCLASSIFIED

21. No. of Pages 521

22. Price UNCLASSIFIED
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. System Design</td>
<td></td>
</tr>
<tr>
<td>2.1 System Hierarchy</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Run Switch</td>
<td></td>
</tr>
<tr>
<td>2.2.1 Overview</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2 Operation</td>
<td>6</td>
</tr>
<tr>
<td>2.2.3 Functions</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Internal Message Structure</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Functional Decomposition and Interconnectivity</td>
<td>22</td>
</tr>
<tr>
<td>2.4.1 Message Input</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2 Message Queueing</td>
<td>42</td>
</tr>
<tr>
<td>2.4.3 Message Routing</td>
<td>47</td>
</tr>
<tr>
<td>2.4.4 Output Message</td>
<td>56</td>
</tr>
<tr>
<td>2.4.5 Manage Intransit and Manage Overflow</td>
<td>65</td>
</tr>
<tr>
<td>2.5 Data Structure Diagrams</td>
<td>68</td>
</tr>
<tr>
<td>3. Detail Design</td>
<td></td>
</tr>
<tr>
<td>3.1 Ada Unit Specification</td>
<td>77</td>
</tr>
<tr>
<td>3.1.1 Message Input</td>
<td>77</td>
</tr>
<tr>
<td>3.1.2 Message Queueing</td>
<td>81</td>
</tr>
<tr>
<td>3.1.3 Message Routing</td>
<td>86</td>
</tr>
<tr>
<td>3.1.4 Output Message</td>
<td>88</td>
</tr>
<tr>
<td>3.1.5 Manage Intransit and Manage Overflow</td>
<td>91</td>
</tr>
<tr>
<td>3.1.6 Support Routines</td>
<td>94</td>
</tr>
<tr>
<td>3.2 Package Specification Dependencies</td>
<td>115</td>
</tr>
<tr>
<td>3.3 Traceability</td>
<td>117</td>
</tr>
<tr>
<td>3.4 Data Structures</td>
<td>127</td>
</tr>
<tr>
<td>3.5 Rationale for Hardware/Software Partitioning</td>
<td>136</td>
</tr>
<tr>
<td>4. Detail Hardware Design</td>
<td></td>
</tr>
<tr>
<td>4.1 General Configuration</td>
<td>141</td>
</tr>
<tr>
<td>4.2 Serial Data Bus</td>
<td>143</td>
</tr>
<tr>
<td>4.3 Line Termination Unit</td>
<td>143</td>
</tr>
<tr>
<td>4.4 Design Features</td>
<td>145</td>
</tr>
<tr>
<td>5. Implementation of Selected Function (Output Message)</td>
<td></td>
</tr>
<tr>
<td>5.1 Hardware Implementation</td>
<td>146</td>
</tr>
<tr>
<td>5.2 LTU Processor PCB</td>
<td>146</td>
</tr>
<tr>
<td>5.3 LTU Channel PCB</td>
<td>146</td>
</tr>
<tr>
<td>5.4 Software Implementation</td>
<td>150</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1-1</td>
<td>System Hierarchy Chart</td>
<td>4</td>
</tr>
<tr>
<td>2.2-1/4</td>
<td>Run Switch Hierarchy Charts</td>
<td>16</td>
</tr>
<tr>
<td>2.3-1</td>
<td>Message Schema</td>
<td>21</td>
</tr>
<tr>
<td>2.4-1</td>
<td>Receive Valid Message</td>
<td>23</td>
</tr>
<tr>
<td>2.4-2</td>
<td>Receive Valid Message Interconnectivity</td>
<td>24</td>
</tr>
<tr>
<td>2.4-3</td>
<td>Build Valid Asynchronous Segment</td>
<td>25</td>
</tr>
<tr>
<td>2.4-4/6</td>
<td>Build Valid Asynchronous Segment Interconnectivity</td>
<td>26</td>
</tr>
<tr>
<td>2.4-7</td>
<td>Build Synchronous Segment</td>
<td>29</td>
</tr>
<tr>
<td>2.4-8</td>
<td>Build Synchronous Segment Interconnectivity</td>
<td>30</td>
</tr>
<tr>
<td>2.4-9</td>
<td>Process MCB</td>
<td>31</td>
</tr>
<tr>
<td>2.4-10</td>
<td>Process MCB Interconnectivity</td>
<td>32</td>
</tr>
<tr>
<td>2.4-11</td>
<td>Data Chart for Process MCB</td>
<td>33</td>
</tr>
<tr>
<td>2.4-12</td>
<td>Validate Message</td>
<td>34</td>
</tr>
<tr>
<td>2.4-13</td>
<td>Validate JANAP Message</td>
<td>35</td>
</tr>
<tr>
<td>2.4-14</td>
<td>Validate ACP Message</td>
<td>36</td>
</tr>
<tr>
<td>2.4-15</td>
<td>Validate HPJ Message</td>
<td>37</td>
</tr>
<tr>
<td>2.4-16</td>
<td>Validate HPA Message</td>
<td>38</td>
</tr>
<tr>
<td>2.4-17</td>
<td>Validate Message Interconnectivity</td>
<td>39</td>
</tr>
<tr>
<td>2.4-18</td>
<td>Abbreviation Chart for Validate Message</td>
<td>40</td>
</tr>
<tr>
<td>2.4-19</td>
<td>Data Chart for Validate Message</td>
<td>41</td>
</tr>
<tr>
<td>2.4-20</td>
<td>Manage Message Queues</td>
<td>43</td>
</tr>
<tr>
<td>2.4-21</td>
<td>Manage Message Queues Interconnectivity</td>
<td>44</td>
</tr>
<tr>
<td>2.4-22</td>
<td>Manage Translated Message Queues</td>
<td>45</td>
</tr>
<tr>
<td>2.4-23</td>
<td>Manage Translated Message Queues Interconnectivity</td>
<td>46</td>
</tr>
<tr>
<td>2.4-24</td>
<td>Process Message</td>
<td>48</td>
</tr>
<tr>
<td>2.4-25</td>
<td>Process Message Interconnectivity</td>
<td>49</td>
</tr>
<tr>
<td>2.4-26</td>
<td>Route Message</td>
<td>50</td>
</tr>
<tr>
<td>2.4-27</td>
<td>Route Message Interconnectivity</td>
<td>51</td>
</tr>
<tr>
<td>2.4-28</td>
<td>Segregate Routing Line Interconnectivity</td>
<td>52</td>
</tr>
<tr>
<td>2.4-29</td>
<td>Complete Version Header Interconnectivity</td>
<td>53</td>
</tr>
<tr>
<td>2.4-30</td>
<td>Translate Message</td>
<td>54</td>
</tr>
<tr>
<td>2.4-31</td>
<td>Translate Message Interconnectivity</td>
<td>55</td>
</tr>
<tr>
<td>2.4-32</td>
<td>Output Message</td>
<td>57</td>
</tr>
<tr>
<td>2.4-33</td>
<td>Output Message Interconnectivity</td>
<td>58</td>
</tr>
<tr>
<td>2.4-34</td>
<td>Send Mode I Interconnectivity</td>
<td>59</td>
</tr>
<tr>
<td>2.4-35</td>
<td>Send Mode V and Send Mode II &amp; IV Interconnectivity</td>
<td>60</td>
</tr>
<tr>
<td>2.4-36</td>
<td>Header Valid</td>
<td>61</td>
</tr>
<tr>
<td>2.4-37</td>
<td>Header Valid Interconnectivity</td>
<td>62</td>
</tr>
<tr>
<td>2.4-38</td>
<td>Decouple Log</td>
<td>63</td>
</tr>
<tr>
<td>2.4-39</td>
<td>Decouple Log Interconnectivity</td>
<td>64</td>
</tr>
<tr>
<td>2.4-40</td>
<td>Message Storage Allocation/Deallocation</td>
<td>66</td>
</tr>
<tr>
<td>2.4-41</td>
<td>Message Storage Allocation/Deallocation Interconnectivity</td>
<td>67</td>
</tr>
<tr>
<td>Figure No.</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.5-1</td>
<td>Journal Object</td>
<td>69</td>
</tr>
<tr>
<td>2.5-2</td>
<td>Reference Storage Object</td>
<td>70</td>
</tr>
<tr>
<td>2.5-3</td>
<td>Physical Port Object</td>
<td>71</td>
</tr>
<tr>
<td>2.5-4</td>
<td>Message Object</td>
<td>72</td>
</tr>
<tr>
<td>2.5-5</td>
<td>Logical Port Table and Queue Objects</td>
<td>73</td>
</tr>
<tr>
<td>2.5-6</td>
<td>Route Table Object</td>
<td>74</td>
</tr>
<tr>
<td>2.5-7</td>
<td>Example of Message Input Utilizing Object Oriented Structure</td>
<td>75</td>
</tr>
<tr>
<td>2.5-8</td>
<td>Example of Process Message Utilizing Object Oriented Structure</td>
<td>76</td>
</tr>
<tr>
<td>3.2-1</td>
<td>Package Specification Dependencies</td>
<td>116</td>
</tr>
<tr>
<td>3.5-1</td>
<td>Logical Result of Hardware/Software Partitioning</td>
<td>138</td>
</tr>
<tr>
<td>3.5-2</td>
<td>Output Message Main Processor Software Diagram</td>
<td>139</td>
</tr>
<tr>
<td>3.5-3</td>
<td>Output Message Remote Processor Software Diagram</td>
<td>140</td>
</tr>
<tr>
<td>4.1-1</td>
<td>Message Switch Hardware Block Diagram</td>
<td>142</td>
</tr>
<tr>
<td>4.3-1</td>
<td>LTU Block Diagram</td>
<td>144</td>
</tr>
<tr>
<td>5.2-1</td>
<td>LTU Processor PCB Block Diagram</td>
<td>147</td>
</tr>
<tr>
<td>5.3-1</td>
<td>LTU Channel PCB Block Diagram</td>
<td>149</td>
</tr>
</tbody>
</table>
1. **Introduction**

This document is a description of the final Army TYC-39 Message Switch design using the "Ada Integrated Methodology" developed by General Dynamics Data Systems Division. Although the rationale for the design decisions is presented here, the issues that were debated in order to arrive at this design are discussed in the project "Final Report". The basis for this design is contained in another document, "Ada Equivalent System Requirements Specification", which explains the functional operation of a message switch and compatible interface equipment. All three of the documents mentioned are separately submitted as a part of this project.

The design description is organized into four major sections, which are: 1) System Design, 2) Detail Design, 3) Detail Hardware Design and 4) Implementation. This project was accomplished without any preconceived ideas about hardware/software partitioning. Therefore, the entire system design was completed before the partitioning was done. Then, because of the limited scope of the project, the "message output" section was chosen for detailed design and implementation.

The "System Design" section contains charts and diagrams from which the structure of the message switch was derived. The charts begin with the system hierarchy, which shows the major packages required for message processing. Run switch is presented next because of the order in which the system must be started and initialized. Once up and running with a loaded database, the message processing can begin. Most of the design effort was concentrated in this area because of the critical real time nature of this process. The internal message storage structure is presented in the message schema and the detailed logic of message processing is shown in the functional decomposition diagrams. The data structure diagrams are included because much of the resulting design is based on the ideas derived from these early design sessions (see final report).

The "Detail Design" section provides more specific details of system operation. The Ada Unit Specifications were developed for the structure of the entire switch before considering hardware/software partitioning, as were the data structures, traceability, and package specification dependency chart. Then the analysis that was given to system partitioning is presented.

The "Detail Hardware Design" provides some information required to implement the system as partitioned so that the messages can be processed within the constraints of the non-functional requirements. Since "message output" was selected for implementation, only the input/output section of the message switch hardware detail is shown. The input processing at the line termination unit is so tightly coupled
to message output that both processes had to be done together. Because a multiprocessing environment was chosen, the detailed hardware section describes the added complication of interprocessor operations.

Finally, the "Implementation" section describes the lowest level block diagram of the Line Termination Unit and an explanation of the intended method to dynamically allocate the tasks required for message output. Due to its size, the Ada source code is contained in another volume entitled, "Source Code Document".
2. **System Design**

This section contains system charts and diagrams representing design decisions based on the "Ada Equivalent System Requirements Specification for the AN-TYC-39 Store and Forward Message Switch". The design proceeded according to a methodology tailored for this project and described in the document "Ada Integrated Methodology".

2.1 **System Hierarchy**

The chart shown in Figure 2.1-1 shows the top level operation of the message switch. Run Switch and Response to Operator are functions necessary for supervision of the message processing. They are described in the Run Switch Section, but not to the detail required by an implementation. The scope of the project was somewhat limited, thus this aspect being less time critical, a minimal effort was applied to this task.

The other parallelograms presented represent Ada tasks with the name before the period being the package name containing the task. The name after the period is the task name if the figure is a parallelogram or a procedure name if the figure is a rectangle. The internal memory structure required to store the large volumes of data handled by the message switch is shown by the Manage Intransit and Manage Overflow diagram contained in section 2.4.5. These tasks and the History Ops package are utilized as necessary out of the other operational tasks.
* MANAGE_TRANSLATED_QUEUES.
  MOVE_INTERCEPT

Figure 2.1-1 HIERARCHY CHART
2.2 Run Switch

2.2.1 Overview

In any large system design there are multitudes of details which must be considered and coordinated in order to produce the most efficient system. Since the primary purpose of this contract was to develop methods of using Ada and to investigate its applicability to a communication system, some areas of the system design were considered in less detail than others. These areas were chosen for two reasons: 1) they were standard problems which were not particularly related to Ada or to communications systems, and 2) to reduce the complexity of the job and allow indepth design and analysis of unique Ada applications. Although Run Switch, including the Operator Interface, was selected as an area less deserving of detailed examination, in the interest of thoroughness and integration with the rest of the system design some work was done to describe the basic structure and define the required operations. The following Description and Theory of Operation is an overview of Run Switch and is not intended to be a full treatment of the problem.
2.2.2 Operation

Run Switch is the program unit which initiates, monitors and terminates switch operation. In order to accomplish these functions, Run Switch contains various sub-program units in the relationships illustrated in sheets 1-4 of the Run Switch diagram and described below. The software design does not make use of Run Switch as an operating system or an executive. Ada provides the operating system and an executive is unnecessary. Once initiated, the switch software will be either self driven or externally driven rather than depending on an executive. Run Switch will simply monitor the operation.

A. Initiation - Initiation consists of the capability to begin switch operation under one of three possible start-up conditions: 1) Cold Start, 2) Restart, after an ordered (Normal) shut-down, or 3) Recovery, after an emergency or unscheduled shut-down. Although all three operations achieve the same results (an operational switch with valid tables and status and an open log), the methods by which these results are achieved are different because of different initial conditions.

1. Start Switch - This is the "cold-start" sequence and assumes a lack of reliance on previous operating information. This situation might be encountered after a move to a new location or after switch reconfiguration. Provisions are made for operator entry of table and status information as well as use of prestored information. Since previous messages and journal information are either not available or not used, these items are initially set to the null state, in preparation for actual switch operation.

2. Restart Switch - This sequence assumes resumption of switch operation after an orderly shut-down. In this case, the operating tables, status information, and the final balanced journal would have been saved on non-volatile media. Restart would then retrieve this information and request any changes from the operator. The final journal would be the basis for the new journal and saved messages would be retrieved from the long history file and checked. The retrieved messages would be written into Intransit Storage.

3. Recover Switch - This sequence is slightly different from Restart in that the previous shut-
down was either unscheduled or an emergency and the normal, ordered shut-down process was not followed. In this case, all of the required information may not be available in the desired format, and some preliminary processing may be required. For example, the journal will be available from non-volatile media but in an emergency shut-down time may not have been available to balance it. Therefore the balancing must be done to maintain accountability, before normal switch operation resumes. Tables and status information do not present the same problems because in normal operation this information is periodically saved, as well as updated when changes are made. Of course, the operator will be able to change the tables or status as conditions warrant, after they are reloaded.

B. Operation - During normal switch operation, Run Switch will be concerned with monitoring the operation of the switch and maintaining the operator interface. This also includes monitoring of current status and potential hardware and software error conditions, as well as performing background diagnostics and periodically balancing the log.

C. Termination - Termination of switch operation can take place in either of two forms: 1) Normal (ordered) Shut-Down or 2) Emergency (unscheduled) Shut-Down. The normal sequence is more desirable, however, it cannot be assured under all circumstances.

1. Stop Switch (Normal) - This sequence assumes that sufficient time is available to complete an orderly shut-down. Since this time is a minimum only, the DRY-UP Command is a sub-set of, or a preliminary to a Normal Stop. Depending on the time available, message reception and transmission could be terminated as soon as the current message on a channel is finished (Fast Dry-Up), or reception could be terminated with receipt of the current message and transmission on a channel terminated after all messages available for that channel have been transmitted. In any case, after these conditions are met, generation and saving on non-volatile media of final journal and other operating information would be accomplished. At this point the switch is no longer operational, although the operator interface portion of run switch will continue to run, allowing off line diagnostics, data base changes, etc. From this state, power could be cut off with no detrimental effects.
2. Stop Switch (Emergency) - This sequence causes immediate termination of message I/O, and if time is available, updates the :g. Since the assumption of this sequence is that the required time for a normal shut-down is not available, the sequence attempts to accomplish only essential items. It is anticipated that the sequence may be entered by operator command or upon automatic detection of a catastrophic fault (such as power failure).
2.2.3 Functions

Reference Figure 2.2-1

A. Load Program - controls the loading of the nonresident portions of the switch software from nonvolatile storage.

B. Get RI Table - determines if the RI Table should be retrieved from nonvolatile storage or requested from the operator, and initiates the following actions as appropriate.

1. RI_OPS.INIT_RI_TABLE - prepares the RI Table storage space for the entry of new information.

2. RI_OPS.LOAD_RI_TABLE - retrieves the prestored or saved RI Table data from nonvolatile storage and loads that information into the RI Table storage space.

3. Prompt Operator for Changes - initiates an interactive mode for operator verification of loaded RI Table information and entry or modification of information as required.

4. RI_OPS.SAVE_RI_TABLE - duplicates the Current RI Table in nonvolatile storage.

C. Get Line Table - determines if the Line Table should be retrieved from nonvolatile storage or requested from the operator, and initiates the following actions as required.

1. LINE_TBL_OPS.INIT - prepares the Line Table storage space for the entry of new information.

2. LINE_TBL_OPS.LOAD - retrieves the prestored or saved Line Table data from nonvolatile storage and loads that information into the Line Table storage space.

3. Prompt Operator for Changes - initiates an interactive mode for operator verification of loaded Line Table information and entry or modification of information as required.

4. LINE_TBL_OPS.SAVE - duplicates the current Line Table in nonvolatile storage.
Figure 2.2-1 (continued)

D. **Get Status** - determines if the Status information should be retrieved from nonvolatile storage or requested from the operator, and initiates the following actions as required.
   1. **Initialize Status** - prepares the Status storage space for the entry of new information.
   2. **Load Status** - retrieves the prestored or saved Status from nonvolatile storage and loads that information into the Status storage space.
   3. **Prompt Operator for Changes** - initiates an interactive mode for operator verification of loaded Status information and entry or modification of information as required.
   4. **Save Status** - duplicates the current Status in nonvolatile storage.

E. **AUDIT.AUDIT_INTRANSIT** - used during recovery to read the previous log and generate an audit which will become the basis for the new Running Audit. Also produces a list of messages to be recovered from Reference Storage and reintroduced.

F. **HISTORY_OPS.REENTER_MESSAGES** - accepts a list of messages to be recovered, attempts to recover those messages and reintroduce them into the system. If necessary, generates a list of messages which could not be recovered.

G. **HISTORY_OPS.INIT_HISTORY** - begins a new History with no initial entries. Used when there is no previous Journal.

H. **Activate Channels** - controls and coordinates the activation of the individual communication channels.
   1. **Initializes Channels** - prepares the channels for message transmission and reception by specifying the necessary operating controls and parameters for each individual channel.
   2. **Generate Service Messages** - as appropriate, Service Messages will be generated for transmission to the channels' distant ends to indicate that a link has been activated.
   3. **Generate Control Characters** - as appropriate, Control Characters will be generated for transmission to the Channel's distant ends to indicate that a link has been activated.
4. Save Status - the operational status and operating parameters of all channels will be saved in nonvolatile storage.

Figure 2.2-2

I. Monitor Hardware Errors - This section provides for software acceptance of monitoring outputs. It also will initiate alarms, corrective action, or operator notification as appropriate. Automatic hardware error checking and monitoring is assumed.

J. Monitor Software Errors - provide for the detection, monitoring, and correction of software errors as well as tabulating and correcting (when possible) exceptions raised by the software, perform independent error detection, and initiate the appropriate action for the error condition.

K. Monitor Status - changes in equipment status will be automatically monitored. This may cause automatic switchovers for backed-up equipment as well as alarms and notices to the operator. Operator entered status changes will be saved for future reference.

L. Perform Background Diagnostics - this task will be performed continuously during switch operation to test both operational capabilities and error detection capabilities.

   1. Save - Periodically saves in nonvolatile storage the results of monitoring hardware, software and status and of Background Diagnostics.

   2. Print - initiates the hard copy output of detected error conditions and the periodic printing of the status of the monitored conditions.

M. HISTORY_OPS.NEW_DAY - a standard package that closes the old day's history and saves the current audit, begins a new day's history, then audits the old day's history and compares the generated audit to the saved running audit. Also compiles the statistics for the old day, initiates printing the statistics and clears the file for the new day.
Figure 2.2-3

II. Interface to Operator - provides coordination between the various subfunctions of the operator interface. These subfunctions interact with the operator as well as initiating the requested operation(s). The interactive portions will be of the menu driven, selection and response type with automatic verification before activation.

1. Command Interface - coordinates the activities required for command acceptance before routing the command to the appropriate subfunction for execution.

   a. Verify Command - insures that an operator entered command is correct and valid for the context in which it is being saved.

   b. Interpret Command - converts the operator entered command from a series of key strokes to a selection code with appropriate parameters for routing to and use by the other subfunctions.

2. Control RI Table - depending on the selection code and parameters which initiate this subfunction, the appropriate entry point to the RI_OPS Package is selected.

   a. Read - provisions to allow the operator to examine or verify the entries in the RI Table.

   b. Modify - provisions to allow the changing, addition or deletion of any or all of the RI Table entries.

   c. Save - initiates the storage of the RI Table in nonvolatile storage for future reference.

3. Control Line Table - depending on the selection code and parameters which initiate this subfunction, the appropriate entry point to the LINE-OPS Package is selected.

   a. Read - provisions to allow the operator to examine or verify the entries in the Line Table.

   b. Modify - provisions to allow the changing, addition or deletion of any or all of the Line Table entries.

   c. Save - initiates the storage of the Line Table in nonvolatile storage for future reference.
4. Trace Message - provides the capability to search the journal and retrieve all log entries pertaining to a particular message for presentation to the operator.
   a. Search - initiates a sequential journal search to locate the next log entry for a particular message.
   b. Read - reads the next log entry for presentation to the operator.

5. Control Intransit Thresholds - allows the operator to check or change the threshold levels for Intransit memory. The value of the thresholds (in percent) determines when Overflow Storage will be activated and deactivated.
   a. Read - allows the operator to examine the current thresholds.
   b. Set - allows the operator to set or modify the current thresholds.

6. Access Message - provides facilities for monitoring, correcting, originating and terminating messages.
   a. Read - allows the operator to examine a message which is already in the system.
   b. Modify - allows the operator to change or correct a message.
   c. Generate - allows the operator to manually type in a message in one of several predefined formats.
   d. Insert - places or replaces a message in the system.
   e. Remove - takes a message out of the system.
   f. Save - causes a message to be saved in volatile storage.

7. Print - is an output driver which allows certain types of information and operator requested items to be printed.

8. Maintain Switch - contains the off-line diagnostics, and fault isolation functions as well as on-line maintenance functions and the Supervisor/Maintainer functions.
9. Notify Operator - an output interface to the operator for special purpose, time critical or warning information.

10. Control Status - provides capabilities for the operator to examine and change the various items of status and condition information.
   a. Read - allows the operator to examine the current status.
   b. Modify - allows the operator to change certain status items to reflect the current conditions.
   c. Save - causes all of the current status information to be saved in nonvolatile storage.

Figure 2.2-4

O. Stop Receiving Messages - the first step in any ordered shut-down or "DRY-UP" sequence. In general, when activated will notify each receiving function to cease accepting new messages. Qualifiers may be applied to initiate a slower or a line specific dry-up.

P. Finish Transmitting Messages - this function may be initiated to cease transmission on any or all lines either when transmission of the current message is complete or when all of the queued messages have been transmitted.

Q. Deactivate Channels - causes the transmission of control characters or service messages, as appropriate, to indicate the impending channel deactivation. After this transmission channels are removed from active service.

R. Save All Messages Not Transmitted - messages or message versions which have not been transmitted are saved in nonvolatile storage, for later recovery and transmission.

S. HISTORY OPS.CLOSE HISTORY - A standard package that closes the day's history and saves the current audit. The day's history is reaudited and the generated audit is compared to the saved running audit. Statistics are compiled for the day.

T. Notify Operator - at the various stages of a normal stop the operator will be advised of the current progress and status. The operator will also be advised after the initial operations of an emergency stop are completed.
U. **Terminate Input/Output Processing** - the receipt of incoming messages and the transmission of outgoing messages will be terminated immediately when an emergency condition is detected.

V. **Finish Writing Log Entries** - log entries which are in progress or pending are completed so that the most complete journal will be available for recovery. This reduces the probability of transmission of duplicate messages.

W. **Save Running Audit** - the current running audit is saved in nonvolatile storage to be used as a cross-check during recovery.
Figure 2.2-1 Run Switch
Figure 2.2-2 Run Switch
Figure 2.2-4  Run Switch
2.3 Internal Message Structure

The message schema shown in Figure 2.3-1 was chosen to take advantage of two facets of the messages handled by the switch.

First, the messages are received in pieces. Those messages which are sent in synchronous mode are received in blocks of 80 characters. Asynchronous messages may be interrupted by sequences of control characters. In addition to these problems, the first part of the message must be validated during the reception of the remainder, since the message must be acknowledged very shortly after the end of message is received. Therefore, the message is stored in "chunks" of 80 characters, which the designers chose to call segments. The segments are placed in a doubly linked list, using next segment and prior segment pointers. Since the segments are not stored together, information must be added to ensure that they remain properly linked together. The only item that is required by the performance specification is the security classification of the message. For further checking to be sure that no segments are lost or improperly linked, the following information was added: the serial number of the message, the part of the message involved (MCB, header, body, or trailer), and a number which starts with one for each part, and increases by one for each segment. The segments are also time-stamped for logging purposes.

Most messages are split into more than one copy during processing. In order to save storage, the designers decided to share text between versions of messages at the part level. To do so, a structure called the part header was created. The part headers are linked to the first and last segments of their respective parts. The part header contains the name of the part involved, a count of the number of messages which are using this part, and a count of the segments which comprise the part. This last item can be compared with the segment number of the last segment in the part.

The only remaining part of the message structure is the version header. The information contained in the version header includes the serial number of the message plus an extension to differentiate between version, the security classification of the message, the precedence of the message, its time of receipt, its category (normal or service) the output line to which the message is routed, and the character set (ASCII or ITA) of the message. The version headers are linked to the part headers through an array of access variables indexed by the part name.
2.4 Functional Decomposition and Interconnectivity

2.4.1 Message Input

The Message Input function is illustrated in Figures 2.4-1 through 2.4-19. The dotted parallelograms indicate that the referenced task or procedure is elaborated on a separate page. Other symbology and guidelines to interpreting the charts is discussed on page 64 of the Ada Integrated Methodology.
Figure 2.4-1 Receive Valid Message
Figure 2.4-2 Receive Valid Message Interconnectivity
Figure 2.4-3 Build Valid Asynchronous Segment
Module: Build Val. Asynch. Seg
Level: #2, Format, Asynch. Segment
Date: March 31, 1982

Figure 2.4-5 Build Valid Asynchronous Segment Interconnectivity
Module: Build Valid Asynch. Seg

Level: "2, Validate First Segment

Date: March 30, 1982

Figure 2.4-6 Build Valid Asynchronous Segment Interconnectivity
<table>
<thead>
<tr>
<th>LABEL</th>
<th>INTERFACE COMPONENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Hdr_Seg_Ptrs</td>
</tr>
<tr>
<td>B</td>
<td>MCB_Seg_Ptr</td>
</tr>
<tr>
<td>C</td>
<td>MCB_Seg_Ptr,Hdr_Seg_Ptrs</td>
</tr>
<tr>
<td>D</td>
<td>Hold</td>
</tr>
<tr>
<td>E</td>
<td>Collective_Present</td>
</tr>
<tr>
<td>F</td>
<td>Operator_Message</td>
</tr>
<tr>
<td>G</td>
<td>Log_Entry</td>
</tr>
<tr>
<td>H</td>
<td>Seg_Ptr</td>
</tr>
<tr>
<td>I</td>
<td>Seg_Ptr,Position,Count,Chars</td>
</tr>
<tr>
<td>J</td>
<td>Seg_Ptr,Count,Position</td>
</tr>
<tr>
<td>K</td>
<td>Chars</td>
</tr>
<tr>
<td>L</td>
<td>Line #</td>
</tr>
<tr>
<td>M</td>
<td>Channel_Mode</td>
</tr>
<tr>
<td>N</td>
<td>Channel_Des</td>
</tr>
</tbody>
</table>

*Figure 2.4-11 Chart for Process MCB*
Figure 2.4-12 Validate Message

- \(\star = \text{STATE, MSGID, SEG}\)
- \(* = \text{STATE}\)
Figure 2.4-13 Validate JANAP Message
Figure 2.4-15  Validate HPJ Message

\[\text{VALIDATE HPJ MESSAGE}\]
Figure 2.4-17 Validate Message Interconnectivity
### 'VALIDATE MESSAGE'

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val. Msg.</td>
<td>Validate Message</td>
</tr>
<tr>
<td>VJ</td>
<td>Validate JANAP Msg</td>
</tr>
<tr>
<td>VA</td>
<td>Validate ACP Msg</td>
</tr>
<tr>
<td>VHPJ</td>
<td>Validate HPJ Msg</td>
</tr>
<tr>
<td>VHHA</td>
<td>Validate HPA Msg</td>
</tr>
<tr>
<td>CJH</td>
<td>Check JANAP Header</td>
</tr>
<tr>
<td>CRC</td>
<td>Check Record Count</td>
</tr>
<tr>
<td>CRI's</td>
<td>Check RI's</td>
</tr>
<tr>
<td>CRI</td>
<td>RI_Ops. Check RI</td>
</tr>
<tr>
<td>CEOM</td>
<td>Check EOM</td>
</tr>
<tr>
<td>CEOT</td>
<td>Check EOT</td>
</tr>
<tr>
<td>CAH</td>
<td>Check ACP Header</td>
</tr>
<tr>
<td>CCAH</td>
<td>Complete Checking ACP Header</td>
</tr>
<tr>
<td>CLMF</td>
<td>Check LMF</td>
</tr>
<tr>
<td>CHSH</td>
<td>Check HPJ Header</td>
</tr>
<tr>
<td>CHAH</td>
<td>Check HPA Header</td>
</tr>
<tr>
<td>CCHAH</td>
<td>Complete Checking HPA Header</td>
</tr>
<tr>
<td>RP</td>
<td>Segment_Ops.Read_Part</td>
</tr>
<tr>
<td>RCC</td>
<td>Segment_Ops.Read_Char_Count</td>
</tr>
<tr>
<td>RC</td>
<td>Segment_Ops.Read_Chars</td>
</tr>
<tr>
<td>RSP</td>
<td>Line_Table_Ops.Read_Sec_Prosign</td>
</tr>
<tr>
<td>RCM</td>
<td>Line_Table_Ops.Read_Channel_Mode</td>
</tr>
<tr>
<td>GSM</td>
<td>Generate Service Message</td>
</tr>
<tr>
<td>GCC</td>
<td>Generate CC</td>
</tr>
<tr>
<td>WS</td>
<td>Segment_Ops.Write_Specific</td>
</tr>
</tbody>
</table>

Figure 2.4-18 Chart for Validate Message
<table>
<thead>
<tr>
<th>LABEL</th>
<th>INTERFACE COMPONENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>State,Msgid,Seg</td>
</tr>
<tr>
<td>B</td>
<td>State</td>
</tr>
<tr>
<td>C</td>
<td>Msgid,Seg,Char_count,Where,State</td>
</tr>
<tr>
<td>D</td>
<td>Where,State</td>
</tr>
<tr>
<td>E</td>
<td>Seg</td>
</tr>
<tr>
<td>F</td>
<td>Part</td>
</tr>
<tr>
<td>G</td>
<td>Char_count</td>
</tr>
<tr>
<td>H</td>
<td>RI</td>
</tr>
<tr>
<td>I</td>
<td>Check</td>
</tr>
<tr>
<td>J</td>
<td>Seg,Where,Count</td>
</tr>
<tr>
<td>K</td>
<td>Chars</td>
</tr>
<tr>
<td>L</td>
<td>Line #</td>
</tr>
<tr>
<td>M</td>
<td>Security Prosign</td>
</tr>
<tr>
<td>N</td>
<td>Channel_Mode</td>
</tr>
<tr>
<td>O</td>
<td>Msgid,Reason_for_Svc_Msg</td>
</tr>
<tr>
<td>P</td>
<td>Msgid,Reason_for_CC</td>
</tr>
<tr>
<td>Q</td>
<td>Seg,Where,Chars</td>
</tr>
</tbody>
</table>

Figure 2.4-19 Chart for Receive Valid Message
Build Message from Segments
Validate Message
2.4.2 Message Queueing

The Message Queueing function is illustrated in Figures 2.4-20 through 2.4-23.
Figure 2.4-20 Manage Message Queues
Figure 2.4-22 Manage Translated Message Queues
Figure 2.4-23 Manage Translated Message Queues Interconnectivity
2.4.3 Message Routing

The Message Routing function is illustrated in Figures 2.4-24 through 2.4-31.
Module: Process Message
Level: Top
Date: April 1, 1982

Figure 2.4-25 Process Message Interconnectivity
Figure 2.4-26 Route Message
Figure 2.4-28 Segregate Routing Line Interconnectivity

Module: Segregate Routing Line
Date: April 2, 1982
Figure 2.4-30 Translate Message
Figure 2.4-31 Translate Message Interconnectivity
2.4.4 Output Message

The Output Message function is illustrated in Figures 2.4-32 through 2.4-39.
Figure 2.4-33 Output Message Interconnectivity

<table>
<thead>
<tr>
<th>LABEL</th>
<th>INTERFACE COMPONENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>MSGID</td>
</tr>
<tr>
<td>B</td>
<td>VALID_FLAG</td>
</tr>
<tr>
<td>C</td>
<td>MSGID,PREEMPT</td>
</tr>
<tr>
<td>D</td>
<td>LOG_ENTRY_REQUEST,SUCCESS</td>
</tr>
</tbody>
</table>
Figure 2.4-37  Header Valid Interconnectivity
2.4.5 *Manage Intransit, Manage Overflow*

The Manage Intransit, Manage Overflow function is illustrated in Figures 2.4-40 through 2.4-41.
Figure 2.4-40 MESSAGE STORAGE ALLOCATION/DEALLOCATION
Module: Manage Intransit, Manage Overflow

Date: Apr. 2, 1982

Figure 2.4-41 Manage Intransit, Manage Overflow Interconnectivity
2.5 Data Structure Diagrams
Figure 2.5-1 Journal Object
Figure 2.5-2 Reference Storage Object
Figure 2.5-3  Physical Port Object
Figure 2.5-4 Message Object
Figure 2.5-5 Logical Port Table and Queue Objects
Figure 2.5-6 Route Table Object
HIGH LEVEL PORT TASK

LOOP

ASSEMBLE.MESSAGEx

WRITE.INCOMINGQ (MESSAGE)

-- Construct Next Message

-- Place in Q

ENDLOOP

Figure 2.5-7 Example of Message Input Utilizing Object Oriented Structure
PROCESS MESSAGE

LOOP

READ.INCOMINGQ

LOOK_UP,ROUTE_TABLE (MESSAGE)

FOR EACH ROUTE_INDICATOR

CREATE,MESSAGE,VERSION (LMF)

WRITE.OUTGOINGQ (MESSAGE,VERSION) -- SEND THE MESSAGE

ENDFOR

ENDLOOP

-- END LOOP

Figure 2.5-8 Example of Process Message Utilizing Object Oriented Structure
3. Detailed Design

3.1 Ada Unit Specifications

3.1.1 Message Input
with SEGMENT_OPS, LINE_TBL_OPS, MESSAGE_OPS; use SEGMENT_OPS, MESSAGE_OPS;

package INPUT_PORT is

  task type RECEIVE VALID MESSAGE is
    entry INITIATE(PHYSICAL_PORT:LINE_TBL_OPS.PHYSICAL_LINE);
    entry TERM(PHYSICAL_PORT:LINE_TBL_OPS.PHYSICAL_LINE);
  end RECEIVE

  type SEG_STATUS is (GOOD,CAN,CANTRAN,EXPIRED_PAUSE);
  type RCV_STATUS is (VALID,HOLD,STOREFAILED,BAD_MCB,BAD_RIS,
                      BAD_MODE,BAD_SECURITY);

  task type BUILD_MESSAGE_FROM_SEGMENTS is
    entry BUILD_SEG(NEW_SEG:SEG_PTR;SEG_RESULT:SEG_STATUS);
    entry CHECK_MCB(MCB_SEG:out SEG_PTR);
    entry BUILD_MCB(HDR_SEG:out SEG_PTR);
    entry COMPLETE_MCB(MCB_SEG:SEG_PTR;TRL_SEG:out SEG_PTR);
    entry VALID_MCB(MCB_RESULT:RCV_STATUS);
    entry STORE_REF(STORE_SEG:out SEG_PTR);
    entry STORE_FAIL(FAIL_SEG:SEG_PTR);
    entry CHECK_SEG(VALID_SEG:out SEG_PTR);
    entry VALID_SEG(SEG_RESULT:RCV_STATUS);
    entry POST_STATUS(MSG_STATUS:out RCV_STATUS);
    entry RECEIVE_MESSAGE(MESSAGE:out MSGID;MSG_STATUS:out
                           RCV_STATUS);
  end BUILD_MESSAGE_FROM_SEGMENTS;

end INPUT_PORT;
SEGMENT_OPS

with GLOBALTYPES, MANAGE_STORAGE; use GLOBALTYPES, MANAGE_STORAGE;

package SEGMENT_OPS is

-- NAME: SEGMENT_OPS
-- PURPOSE: contains the data definitions and operations for the
-- manipulation of segments, which are used to hold the
-- actual text of messages.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

type CHAR_COUNT is range 0..81;
type SEGMENT is private;
type SEG_PTR is access SEGMENT;
type SEGMENT_NUMBER is range 0..550;

procedure RESET_SEC(SEG:SEG_PTR);
-- SETS TEXT BUFFER IN SEGMENT TO EMPTY

procedure WRITE_SPECIFIC(SEG:SEG_PTR; WHERE: CHAR_COUNT; TEXT: STRING);
-- PUTS TEXT INTO SEGMENT BUFFER AT SPECIFIC LOCATION
-- USED BY REWRITE IN MESSAGE_OPS

procedure ADD_TEXT(SEG:SEG_PTR; TEXT: STRING);
-- PUTS TEXT INTO SEGMENT BUFFER

function READ_CHARS(SEG:SEG_PTR; WHERE,COUNT: CHAR_COUNT) return STRING;
-- READS COUNT CHARACTERS STARTING AT WHERE

function READ_CHARACTER_COUNT(SEG:SEG_PTR) return CHAR_COUNT;
-- RETURNS THE NUMBER OF CHARACTERS IN THE BUFFER

procedure SET_TIME(SEG:SEG_PTR; HACK: DATE_TIME);
function READ_TIME(SEG:SEG_PTR) return DATE_TIME;
-- SET AND READ TIME HACK OF SEGMENT

procedure SET_PART(SEG:SEG_PTR; PART: PART_NAME);
function READ_PART(SEG:SEG_PTR) return PART_NAME;
-- SET AND READ THE PART_NAME FIELD OF THE SEGMENT

procedure SET_EASN(SEG:SEG_PTR; SERNO: EXT_SERIAL_NO);
function READ_EASN(SEG:SEG_PTR) return EXT_SERIAL_NO;
-- SET AND READ EXTENDED SERIAL NUMBER

procedure SET_SEGMENT_NUMBER(SEG:SEG_PTR; NUMBER: SEGMENT_NUMBER);
function READ_SEGMENT_NUMBER(SEG : SEG_PTR) return SEGMENT_NUMBER;
-- SET AND READ SEGMENT SEQUENCE FIELD

procedure SET_CLASS(SEG : SEG_PTR; CLASS : SECURITY_CLASSIFICATION);
function READ_CLASS(SEG : SEG_PTR) return SECURITY_CLASSIFICATION;
-- SET AND READ CLASSIFICATION

procedure LINK_SEGMENTS(SEG, PRIOR_SEG : SEG_PTR);
-- LINKS SEG INTO A SEGMENT CHAIN
-- SETS FORWARD POINTER IN PRIOR SEGMENT TO SEG
-- SETS BACKWARD POINTER IN SEG TO PRIOR_SEG
-- SETS FORWARD POINTER IN SEG TO NULL

function NEXT_SEGMENT(SEG : SEG_PTR) return SEG_PTR;
function PRIOR_SEGMENT(SEG : SEG_PTR) return SEG_PTR;
-- READ NEXT AND PRIOR SEGMENT POINTERS

procedure FREE_SEGS(SEG : SEG_PTR);
-- FREES AN ENTIRE CHAIN OF SEGMENTS

function GET is new GET_INTRANSIT(SEGMENT,SEG_PTR);
procedure FREE is new FREE_INTRANSIT(SEGMENT,SEG_PTR);
-- GET AND FREE INTRANSIT STORAGE FOR SEGMENTS
-- SEE MANAGE_STORAGE FOR DETAILS

private

type SEGMENT is
record
  BACK_SEG : SEG_PTR := null;
  CLASS : SECURITY_CLASSIFICATION;
  EASN : EXT_SERIAL_NO;
  FWD_SEG : SEG_PTR := null;
  NUMBER_CHARACTERS : CHAR_COUNT := 0;
  PART : PART_NAME;
  SEG_NO : SEGMENT_NUMBER;
  TEXT : STRING(1..80);
  TIME : DATE_TIME;
end record;

end SEGMENT_OPS;
3.1.2 Message Queueing
with MESSAGE_OPS;
use MESSAGE_OPS;

package MANAGE_QUEUES is
  -- NAME: MANAGE_QUEUES
  -- PURPOSE: Task that manages messages after reception while they are awaiting processing for routing.

  task MANAGE_MESSAGE_QUEUES is
    entry QUEUE_NORMAL ( MESSAGE: MSGID );
    entry QUEUE_SERVICE ( MESSAGE: MSGID );
    entry QUEUE_REINTROD ( MESSAGE: MSGID );
    entry DEQUEUE_OVERFLOW ( MESSAGE: out MSGID );
    entry RETRIEVE_FOR_PROCESS ( PRECEDENCE ) ( MESSAGE: out MSGID );
  end MANAGE_MESSAGE_QUEUES;

end MANAGE_QUEUES;
package MANAGE_TRANSLATED_QUEUES is

-- NAME: MANAGE_TRANSLATED_QUEUES
-- PURPOSE: Tasks that manage messages after translation while they are
-- awaiting line availability.

task type MANAGE_OUTGOING is
  entry INIT ( LOG_ID: LOGICAL_LINE );
  entry SET_PHYS_LINES ( PHYS_LINES: LINE_TBL_OPS.PHYS_PTR );
  entry QUEUE_TRANSLATED ( THIS_LINE: LOGICAL_LINE; MESSAGE: MSGID );
  entry QUEUE_REINTROD ( THIS_LINE: LOGICAL_LINE; MESSAGE: MSGID );
  entry LINE_READY ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
  entry KILL_NOW ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
  entry KILL_LINE ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
  entry KILL_ON_EMPTY ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
  entry INTERCEPT ( THIS_LINE: LOGICAL_LINE; PREC: PRECEDENCE; MEDIUM: LINE_TBL_OPS.MEDIA );
  entry REMOVE_INTERCEPT ( THIS_LINE: LOGICAL_LINE; PREC: PRECEDENCE; MEDIUM: LINE_TBL_OPS.MEDIA );
end MANAGE_OUTGOING;

type MANAGER is access MANAGE_OUTGOING;

task MANAGE_INTERCEPT is
  entry QUEUE_ONE ( MESSAGE: MSGID );
  entry QUEUE_MANY ( LIST: QUEUE.HEAD );
  entry DEQUEUE ( MESSAGE: MSGID );
end MANAGE_INTERCEPT;

task MOVE_INTERCEPT is
  entry START_INTERCEPT_OUT;
  entry STOP_INTERCEPT_OUT;
  entry START_REINTRO_OF_INTERCEPT;
  entry STOP_REINTRO_OF_INTERCEPT;
end MOVE_INTERCEPT;
end MANAGE_TRANSLATED_QUEUES;
MESSAGE_IO

package MESSAGE_IO is

-- NAME: MESSAGE_IO
-- PURPOSE: Input/output subprograms used to read/write messages to/from
-- files (ie., OVERFLOW, INTERCEPT)

procedure READ(FILE_NAME:STRING; MESSAGE:out MSGID);
procedure WRITE(FILE_NAME:STRING; MESSAGE:MSGID);

private

type KIND is (VERSION_HEADER, PART HEADER, SEGMENT);
type BLOCK(PART_KIND:KIND := SEGMENT) is
record
  case PART_KIND is
    when VERSION_HEADER =>
      VERSION:VERSION_HEADER;
    when PART_HEADER =>
      PART_HEAD:PART_HEADER;
    when SEGMENT =>
      SEG:SEGMENT;
  end case;
end record;
end MESSAGE_IO;
with GLOBAL_TYPES, MESSAGE_OPS; use GLOBAL_TYPES, MESSAGE_OPS;

package QUEUE is

-- NAME: QUEUE
-- PURPOSE: Encapsulates the queue mechanisms used to hold the messages
during waiting stages as they pass thru the switch.
-- PROGRAMMER: SLN
-- DATE: 18 MAY 82

-----------------------------
type HEAD is private;
type SET is array (MESSAGE_CAT, PRECEDENCE) of HEAD;
procedure ON_FRONT (QUEUE: in out HEAD; MESSAGE: in MSGID);
-- Place a message properly in a queue, (when it is being
-- reintroduced as from overflow or intercept).
procedure ON_BACK (QUEUE: in out HEAD; MESSAGE: in MSGID);
-- Place a message properly on a queue.
procedure FROM_FRONT (QUEUE: in out HEAD; MESSAGE: out MSGID);
-- Retrieve a message from the front of a queue for normal
-- processing.
procedure FROM_BACK (QUEUE: in out HEAD; MESSAGE: out MSGID);
-- Retrieve a message from the back of a queue for overflow.
function IS_NOT_EMPTY (QUEUE: in HEAD) return BOOLEAN;
-- Test the empty state of a queue
procedure SPLIT (MEDIUM: in MEDIA; QUEUE: in out HEAD;
-- Segregate messages of a particular media from a queue into a
-- separate list.
LIST: out HEAD);
procedure MERGE (MASTER: in out HEAD; SLAVE: in HEAD);
-- Combine two queues into one.
FOR_PROCESSING: SET;
FOR_OUTGOING: array (LOGICAL_LINES) of SET;
FOR_INTERCEPT: SET;

private

type ITEM;
type ITEM_PTR is access ITEM;
type ITEM is
record
  NEXT, PRIOR: ITEM_PTR;
  MESSAGE: MSGID;
  TOR: DATE_TIME;
end record;
type HEAD is
record
  FIRST, LAST: ITEM_PTR := null;
  COUNT: INTEGER := 0;
end record;
end QUEUE;
3.1.3 Message Routing
PROCESS_MSG

with MESSAGE_OPS, RI_OPS, LINE_TBL_OPS; use MESSAGE_OPS, RI_OPS, LINE_TBL_OPS;

package PROCESS_MSG is

-- NAME: PROCESS_MSG
-- PURPOSE: contains the task required to route and translate
-- a message. (translations required depend on the
-- type of exchange required.)
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

task PROCESS_MESSAGE;
end PROCESS_MSG;
3.1.4 Message Output
package PHYSICAL_PORT is

-- NAME: PHYSICAL_PORT
-- PURPOSE: contains the operations and data definitions dealing with
-- operations of the physical output ports, not including
-- operator output.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

type LOG_REQUEST (LOG_TYPE : LOG_ENTRY) is
  record
    BLOCKS : NUM_BLOCKS;
    LINE  : PHYSICAL_LINE;
    MSG   : MSGID;
    SEQ_NUM : CSN;
    case LOG_TYPE is
      when CANCEL_OUT | CANTRAN_OUT =>
        CAN_REASON : CANCEL_REASON;
      when SCRUB =>
        SCR_REASON : SCRUB_REASON;
      when REJECT =>
        REJ_REASON : REJECT_REASON;
      when others =>
        null;
    end case;
  end record;

end LOG_REQUEST;

end PHYSICAL_PORT;
task type SEND_MODE_II_IV is
  entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
end SEND_MODE_II_IV;

task type SEND_MODE_V is
  entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
end SEND_MODE_V;

task type SEND_MODE_I is
  entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
end SEND_MODE_I;

task type CONTROL_CHARACTER is
  entry INIT(P_LINE : PHYSICAL_LINE); -- called by run switch
  entry PUT(CH : CHARACTER); -- called by receiver
  entry GET(CH : out CHARACTER); -- called by send
end CONTROL_CHARACTER;

task type DECOUPLE is
  entry LOG(REQ : LOG_REQUEST); -- called by send
end DECOUPLE;

type DECOUPLE_ACC is access DECOUPLE;

task type THROTTLE_INPUT is
  entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
  entry START_THROTTLE;
  entry STOP_THROTTLE;
end THROTTLE_INPUT;

end PHYSICAL_PORT;
3.1.5 Manage Intransit, Manage Overflow
package MANAGE_INTRANSIT is
  -- NAME: MANAGE_INTRANSIT
  -- PURPOSE: contains procedures and tasks to manage the intransit
  --          storage memory resource.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982

  type MEM_SIZE is INTEGER;
  type THRESHOLD is range 0..100;

  procedure SET_THRESHOLD(LOWER:THRESHOLD := 60;
                          UPPER:THRESHOLD := 70);
  -- SETS LOWER AND UPPER THRESHOLD AS SPECIFIED IN CALL
  -- MIDDLE THRESHOLD IS SET TO THE AVERAGE OF LOWER AND UPPER

  procedure READ_THRESHOLD(LOWER,MIDDLE,UPPER: out THRESHOLD);
  -- READS CURRENT THRESHOLD SETTINGS

  task CALCULATE_THRESHOLD is
    -- CALCULATES THRESHOLD VALUES AND INITIATES OVERFLOW
    -- ACTIONS
    entry GET(BITS:MEM_SIZE);
    -- USED WHEN GETTING STORAGE
    entry PUT(BITS:MEM_SIZE);
    -- USED WHEN RETURNING STORAGE TO INTRANSIT
  end CALCULATE_THRESHOLD;
end MANAGE_INTRANSIT;
package MANAGEOMATIC is

-- NAME: MANAGEOMATIC
-- PURPOSE: contains the procedures to handle the overflow of
-- intransit storage
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

task OVERFLOW_DRIVER is
  entry START_OVERFLOW;
  entry STOP_OVERFLOW;
  entry START_REENTRY;
  entry STOP_REENTRY;
end OVERFLOW_DRIVER;

end MANAGEOMATIC;
3.1.6 Support Routines
package ASN is

-- NAME: ASN
-- PURPOSE: provides the definition of the type SER_NO_TYPE and
-- a task to provide sequential serial numbers.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 14, 1982

private

task MANAGE ASN is
  entry SET(START:SER_NO_TYPE);
  entry GET(NO:out SER_NO_TYPE);
end MANAGE ASN;

private

  type SER_NO_TYPE is range 1..1_000_000;
end ASN;
package AUDIT is

-- NAME: AUDIT
-- PURPOSE: contains data definitions and operations required to audit the history and produce lists of current messages
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

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

type AUDIT_LIST_NODE;
type AUDIT_LIST_PTR is access AUDIT_LIST_NODE;
type AUDIT_RECORD is
  record
    MESSAGE:EASN;
    NEXT: AUDIT_LIST_PTR;
  end record;
CURRENT_AUDIT: AUDIT_RECORD;
-- RUNNING AUDIT FIGURES

function AUDIT_INTRANSIT return AUDIT_RECORD;
-- READS LOG AND PRODUCES A NEW AUDIT WITH AT LIST OF
-- INTRANSIT MESSAGES

function AUDIT_OVERFLOW return OVERFLOW_RECORD;
-- READS LOG & PRODUCES AN AUDIT WITH A LIST OF OVERFLOW MESSAGES

function AUDIT_INTERCEPT return INTERCEPT_RECORD;
-- READS LOG AND PRODUCES AN AUDIT WITH A LIST OF INTERCEPTED
-- MESSAGES

function EQUAL(A,B: AUDIT_RECORD) return BOOLEAN;
-- COMPARES TWO AUDITS FOR SAME NUMBERS AND SAME MESSAGES ON
-- LISTS. THE ORDER OF LISTS ARE NOT SIGNIFICANT.

procedure WRITE_AUDIT(OUTPUT_AUDIT:AUDIT_RECORD);
-- WRITES AUDIT RECORD ON HISTORY TAPES

end AUDIT;
package BYTE_DEF is

---Name: BYTE_DEF
---Purpose: to define bytes, words, and long words
---Programmer: Brian G. Sharp
---Date: 11 June 1982
---Description: Bytes, words, and long words are defined to have 8, 16, and 32 bits, respectively. Also defined are conversions between the different types.

BYTE_SIZE : constant INTEGER := 8;
MAX_BYTE_VALUE: constant INTEGER := 2 ** BYTE_SIZE - 1;
subtype BYTE is INTEGER 0..MAX_BYTE_VALUE;

WORD_SIZE : constant INTEGER := 2 * BYTE_SIZE;
MAX_WORD_VALUE: constant INTEGER := 2 ** WORD_SIZE - 1;
subtype WORD is INTEGER 0..MAX_WORD_VALUE;

LONG_WORD_SIZE : constant INTEGER := 2 * WORD_SIZE;
MAX_LONG_WORD_VALUE: constant INTEGER := 2 ** LONG_WORD_SIZE - 1;
subtype LONG_WORD is INTEGER 0..MAX_LONG_WORD_VALUE;

-- representation specifications for the sizes of the above types
for BYTE'SIZE use BYTE_SIZE;
for WORD'SIZE use WORD_SIZE;
for LONG_WORD'SIZE use LONG_WORD_SIZE;

--merge two bytes into a word
function BYTES_TO_WORD( 
    MST_SIG_BYTE, 
    LST_SIG_BYTE : in BYTE ) return WORD;

--merge two words into a long word
function WORDS_TO_LONG_WORD( 
    MST_SIG_WORD, 
    LST_SIG_WORD : in WORD ) return LONG_WORD;

end BYTE_DEF;

package body BYTE_DEF is

-97-
package CONTROL is

-- NAME: CONTROL
-- PURPOSE: provides a task type for controlling access to
-- a shared resource.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 5, 1982

task type CONTROL_ACCESS is
  entry START_READ;
  entry FINISHED_READ;
  entry START_WRITE;
  entry FINISHED_WRITE;
end CONTROL_ACCESS;

end CONTROL;
HARDWARE_CLOCK

with INTERFACE_TO_8254, BYTE_DEF;
use INTERFACE_TO_8254, BYTE_DEF;

package HARDWARE_CLOCK is

-- Name: HARDWARE_CLOCK
-- Purpose: to implement a hardware, binary counter clock
-- Programmer: Brian G. Sharpe
-- Date: 8 June 1982
-- Description:
-- This package implements a hardware clock using a binary counter.
-- It is implemented when one needs a time DELTA which is less than
-- DURATION*DELTA.
--
-- !!!!!!!!!! NOTE !!!!!!!!!!
-- This is useful for elapsed time only; it is not an absolute
-- (i.e., time of day) clock.
--
-- Since the elapsed timer is implemented in hardware with binary clock,
-- it will "wrap around" from "COUNTER RANGE_MAX" to 0.
--
-- Thus, the user must ensure that the largest possible elapsed time will
-- be less than the time to pass once through the counter.
--
-- package declaration section:

type HARDWARE_CLOCK_TIME is private;

-- Subprogram to start the clock chip running in the proper configuration
procedure INIT_HARDWARE_CLOCK;

function CONVERT_TO_SECONDS( TIME_IN_HCT : HARDWARE_CLOCK_TIME )
  return float;

-- Determines the elapsed time between inputs
function ELAPSED_TIME( START_TIME, FINAL_TIME : in HARDWARE_CLOCK_TIME );

-- Function to read the hardware clock
function HARDWARE_CLOCK_READING return HARDWARE_CLOCK_TIME;

private -- section

CLOCK_SIZE : constant := LONG_WORD_SIZE;   -- # bits in counter clock
COUNTER_RANGE_MAX : constant := MAX-LONG WORD VALUE;
CLOCK_CHANNEL_LOW_WORD is constant CHANNEL_0;
CLOCK_CHANNEL_HIGH_WORD is constant CHANNEL_1;
USE_CHANNEL : SELECT_CHANNEL_ARRAY := ( true, true, false );

-99-
package HISTORY_OPS is
  type ENTRY_TYPE is (LOG, RI_SET, SEGMENT);
  type NUM_RIS is range 0..50;
  type LOG_ENTRY is (SOM_IN, EOM_IN, REJECT, CANCEL_IN, CANCEL_OUT, SOM_OUT, EOM_OUT, SCRUB, OUT_TO_OVFL, IN_FROM_OVFL, OUT_TO_INT, IN_FROM_INT, SVC_GEN, VERSION_OUT, VERSION_IN);
  type REJECT_REASON is (INVALID_HDR, INVALID_RI, INVALID_SECURITY, SECURITY_MISMATCH);
  type CANCEL_REASON is (PREMPTED, REJ_BY_RCVR);
  type SCRUB_REASON is (OPERATOR);
  type NUM_BLOCKS is range 0..50;
  type RI_ARRAY is array (INTEGER range <>) of RI_STRING;
  type HISTORY_ENTRY (ENT:ENTRY_TYPE:=SEGMENT; RI_COUNT:NUM_RIS:=0; LOG_TYPE:LOG_ENTRY:=SOM_IN) is record
    case ENT is
      when LOG =>
        CHANNEL:CHANNEL_DESC;
        SEQ_NUM:CSN;
        HACK:DATE_TIME;
        MODE:CHANNEL_MODE;
        BLOCK_COUNT:NUM_BLOCKS:=0;
        MCB:BOOLEAN:=FALSE;
        HEADER_SEGs:NUM_BLOCKS:=0;
        RIS:BOOLEAN:=FALSE;
      case LOG_TYPE is
        when REJECT =>
          RE_REASON:REJECT_REASON;
        when CANCEL_OUT =>
          CAN_REASON:CANCEL_REASON;
        when SCRUB =>
          SCR_REASON:SCRUB_REASON;
        when Others =>
          null;
        end case;
      end when;
      when RI_SET =>
        RIS:RI_ARRAY (1..RI_COUNT);
      when SEGMENT =>
        SEG_BUF:SEGMENT;
    end case;
  end record;
  type ENTRY_SET is array (INTEGER range <>) of HISTORY_ENTRY;
  type HIST_STATUS is (OK, END_OF_MEDIA, ERROR);
  type MESSAGE_LIST is array (INTEGER range <>) of EASN;

task ENTER_HIST is
  entry WRITE(REC:ENTRY_SET; STATUS: out HIST_STATUS);
end ENTER_HIST;
-- USED TO WRITE A RECORD OR SET OF RECORDS ON THE HISTORY
with ASN; use ASN;

package GLOBAL_TYPES is

-- NAME: GLOBAL_TYPES
-- PURPOSE: This package contains the types global to the entire software project.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

subtype CHANNEL_DES is STRING(1..3);
type CHANNEL_MODE is range 1..5;
type CHAR_SET_TYPE is (ANY, ASC, ITA);
type CSN is range 0..999;

type EXT_SERIAL_NO is record
  ASN : SER_NO_TYPE;
  EXTENSION : INTEGER range 0..100;
end record;

type JULIAN_DATE is range 1..366;

type DATE_TIME is record
  DATE : JULIAN_DATE;
  TIME : DURATION;
end record;

type LOGICAL_LINE is range 0..50;
type PART_NAME is (MCB, HEADER, MSG_BODY, TRAILER);
type PRECEDENCE is (ROUTINE, PRIORITY, IMMEDIATE, FLASH, ECP, CRITIC);

type SECURITY_CLASSIFICATION is (UNCLASS, EFTO, RESTRICTED, CONFIDENTIAL, SECRET, TOP_SECRET, SPECAT, DSSCS);
GLOBAL_TYPES

end GLOBAL_TYPES;
with RI_OPS, LINE_TBL_OPS, SEGMENT_OPS, GLOBAL_TYPES, MESSAGE_OPS;
use RI_OPS, LINE_TBL_OPS, SEGMENT_OPS, GLOBAL_TYPES, MESSAGE_OPS;

package HISTORY_OPS is

-- NAME: HISTORY_OPS
-- PURPOSE: contains data definitions and operations pertaining to
-- the history files (log and journal).
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

---

type CANCEL_REASON is (PREEMPTED, REJ_BY_RCVR);
type ENTRY_TYPE is (LOG, RI_SET, SEGMENT);
type HIST_STATUS is (OK, END_OF_MEDIA, ERROR);
type LOG_ENTRY is (SOM_IN, EOM_IN, REJECT, CANCEL_IN, CANTRAN_IN,
  SOM_OUT, EOM_OUT, CANCEL_OUT, CANTRAN_OUT,
  SCRUB, OUT_TO_OVFL, IN_FROM_OVFL, OUT_TO_INT,
  IN_FROM_INT, SVC_GEN, VERSION_OUT, VERSION_IN);
type MESSAGE_LIST is array (INTEGER range <>) of EXT_SERIAL_NO;
type NUM_BLOCKS is range 0..50;
type REJECT_REASON is (INVALID_HDR, INVALID_RI,
  INVALID_SECURITY, SECURITY_MISMATCH);
type RI_ARRAY is array (NUM_RIS range <>) of RI_STRING;
type SCRUB_REASON is (OPERATOR);

type HISTORY_ENTRY (ENT : ENTRY_TYPE := SEGMENT;
  RI_COUNT : NUM_RIS := 0;
  LOG_TYPE : LOG_ENTRY := SOM_IN) is

  record
    case ENT is
      when LOG =>
        BLOCK_COUNT : NUM_BLOCKS; -- Blocks transmitted or received
        CHANNEL : CHANNEL_DESC; -- Channel designator
        HACK : DATE_TIME; -- Time of event
        HEADER_SEGS : NUM_BLOCKS; -- Number of header segments which
          -- are present in the log entry
          -- May be zero
        MCB : BOOLEAN; -- Whether an MCB segment
          -- is present
        MODE : CHANNEL_MODE; -- Mode of the channel
        RIS : BOOLEAN; -- Whether an RI_set is included
        SEQ_NUM : CSN; -- CSN for asynch only
        SER_NO : EXT_SERIAL_NO; -- Message extended serial number
      when REJECT =>
        REJ_REASON : REJECT_REASON;
      when CANCEL_OUT =>
        CAN_REASON : CANCEL_REASON;
      when SCRUB =>

---
HISTORY_OPS

SCR_REASON : SCRUB_REASON;
when Others =>
  null;
end case;
when RI_SET =>
  R1_S: R1 ARRAY (1..RI_COUNT);
when SEGMENT =>
  SEG_BUF : SEGMENT_OPS.SEGMENT;
end case;
end record;

type ENTRY_SET is array (INTEGER range <>) of HISTORY_ENTRY;
-- An ENTRY_SET consists of a HISTORY_ENTRY of type LOG followed by:
  -- An optional RI SET
  -- An optional MCB SEGMENT
  -- 0 or more header SEGMENTS
-- This ordering MUST NOT be violated

task ENTER_HIST is
  entry WRITE(REC : ENTRY_SET;
               STATUS: out HIST_STATUS);
end ENTER_HIST;
-- USED TO WRITE A RECORD OR SET OF RECORDS ON THE HISTORY

procedure SEARCH_LOG(SER_NO : EXT_SERIAL_NO;
                    REC : out ENTRY_SET;
                    STATUS : out HIST_STATUS);
-- SEARCHES LOG FOR THE NEXT LOG ENTRY FOR A PARTICULAR EASN

procedure READ_LOG(REC : out ENTRY_SET;
                   STATUS : out HIST_STATUS);
-- READS THE NEXT LOG ENTRY -- MAY SKIP REFERENCE ENTRIES

procedure REWIND_LOG;
-- RESETS LOG TO THE BEGINNING OF THE SHORT HISTORY

procedure INIT_HISTORY;
-- STARTS A FRESH HISTORY

procedure NEW_DAY;
-- CLOSES PREVIOUS DAY'S HISTORY AND STARTS NEW DAY
  -- CLOSES OLD DAY'S TAPE
  -- WRITES CURRENT RUNNING AUDIT ON END OF OLD DAY
  -- WRITES CURRENT RUNNING AUDIT ON START OF NEW DAY
  -- OPENS NEW DAY'S TAPE
  -- AUDITS OLD DAY TAPE AND COMPARES WITH RUNNING AUDIT
  -- PRINTS STATISTICAL SUMMARY OF DAY'S ACTIVITIES

procedure CLOSE_HISTORY;
-- CLOSES HISTORY WITHOUT STARTING A NEW ONE
HISTORY_OPS

-- Closes old day's tape
-- Writes current running audit to end of tape
-- Audits tape and compares with running audit
-- Prints a statistical summary of day's activities

procedure RE_ENTER_MESSAGES(List : MESSAGE_LIST;
        Status : out HISTSTATUS;
        Not_Found : out MESSAGE_LIST);
-- Reads a list of messages from history into intransit and
-- Places on queue
-- Returns a list of messages not recovered

procedure READ_MESSAGE(Ser_No : EXT_SERIAL_NO;
        Ptr : out MSGID;
        Status : out HISTSTATUS);
-- Reads a message into intransit and returns a msgid

end HISTORY_OPS;
package INTERFACE_TO_8254 is

-- Name: INTERFACE_TO_8254
-- Purpose: interface to the Intel 8254 counter/timer IC
-- Programmer: Brian G. Sharpe
-- Date: 14 June 1982
-- Description:
-- This package provides all of the necessary interfaces to the Intel
-- 8254 counter/timer IC. The necessary byte formats are defined as
-- types and subprograms are provided in order to provide information
-- hiding.
-- This package is not complete; basically, only those subprograms
-- that are required have been defined and coded.
--
-- NOTE
-- For sake of simplicity, memory-mapped I/O is assumed to be used.
-- Use of non-memory-mapped I/O will require something other than
-- SHARED_VARIABLE_UPDATE in order to read and write to the hardware.

-- declaration section:

type TYPE_OF_COUNTING_FORMAT is (BINARY, BCD);

-- The following modes determine the operating modes of the
-- programmable 8254.

type PERMISSABLE_Modes is
  ( MODE_0, -- interrupt on terminal count
    MODE_1, -- hardware retriggerable one-shot
    MODE_2, -- rate generator
    MODE_3, -- square wave mode
    MODE_4, -- software triggered strobe
    MODE_5); -- hardware triggered strobe (retriggerable

type PERMISSABLE_CHANNELS_OR_READ_BACK is
  ( CHANNEL_0, -- select counter channel 0
    CHANNEL_1,
    CHANNEL_2,
    READ_BACK ); -- select read-back command mode

subtype PERMISSABLE_CHANNELS is PERMISSABLE_CHANNELS_OR_READ_BACK
  range CHANNEL_0 .. CHANNEL_2;

type CHANNEL_WORD_ARRAY is array( CHANNEL_0 .. CHANNEL_2 ) of WORD;
type SELECT_CHANNEL_ARRAY is array( CHANNEL_0 .. CHANNEL_2 ) of BOOLEAN;

-106-
-- read any combination of the three 16-bit channels
function READ_MULTIPLE_CHANNELS
  ( READ_CHNL : in SELECT_CHANNEL_ARRAY )
  return CHANNEL_WORD_ARRAY;

-- get the clock frequencies of the given channel
function CHANNEL_CLOCK_FREQ ( CHANNEL_NO : in PERMISSABLE_CHANNELS )
  return FLOAT;

-- set the counting format (binary or BCD) of the given channel
procedure SET_COUNTING_FORMAT
  ( CHANNEL_NO : in PERMISSABLE_CHANNELS;
    COUNTING_FORMAT : in TYPE_OF_COUNTING_FORMATS );

-- set the operating mode of the given channel
procedure SET_OPERATING_MODE
  ( CHANNEL_NO : in PERMISSABLE_CHANNELS;
    OPERATING_FORMAT : in PERMISSABLE_MODES );

end INTERFACE_TO_8254;
MESSAGE_OPS

with GLOBALTYPES,SEGMENT_OPS,MANAGE_STORAGE;
use GLOBALTYPES,SEGMENT_OPS,MANAGE_STORAGE;

package MESSAGE_OPS is

-- NAME: MESSAGE_OPS
-- PURPOSE: contains data definitions and operation definitions
dealing with the message.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

type BLOCK_COUNT is new STRING(1..3);
type ERRSTAT is (OK,ENDOFTEXT,LINKERROR,OTHERERROR);
type FORMAT is (ACP,JANAP);
type MESSAGECATEGORY is (NORMAL,SERVICE);
type NUM_RIS is range 0..50;
type PART_HEADER is private;
type PART_PTR is access PART_HEADER;
type POSITION is limited private;
type RI_STAT is (OK,LASTRIERROR);
type SEG_ARRAY is array(NATURAL range <=) of SEGMENT;
type VERSIONHEADER is private;
type MSGID is access VERSIONHEADER;

function ESTABLISH_VERSION(EASN : EXT_SERIAL_NO) return MSGID;
-- ESTABLISH A MESSAGE VERSION HEADER
-- HEADER HAS NO PARTS

procedure OPEN_FOR_READ(MESSAGE : MSGID;
START_WITH : PART_NAME;
POS:out POSITION);
-- ESTABLISH INITIAL POSITION FOR READING FROM A MESSAGE

procedure READ_CONTINUOUS(MESSAGE : MSGID;
COUNT : in out CHAR_COUNT;
WHERE_FROM : in out POSITION;
TEXT : out STRING;
STATUS : out ERRSTAT);

procedure READ_FROM_PART(MESSAGE : MSGID;
COUNT : in out CHAR_COUNT;
WHERE_FROM : in out POSITION;
TEXT : out STRING;
STATUS : out ERR_STAT);
-- READ COUNT CHARACTERS STARTING FROM WHERE_FROM
-- CHARACTERS ARE RETURNED IN TEXT
-- IF STATUS IS END_OF_TEXT, COUNT WILL REFLECT ACTUAL NUMBER
-- OF CHARACTERS READ
-- CHECK FOR LINKING ERRORS AND RETURNS ERROR IF SEGMENTS
-- ARE NOT LINKED IN A CONSISTENT FASHION
procedure ATTACH_SEGMENT(MESSAGE : MSGID; SEG : SEG_PTR);  
-- ATTACHES A SEGMENT TO THE MESSAGE IN THE PART  
     -- SPECIFIED IN THE SEGMENT  
     -- IF NO PART_HEADER EXISTS, ONE WILL BE CREATED
 procedure WRITE_TO_PART(MESSAGE : MSGID;  
            PART : PART_NAME;  
            TEXT : STRING);  
-- WRITES TEXT TO THE MESSAGE AND PART SPECIFIED  
-- CREATES PART_HEADERS AND SEGMENTS AS REQUIRED
 procedure ATTACH_PART(MESSAGE, OLD_MESSAGE : MSGID;  
            PART : PART_NAME);  
-- MAKES ADDITIONAL USE OF PART IN OLD_MESSAGE IN THE CURRENT  
     -- MESSAGE  
     -- INCREMENTS USER_COUNT IN PART_HEADER
 procedure FREE_PART(MESSAGE : MSGID; PART : PART_NAME);  
-- DETACHES AND FREES (IF NO OTHER USE) A SPECIFIED PART FROM  
     -- A MESSAGE

function CALCULATE_BLOCK_COUNT (MESSAGE : MSGID) return BLOCK_COUNT;  
-- CALCULATES THE NUMBER OF 80 CHARACTER OUTPUT BLOCKS  
     -- OCCUPIED BY A MESSAGE  
     -- RETURNS THE NUMBER IN STRING FORM  
     -- DOES NOT ALLOW FOR MCB
 procedure SET_BLOCK_COUNT(MESSAGE : MSGID; COUNT : BLOCK_COUNT);  
     -- SET BLOCK COUNT IN MCB

task FREE_VER is  
    entry FREE_VERSION(MESSAGE : MSGID);  
    -- FREES HEADERS AND SEGMENTS WHICH ARE NOT IN USE BY OTHER  
        -- VERSIONS  
end FREE_VER;

function READ_EASN(MESSAGE : MSGID) return EXT_SERIAL_NO;  
-- READ EXTENDED SERIAL NUMBER

procedure SET_CATEGORY(MESSAGE : MSGID; CATEGORY : MESSAGE_CATEGORY);  
function READ_CATEGORY(MESSAGE : MSGID) return MESSAGE_CATEGORY;  
-- READ AND SET MESSAGE CATEGORY (NORMAL OR SERVICE)

procedure SET_CLASS(MESSAGE : MSGID;  
                   CLASS : SECURITY_CLASSIFICATION);  
function READ_CLASS(MESSAGE : MSGID) return SECURITY_CLASSIFICATION;  
-- SET AND READ SECURITY CLASSIFICATION OF MESSAGE

procedure SET_NUM_RIS(MESSAGE : MSGID; RI_COUNT : NUM_RIS);
MESSAGE_OPS

function READ_NUM_RIS(MESSAGE : MSGID) return NUM_RIS;
-- SET AND READ THE NUMBER OF RIS

procedure SET_TIME_OF_RECEIPT(MESSAGE : MSGID; TOR : DATE TIME);
function READ_TIME_OF_RECEIPT(MESSAGE : MSGID) return DATE_TIME;
-- SET AND READ TIME OF RECEIPT

procedure SET_PRECEDENCE(MESSAGE : MSGID; PREC : PRECEDENCE);
function READ_PRECEDENCE(MESSAGE : MSGID) return PRECEDENCE;
-- SET AND READ PRECEDENCE

procedure SET_LOGICAL_LINE(MESSAGE : MSGID;
LINE_NUMBER : LOGICAL_LINE);
function READ_LOGICAL_LINE(MESSAGE : MSGID) return LOGICAL_LINE;
-- SET AND READ LOGICAL OUTPUT LINE NUMBER

procedure SET_CHAR_TYPE(MESSAGE : MSGID; SET : CHAR_SET_TYPE);
function READ_CHAR_TYPE(MESSAGE : MSGID) return CHAR_SET_TYPE;
-- SET AND READ CHARACTER SET TYPE

procedure SET_FORMAT(MESSAGE : MSGID; FMT : FORMAT);
function READ_FORMAT(MESSAGE : MSGID) return FORMAT;
-- SET AND READ MESSAGE FORMAT

procedure FIND_CLASS(MESSAGE : MSGID;
CLASS : out STRING;
ERROR : out BOOLEAN);
procedure FIND_LMF(MESSAGE : MSGID;
LMF : out STRING;
ERROR : out BOOLEAN);
-- RETURN THE DESIRED PORTION OF THE HEADER

procedure FIND_FIRST_RI(MESSAGE : MSGID;
POS : out POSITION;
STATUS : out RI_STAT);
-- RETURNS THE POSITION OF THE FIRST RI IN A MESSAGE

procedure FIND_RI(MESSAGE : MSGID;
POS : in out POSITION;
RI : out STRING;
STATUS : out RI_STAT);
-- CALLED WITH POS POINTING TO AN RI
-- RETURNS THAT RI AND SETS POS TO NEXT RI

function NUM_SEGMENTS(MSG : MSGID;
PART : PART_NAME) return SEGMENT_NUMBER;
-- RETURNS THE NUMBER OF SEGMENTS IN A PART OF A MESSAGE

procedure GET_PART(MSG : MSGID;
PART : PART_NAME);
MESSAGE_OPS

SEGS : out SEG_ARRAY;
-- READS THE SEGMENTS OF A PART INTO THE ARRAY SEGS FOR USE BY HISTORY

function GET is new GET_INTRINSIT(PART_HEADER, PART_PTR);
function GET is new GET_INTRINSIT(VERSION HEADER, MSGID);
procedure FREE is new FREE_INTRINSIT(PART_HEADER, PART_PTR);
procedure FREE is new FREE_INTRINSIT(VERSION HEADER, MSGID);
-- GET AND FREE STORAGE FOR VERSION_HEADERS AND PART_HEADERS
-- IN THE INTRANSIT STORAGE AREA. SEE MANAGE_STORAGE FOR DETAILS.

private

type USER COUNTER is range 0..50;
type PART_HEADER is
  record
    FIRST SEGMENT : SEG_PTR;
    NEXT SEGMENT : SEG_PTR;
    SEGMENT COUNT : SEGMENT_NUMBER;
    PART : PART_NAME;
    USER COUNT : USER COUNTER;
  end record;
type PARTS ARRAY is array(PART NAME) of PART PTR;
type VERSION HEADER is
  record
    CATEGORY : MESSAGE CATEGORY;
    CHAR TYPE : CHAR SET_TYPE;
    CLASS : SECURITY CLASSIFICATION;
    EASN : EXT SERIAL NO;
    FMT : FORMAT;
    LOGICAL OUTPUT LINE : LOGICAL LINE;
    PREC : PRECEDENCE;
    RI COUNT : NUM_RIS;
    SECTIONS : PARTS ARRAY;
    TIME OF RECEIPT : DATE TIME;
  end record;
type POSITION is
  record
    SEG : SEG_PTR;
    COUNT : CHAR_COUNT;
  end record;
end MESSAGE_OPS;
package SERVICE_MESSAGE_OPTS is

-- NAME: SERVICE_MESSAGE_OPTS
-- PURPOSE: contains data definitions and operations required for
-- generating service messages.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982

type SVC_MSG_TYPE is (ALL RI INVALID, EXCESSIVE_ROUTING_REJ,
  HI_PREC_ACC, ILLEGAL_EXCHANGE, INCORRECT_CSN,
  INPUT_SCTY_MISMATCH, INVALID_BLOCK_COUNT, INVALID_EOL_ACC,
  INVALID_EOL_REJ, INVALID_HEADER_ACC, INVALID_HEADER_REJ,
  INVALID_RI, INVALID_RI_FIELD, INVALID_SCTY_FIELD,
  INVALID_TI_ACC, INVALID_TI_REJ, NO_EOL, OPEN_CSN,
  OUTPUT_SCTY_MISMATCH, SUSPECTED_STRAGGLER,
  SUSPENDED_TRANSMISSION, TRAFFIC_CHECK, TWO_CONSEC_SOM);

type SVC_MSG_INFO is record
  MSG_REF: MSGID;
  MSG_TYPE: SVC_MSG_TYPE;
  RIS: RI_PTR;
  LOW_CSN: CSN;
  HIGH_CSN: CSN;
end record;

task type GEN_SVC is
  entry GENERATE_SVC_MESSAGE(INFO:SVC_MSG_INFO);
end GEN_SVC;

end SERVICE_MESSAGE_OPTS;
package TASKS is

-- NAME: TASKS
-- PURPOSE: Data structure used to correlate tasks with the line number
-- of the physical line they are servicing.

type OUTPUT_PTR is access OUTPUT_MESSAGE;
type SEND_II_IV_PTR is access SEND_MODE_II_IV;
type SEND_V_PTR is access SEND_MODE_V;
type SEND_I_PTR is access SEND_MODE_I;
type CONTROL_PTR is access CONTROL_CHARACTER;

type TASK_LIST is record
  HANDLER: MANAGER;
  OUTPUT: OUTPUT_PTR;
  SEND_II_IV: SEND_II_IV_PTR;
  SEND_V: SEND_V_PTR;
  SEND_I: SEND_I_PTR;
  GENCC,RECVCC: CONTROL_PTR;
end record;

TABLE: array ( PHYSICAL_LINE ) of TASK_LIST;

end TASKS;
package TIME_OPS is

-- NAME: TIME_OPS
-- PURPOSE: provides operations on the type DATE_TIME
-- PROGRAMMER: Paul Dobbs
-- DATE: May 14, 1982

function READ_CLOCK return DATE_TIME;
-- READS THE SYSTEM TIME AND CONVERTS IT TO A JULIAN DATE_TIME

function "<" (T1, T2 : DATE_TIME) return BOOLEAN;
function "<=" (T1, T2 : DATE_TIME) return BOOLEAN;
function ">" (T1, T2 : DATE_TIME) return BOOLEAN;
function ">=" (T1, T2 : DATE_TIME) return BOOLEAN;
-- provide comparison facilities for the Julian DATE_TIME type
end TIME_OPS;
3.2 Package Specification Dependencies

The chart in Figure 3.2-1 illustrates the package specification dependencies. This is especially useful if a package must be recompiled, since the chart shows which other packages must also be recompiled. Any package to the right of the one being recompiled that is connected by a circle must also be recompiled.
3.3 Traceability Matrix

The following section shows forward and backward traceability. The forward direction is from the Requirements Specification to the Functional Decomposition Charts illustrated in Section 2.4 of this document. Reverse traceability is the opposite of the above process.
<table>
<thead>
<tr>
<th>REQUIREMENT</th>
<th>DESIGN MODULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A11</td>
<td>RUN SWITCH</td>
</tr>
<tr>
<td>A121</td>
<td>AUDIT.AUDIT_INTRASIT</td>
</tr>
<tr>
<td>A122</td>
<td>HISTORY_OPS_RE_ENTER_MESSAGES</td>
</tr>
<tr>
<td>A131</td>
<td>MONITOR_HARDWARE_ERRORS</td>
</tr>
<tr>
<td>A1321</td>
<td>AUDIT.AUDIT_OVERFLOW</td>
</tr>
<tr>
<td>A1322</td>
<td>RUN SWITCH</td>
</tr>
<tr>
<td>A1331</td>
<td>AUDIT.AUDIT_INTERCEPT</td>
</tr>
<tr>
<td>A1332</td>
<td>RUN SWITCH</td>
</tr>
<tr>
<td>A134</td>
<td>MONITOR_HARDWARE_ERRORS</td>
</tr>
<tr>
<td>A311</td>
<td>BUILD_MESSAGE_FROM_SEGMENTS</td>
</tr>
<tr>
<td>A312</td>
<td>BUILD_VALID_ASYNC_SEG.RECEIVE_CHARS</td>
</tr>
<tr>
<td>A313</td>
<td>BUILD_SYNC_SEGMENT.RECEIVE_CHARS</td>
</tr>
<tr>
<td>A3141</td>
<td>FORMAT_ASYNC_SEGMENT</td>
</tr>
<tr>
<td>A3142</td>
<td>RECOGNIZE_START_OF_MESSAGE</td>
</tr>
<tr>
<td>A3143</td>
<td>FORMAT_ASYNC_SEGMENT</td>
</tr>
<tr>
<td>A3144</td>
<td>RECOGNIZE_ENDING_SEQUENCE</td>
</tr>
<tr>
<td>A3151</td>
<td>PROCESS_CONTROL_SEQUENCE</td>
</tr>
<tr>
<td>A3152</td>
<td>IDENTIFY_1ST_CHARACTER</td>
</tr>
<tr>
<td>A3153</td>
<td>VALIDATE_2ND_CHARACTER</td>
</tr>
<tr>
<td>A3154</td>
<td>BUILD_SYNC_SEGMENT</td>
</tr>
<tr>
<td>A3155</td>
<td>INTERPRET_CC</td>
</tr>
<tr>
<td>A3156</td>
<td>VALIDATE_83RD_CHARACTER</td>
</tr>
<tr>
<td>A3157</td>
<td>INTERPRET_CC</td>
</tr>
<tr>
<td>A3161</td>
<td>CHECK_BLK_PARITY</td>
</tr>
<tr>
<td>A3162</td>
<td>VALIDATE_FIRST_SEGMENT</td>
</tr>
<tr>
<td>A3163</td>
<td>SET_ECSN</td>
</tr>
<tr>
<td>A3164</td>
<td>CHECK_MODE_II_IV_ECSN</td>
</tr>
<tr>
<td>A3165</td>
<td>READ_SET_E7_ECSN</td>
</tr>
<tr>
<td>A3166</td>
<td>CHECK_MODE_V_ECSN</td>
</tr>
<tr>
<td>A3167</td>
<td>READ_SET_ECSN</td>
</tr>
<tr>
<td>A3168</td>
<td>VALIDATE_FIRST_SEGMENT</td>
</tr>
<tr>
<td>A3169</td>
<td>READ_CHNL_DES</td>
</tr>
<tr>
<td>A3170</td>
<td>COMPLETE_TI</td>
</tr>
<tr>
<td>A32</td>
<td>GENERATE_CONTROL_CHARACTERS</td>
</tr>
<tr>
<td>A331</td>
<td>VALIDATE_MESSAGE</td>
</tr>
<tr>
<td>A3321</td>
<td>CHECK_JANAP_HEADER</td>
</tr>
<tr>
<td>A3322</td>
<td>READ_SECURITY_PROSIGN</td>
</tr>
<tr>
<td>A3323</td>
<td>CHECK_RECORD_COUNT</td>
</tr>
<tr>
<td>A3324</td>
<td>CHECK_RIS</td>
</tr>
<tr>
<td>A3325</td>
<td>CHECK_RIS</td>
</tr>
<tr>
<td>A3331</td>
<td>CHECK_ACP_HEADER</td>
</tr>
<tr>
<td>A3332</td>
<td>CHECK_RIS</td>
</tr>
<tr>
<td>A3333</td>
<td>COMPLETE_ACP_HEADER</td>
</tr>
<tr>
<td>A3334</td>
<td>READ_SECURITY_PROSIGN</td>
</tr>
<tr>
<td>A3341</td>
<td>CHECK_EOM</td>
</tr>
<tr>
<td>A3342</td>
<td>CHECK_LMF</td>
</tr>
<tr>
<td>A3343</td>
<td>CHECK_HPJ_HEADER</td>
</tr>
</tbody>
</table>
## FORWARD TRACEABILITY

<table>
<thead>
<tr>
<th>REQUIREMENT</th>
<th>DESIGN MODULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3342</td>
<td>READ_SECURITY_PROSIGN</td>
</tr>
<tr>
<td>A3343</td>
<td>CHECK_RECORD_COUNT</td>
</tr>
<tr>
<td>A3344</td>
<td>CHECK_RIS</td>
</tr>
<tr>
<td>A3345</td>
<td>CHECK_EOM</td>
</tr>
<tr>
<td>A3346</td>
<td>CHECK_EOT</td>
</tr>
<tr>
<td>A3351</td>
<td>CHECK_HPA_HEADER</td>
</tr>
<tr>
<td>A3352</td>
<td>CHECK_RIS</td>
</tr>
<tr>
<td>A3353</td>
<td>COMPLETE_HPA_HEADER</td>
</tr>
<tr>
<td>A3354</td>
<td>READ_SECURITY_PROSIGN</td>
</tr>
<tr>
<td>A3355</td>
<td>CHECK_EOM</td>
</tr>
<tr>
<td>A341</td>
<td>PROCESS_MCB</td>
</tr>
<tr>
<td>A342</td>
<td>BUILD_JANAP_MCB</td>
</tr>
<tr>
<td>A343</td>
<td>BUILD_ACP_MCB</td>
</tr>
<tr>
<td>A344</td>
<td>MODIFY_MCB</td>
</tr>
<tr>
<td>A345</td>
<td>COMPLETE_MCB</td>
</tr>
<tr>
<td>A35</td>
<td>GENERATE_SERVICE_MESSAGE</td>
</tr>
<tr>
<td>A361</td>
<td>STORE_REFERENCE</td>
</tr>
<tr>
<td>A361</td>
<td>PROCESS_MCB</td>
</tr>
<tr>
<td>A361</td>
<td>MANAGE_OVERFLOW_MOVE_TO_OVERFLOW</td>
</tr>
<tr>
<td>A361</td>
<td>QUEUE_NORM_MSG_BY_PRECEDENCE</td>
</tr>
<tr>
<td>A361</td>
<td>QUEUE_SVS_MSG_BY_PRECEDENCE</td>
</tr>
<tr>
<td>A361</td>
<td>HISTORY_OPS_ENTER_HIST_WRITE</td>
</tr>
<tr>
<td>A362</td>
<td>MANAGE_INTRANSLIT_CALCULATE_THRESHOLD</td>
</tr>
<tr>
<td>A363</td>
<td>MANAGE_OVERFLOW_RECOVER_OVERFLOW</td>
</tr>
<tr>
<td>A363</td>
<td>MANAGE_MESSAGE_QUEUES_QUEUE_REINTROD</td>
</tr>
<tr>
<td>A363</td>
<td>HISTORY_OPS_ENTER_HIST_WRITE_HISTORY</td>
</tr>
<tr>
<td>A364</td>
<td>RETRVE_NEXT_MSG_FOR_PROCESSING</td>
</tr>
<tr>
<td>A364</td>
<td>QUEUE_FROM_FRONT</td>
</tr>
<tr>
<td>A411</td>
<td>GET_RIS</td>
</tr>
<tr>
<td>A412</td>
<td>GET_RIS</td>
</tr>
<tr>
<td>A413</td>
<td>SEGREGATE_ROUTING_LINE</td>
</tr>
<tr>
<td>A413</td>
<td>GET_RIS</td>
</tr>
<tr>
<td>A421</td>
<td>TRANSLATE_MESSAGE</td>
</tr>
<tr>
<td>A4221</td>
<td>TRANSLATE_MESSAGE</td>
</tr>
<tr>
<td>A4222</td>
<td>CONVERT_TO_JANAP</td>
</tr>
<tr>
<td>A4223</td>
<td>CONVERT_TO_ACP</td>
</tr>
<tr>
<td>A4224</td>
<td>TRANSLATE_MESSAGE</td>
</tr>
<tr>
<td>A4225</td>
<td>REMOVE_DBLs</td>
</tr>
<tr>
<td>A4221</td>
<td>TRANSLATE_MESSAGE</td>
</tr>
<tr>
<td>A4232</td>
<td>TRANSLATE_TO_ASCII</td>
</tr>
<tr>
<td>A4233</td>
<td>TRANSLATE_TO_ITA2</td>
</tr>
<tr>
<td>A4234</td>
<td>TRANSLATE_MESSAGE</td>
</tr>
<tr>
<td>A4235</td>
<td>TRANSLATE_TO_CARD</td>
</tr>
<tr>
<td>A4236</td>
<td>TRANSLATE_FROM_CARD</td>
</tr>
<tr>
<td>A4237</td>
<td>TRANSLATE_MESSAGE</td>
</tr>
<tr>
<td>A4238</td>
<td>FILL_LAST_BLOCK</td>
</tr>
<tr>
<td>A431</td>
<td>MANAGE_TRANSLATED_MESSAGE_QUEUES</td>
</tr>
<tr>
<td>A432</td>
<td>MOVE_FROM_INTERCEPT_TO_INTRANSLIT</td>
</tr>
<tr>
<td>A433</td>
<td>RETRIEVE_NEXT_MESSAGE</td>
</tr>
<tr>
<td>A433</td>
<td>QUEUE_FROM_FRONT</td>
</tr>
</tbody>
</table>
## For 3rd Traceability

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Design Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>A511</td>
<td>MESSAGE_OPS.READ_CONTINUOUS</td>
</tr>
<tr>
<td>A512</td>
<td>VALIDATE_RI</td>
</tr>
<tr>
<td>A513</td>
<td>VALIDATE_LMF</td>
</tr>
<tr>
<td>A513</td>
<td>FIND_LMF</td>
</tr>
<tr>
<td>A514</td>
<td>VALIDATE_SECURITY</td>
</tr>
<tr>
<td>A514</td>
<td>READ_LOGICAL_LINE</td>
</tr>
<tr>
<td>A514</td>
<td>FIND_CLASS</td>
</tr>
<tr>
<td>A515</td>
<td>SENDASYNCH</td>
</tr>
<tr>
<td>A531</td>
<td>GEN_FRAME_CHARS</td>
</tr>
<tr>
<td>A5321</td>
<td>FRAME_BLOCK</td>
</tr>
<tr>
<td>A5322</td>
<td>FRAME_BLOCK</td>
</tr>
<tr>
<td>A5323</td>
<td>OUTPUT_MESSAGE</td>
</tr>
<tr>
<td>A541</td>
<td>SEND_SYNCH</td>
</tr>
<tr>
<td>A5411</td>
<td>SEND_SYNCH</td>
</tr>
<tr>
<td>A5411</td>
<td>TRANSMIT_SYNCH_SYNCH</td>
</tr>
<tr>
<td>A5412</td>
<td>SEND_SYNCH</td>
</tr>
<tr>
<td>A5412</td>
<td>TRANSMIT_SYNCH_CHARS</td>
</tr>
<tr>
<td>A5413</td>
<td>SEND_SYNCH</td>
</tr>
<tr>
<td>A5413</td>
<td>TRANSMIT_SYNCH_CHARS</td>
</tr>
<tr>
<td>A5414</td>
<td>SEND_SYNCH</td>
</tr>
<tr>
<td>A5414</td>
<td>TRANSMIT_SYNCH_CHARS</td>
</tr>
<tr>
<td>A5421</td>
<td>GENERATED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5421</td>
<td>RECEIVED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5422</td>
<td>GENERATED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5422</td>
<td>RECEIVED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5423</td>
<td>GENERATED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5423</td>
<td>RECEIVED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5424</td>
<td>GENERATED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5424</td>
<td>RECEIVED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5425</td>
<td>GENERATED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5425</td>
<td>RECEIVED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5426</td>
<td>GENERATED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A5426</td>
<td>RECEIVED_CONTROL_CHARS</td>
</tr>
<tr>
<td>A543</td>
<td>OUTPUT_MESSAGE</td>
</tr>
<tr>
<td>A543</td>
<td>READ_CHANNEL_MODE</td>
</tr>
<tr>
<td>A551</td>
<td>HISTORY_OPS.ENTER_HIST.WRITE</td>
</tr>
<tr>
<td>A552</td>
<td>HISTORY_OPS.ENTER_HIST.WRITE</td>
</tr>
<tr>
<td>A553</td>
<td>HISTORY_OPS.ENTER_HIST.WRITE</td>
</tr>
<tr>
<td>A554</td>
<td>HISTORY_OPS.ENTER_HIST.WRITE</td>
</tr>
<tr>
<td>A555</td>
<td>HISTORY_OPS.ENTER_HIST.WRITE</td>
</tr>
<tr>
<td>A556</td>
<td>HISTORY_OPS.ENTER_HIST.WRITE</td>
</tr>
</tbody>
</table>

### Non Functional Requirements

1. **Sentence 1**
   - RECEIVE VALID MESSAGE
   - MANAGE MESSAGE QUEUES

2. **Sentence 2**
   - THROTTLE_INPUT
   - RECEIVE_CHARS
   - EVERYTHING
FORWARD TRACEABILITY

REQUIREMENT

II.A.(3) RECEIVE CHARS
II.A.(4) RECEIVE CHARS
II.A.(5) BUILD VALID SYNCH SEG
II.A.(6) BUILD VALID SYNCH SEG
II.A.(7) FORMAT SYNCH SEG
II.A.(8) FORMAT SYNCH SEG.RECEIVE CHAR
II.A.(9) PROCESS MCB
II.A.(10) RECEIVE CHAR
II.A.(11) FORMAT SYNCH SEG.RECEIVE CHAR
II.A.(12) FORMAT SYNCH SEG.RECEIVE CHAR
II.B.(1) FORMAT SYNCH SEG.RECEIVE CHAR
II.B.(2) SAME AS II.A(1) - II.A.11)
III.A.(1) & (2) BUILD VALID ASYNCH SEG.RECEIVE CHAR
III.A.(1) & (2) XMIT ISYNCH-CHAR
III.A.(3) THE WHOLE SWITCH
III.B.(1) & (2) & (3) BUILD VALID ASYNCH SEG.RECEIVE CHAR
III.B.(1) & (2) & (3) XMIT ASYNCH CHAR
IV.A.(1) & (2) RECEIVE CHAR AND XMIT CHAR
IV.B.(1) & (2) RECEIVE CHAR AND XMIT CHAR
V.SENTENCE 1 SEGMENT OPS
V.SENTENCE 2 NOT APPLICABLE
V.SENTENCE 3 PACKAGE MANAGE INTRANSIT
V.SENTENCE 4 NOT APPLICABLE
V.SENTENCE 5 NOT APPLICABLE
V.SENTENCE 6 (DISTRIBUTION) NOT APPLICABLE
VI. UNDETERMINABLE AT THIS TIME
VII. UNDETERMINABLE AT THIS TIME
IX. RI OPS
X. MANAGE INTRANSIT
XI. UNDETERMINABLE AT THIS TIME

CONCURRENCY CHART

STARTUP/RECOVERY
ASSEMBLE MESSAGES
PROCESS MESSAGES
TRANSMIT MESSAGES
OPERATOR INTERFACE

DESIGN MODULE

RECEIVE CHARS
RECEIVE CHARS
BUILD VALID SYNCH SEG
BUILD VALID SYNCH SEG
FORMAT SYNCH SEG
FORMAT SYNCH SEG.RECEIVE CHAR
PROCESS MCB
RECEIVE CHAR
FORMAT SYNCH SEG.RECEIVE CHAR
FORMAT SYNCH SEG.RECEIVE CHAR
SAME AS II.A. (1) - II.A.11)
BUILD VALID ASYNCH SEG.RECEIVE CHAR
XMIT ASYNCH CHAR
THE WHOLE SWITCH
BUILD VALID ASYNCH SEG.RECEIVE CHAR
XMIT ASYNCH CHAR
RECEIVE CHAR AND XMIT CHAR
RECEIVE CHAR AND XMIT CHAR
SEGMENT OPS
NOT APPLICABLE
PACKAGE MANAGE INTRANSIT
NOT APPLICABLE
NOT APPLICABLE
NOT APPLICABLE
UNDETERMINABLE AT THIS TIME
UNDETERMINABLE AT THIS TIME
RI OPS
MANAGE INTRANSIT
UNDETERMINABLE AT THIS TIME

DESIGN DOCUMENT COMPONENT

RUN SWITCH
RECEIVE VALID MESSAGE
PROCESS MESSAGE
OUTPUT MESSAGE
MONITOR SWITCH
D.I. = DESIGN IMPLEMENTATION

ADD TEXT
ASH_GET

ASYNCH_SEG.RECOGNIZE_END_OF_PART

AUDIT.AUDIT_INTERCEPT
AUDIT.AUDIT_INTRANSLIT
AUDIT.AUDIT_OVERFLOW
BUILD_ACP_MCB
BUILD_JANAP_MCB
BUILD_MESSAGE_FROM_SEGMENTS
BUILD_SYNCH_SEG
BUILD_SYNCH_SEGMENT
BUILD_SYNCH_SEGMENT_RECEIVE_CHARS
BUILD_VALIDASYNCH_SEG_RECEIVE_CHAR
BUILD_VALIDASYNCH_SEG_RECEIVE_CHAR
BUILDVALIDASYNCH_SEG_RECEIVE_CHARS
BUILDVALIDASYNCH_SEG
BUILDVALID_SYNCH_SEG
BUILDVALID_SYNCH_SEG
CHECK_ACP_HEADER
CHECK_BLK_PARITY
CHECK_EOM
CHECK_EOT
CHECK_HPA_HEADER
CHECK_HPJ_HEADER
CHECK_ICD
CHECK_JANAP_HEADER
CHECK_LMF
CHECK_MODE_II_IV_CSN
CHECK_MODE_V_CSN
CHECK_RECORD_COUNT
CHECK_RIS
CHECK_RIS
COMPLETE_ACP_HEADER
COMPLETE_HPA_HEADER
COMPLETE_MCB
COMPLETE_TI
CONVERT_TO_ACP
CONVERT_TO_JANAP
DEQUEUE_MSG_FOR_OVERFLOW

FILL_LAST_BLOCK
FIND_CLASS
FIND_LMF
FORMATASYNCH_SEGMENT
FORMATASYNCH_SEGMENT
FORMAT_SYNCH_SEG

D.I. - A3143, A3152, A3153, A3154
Fulfills requirement that unique ASN is assigned to each incoming message.
D.I. - breaks the transmission identifier of async messages into segment.

A1331
A1321
A343
A342
A311
A3154
A3151
A313
III.B.(1) & (2) & (3)
III.A.(1) & (2)
A312
II.A.(5)
II.A.(6)
A3331
A3157
A3324, A3334, A3345, A3354
A3325, A3346
A3351
A3342
A3164
A3321
A3341
A3162
A3341
A3322, A3343
A3163
A3322, A3332, A3344, A3354
A3352
A3323, A3332, A3344, A3354
A3333
A3353
A345
A3165
A4223
A4222
Implicit requirement - lower precedence messages overflow.

A4238
A514
A513
A3143
A3141
II.A.(7)
BACKWARD TRACEABILITY

DESIGN MODULE

FORMAT_SYNCH_SEG.RECEIVE_CHAR
FORMAT_SYNCH_SEG.RECEIVE_CHAR
FORMAT_SYNCH_SEG.RECEIVE_CHAR
FORMAT_SYNCH_SEG.RECEIVE_CHAR
FRAME_BLOCK
FRAME_BLOCK
FRAME_BLOCK
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_CONTROL_CHARS
GENERATED_SERVICE_MESSAGE
GEN_FRAME_CHARS
GET_RIS
GET_RIS
GET_RIS
GET_SEG
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
HISTORY_OPS.ENTER_HIST.WRITE
IDENTIFY 1ST_CHARACTER
INTERPRET_CC
INTERPRET_CC
MANAGE_INTRANSIT
MANAGE_INTRANSIT.CALCULATE_THRESHOLD
MANAGE_MESSAGE_QUEUES
MANAGE_MESSAGE_QUEUES.QUEUE_REINTROD
MANAGE_OVERFLOW.MOVE_TO_OVERFLOW
MANAGE_TRANSLATED_MESSAGE_QUEUES
MANAGE_TRANSLATED_MESSAGE_QUEUES
MESSAGE_IO
MESSAGE_OPS.ATTACH_SEGMENT
MESSAGE_OPS.CALCULATE_BLOCK_COUNT
MESSAGE_OPS.ESTABLISH_VERSION
MESSAGE_OPS.FREE_PART
MESSAGE_OPS.READ_CONTINUOUS
MESSAGE_OPS.SET_VLOCK_COUNT
MESSAGE_OPS_OPEN_FOR_READ
MESSAGE_OPS_READ_CONTINUOUS

REQUIREMENT

II.A.(8)
II.A.(11)
II.A.(12)
II.B.(1)
A5323
A5321
A5322
A5425
A5421
A5422
A5426
A5423
A5424
A32
A35
A531
A413
A411
A412
A361
A555
A551
A554
A552
A553
A556
A363
A122
A3152
A3154
A3156
X.
A362
A363
A361
A363
A431
A433
D.I.- break messages into segments.
D.I.- of internal storage of messages
D.I.- of message block counts
D.I.- of internal storage of messages
D.I.- of internal storage of messages
D.I.- of internal storage of messages
Reads/writes to hardware file (tape, disk, virtual memory).
D.I.- of internal storage of messages
D.I.- of internal storage of messages
Req. to keep block count in MCR update.
D.I.- of internal storage of messages
D.I.- of internal storage of messages
**BACKWARD TRACEABILITY**

**DESIGN MODULE**

- MODIFY_MCB
- MONITOR HARDWARE ERRORS
- MONITOR HARDWARE ERRORS
- MOVE_FROM_INTERCEPT_TO_INTRANSIT
- MOVE_TO_INTERCEPT
- MSG_OPS.FREE_VERSION
- MSG_OPS.READCATEGORY
- MSG_OPS.READ_PRECEDENCE
- MSG_OPS.READ_TIME_OF_RECEIPT
- MSG_OPS.SETCATEGORY
- MSG_OPS.SET_PRECEDENCE
- MSG_OPS.SET_TIME_OF_RECEIPT
- NOTIFY_OPERATOR
- OUTPUT_MESSAGE
- OUTPUT_MESSAGE
- PACKAGE_MANAGE_INTRANSIT
- PROCESS_CONTROL_SEQUENCE
- PROCESS_MCB
- PROCESS_MCB
- PROCESS_MCB
- PROCESS_MCB
- QUEUE_MESSAGE
- QUEUE_MSG_BY_PRECEDENCE_AND_BY_TOR
- QUEUE_NORM_MSG_BY_PRECEDENCE
- QUEUE_ON_FRONT
- QUEUE_ON_BACK
- QUEUE_MSG_BY_PRECEDENCE
- READ_CATEGORY
- READ_CHANNEL_DESCRIPTOR
- READ_CHANNEL_MODE
- READ_CHAR_SET
- READ_CLASS
- READ_EASN
- READ_ECSN
- READ_FORMAT
- READ_LOGICAL_LINE
- READ_RCSI
- READ_SECURITY_PROSIGN
- READ_TOR
- READY_LINE_NO

**REQUIREMENT**

- A344
- A134
- A131
- A432
  - Partially fulfills A431
- D.I. - removes outdated messages
  - Part of requirement of A364
- Part of requirement of A364 and A433
- Part of requirement of A364 and A433
- Part of requirement of A364
  - Part of requirement of A364 and A433
- Implementation of 'Notify Operator'
  - A543
  - A533
- V.SENTENCE 3
  - A3144
  - A341
  - II.A.(9)
- A361
  - Control of A4
  - Implicit requirement
  - Partially fulfills A431
  - A361
  - Implicit requirement
  - A361
- A361
  - D.I. to accomplish multiple routing
  - A3164
  - D.I. to make decision of A543
  - D.I. to accomplish multiple routing
  - D.I. to accomplish multiple routing
  - A3162, A3163
  - D.I. to accomplish multiple routing
  - A5426
  - A5425
  - A5421
  - A5422
  - A5423
  - II.A.(10)
  - IV.B.(1) & (2)
- IV.A.(1) & (2)
- II.A.(3)
BACKWARD TRACEABILITY

DESIGN MODULE

RECEIVE_CHARS
RECEIVE_CHARS
RECEIVE_CHARS
RECEIVE_VALID_MESSAGE
RECEIVE_VALID_MESSAGE
RECEIVED_CONTROL_CHARS
RECOGNIZE_ENDING_SEQUENCE
RECOGNIZE_ENDING_SEQUENCE
RECOGNIZE_START_OF_MESSAGE
REFLECT_STATUS
REMOVE_DBLS
RESET_SEG
RETRIEVE_NEXT_MESSAGE
RETRY_NEXT_MSG_FOR_PROCESSING
RI_OPS
RI_OPS.FIND_FIRST_RI
RI_OPS.FIND_RI
RI_OPS.READ_RI_TABLE
ROUTE_MESSAGE
RUN_SWITCH
RUN_SWITCH
RUN_SWITCH
SCAN_FOR_COLLECTIVE_RIS
SEGMENT_OPS

SEGMENT_OPS.NEXT_SEGMENT
SEGMENT_OPS.PRIOR_SEGMENT
SEGMENT_OPS.READ_CHARACTER_COUNT
SEGMENT_OPS.READ_CHARS
SEGMENT_OPS.READ_PART
SEGMENT_OPS.READ_SEGMENT_NUMBER
SEGMENT_OPS.READ_TIME
SEGMENT_OPS.SET_PART
SEGMENT_OPS.SET_SEGMENT_NUMBER
SEGMENT_OPS.SET_TIME
SEGMENT_OPS.WRITE_SPECIFIC
SEGMENT_OPS.WRITE_TO_PART
SEGREGATE_ROUTING_LINE
SENDASYNCH
SEND_SYNCH
SEND_SYNCH
SEND_SYNCH
SEND_SYNCH
SEND_SYNCH
SET_CATEGORY
SET_CHAR_SET
SET_CLASS
SET_ECSN
SET_FORMAT
SET_RCSN
SET_TOR

REQUIREMENT

II.A.(1)
II.A.(4)
I.A & B
D.I. - removes outdated messages
I.SENTENCE 1
A5424
A3166
A3143
A3142
Part of control of A3
A4225
D.I.
Partially fulfills A433
A364
IX.
D.I.
Same as FIND_FIRST_RI
D.I.
Control of A41
A1322
A11
A1332
Partial implementation of A345
V.SENTENCE 1
D.I. of internal storage of messages
D.I. of internal storage of messages
D.I.
D.I. of internal storage of segments
D.I.
Implicit requirement
See TIME_OPS.READ_CLOCK
D.I.
Implicit requirement
See TIME_OPS.READ_CLOCK
D.I. of internal storage of segments
D.I. of internal storage of messages
A413
A515
A5412
A541
A5414
A5411
A5413
D.I. to accomplish multiple routing
D.I. to accomplish multiple routing
D.I. to accomplish multiple routing
A3161, A3162, A3163
D.I. to accomplish multiple routing
A3163
D.I. to accomplish multiple routing
-125-
BACKWARD TRACEABILITY

DESIGN MODULE

STORE_REFERENCE
THROTTLE_INPUT
TIME_OPS_READ_CLOCK
TRANSLATE_FROM_CARD
TRANSLATE_MESSAGE
TRANSLATE_MESSAGE
TRANSLATE_MESSAGE
TRANSLATE_MESSAGE
TRANSLATE_MESSAGE
TRANSLATE_MESSAGE
TRANSLATE_TO_ASCII
TRANSLATE_TO_CARD
TRANSLATE_TO_ITA_2
TRANSMIT_SYNCH_CHARS

REQUIREMENT

A361
I. SENTENCE 2
Implicit requirement
A4236
A4237
A4221
A4234
A421
A4224
A4231
A4232
A4235
A4233
A5412
A5414
A5413
A5411
A3153
A3155
Control of A333
A161
Control of A335
Control of A334
A332
Control of A332
A513
A512
A514
III.B.(1) & (2) & (3)
III.A.(1) & (2)
3.4 Data Structures

The following section illustrates the detailed structure of the site dependent database tables.

3.4.1 Line Table Operations

The table in this section is used by the message switch software to determine how to initialize the serial data channel hardware and the proper protocol to handle the incoming and outgoing messages.
package LINE_TBL_OPS is

-- NAME: LINE_TBL_OPS
-- PURPOSE: Data structures that contain the information describing
-- the characteristics of each line, and the subprograms needed to
-- access the information. ( Note: All accesses must use CONTROL
-- as these data structures are shared. )

type TRANSMISSION_MODE is ( SYNCH_BLK_BY_BLK, SYNCH_CONTINUOUS,
    ASYNCH_NORMAL, ASYNCH_STEPED );
type FORMAT_TYPE is ( ANY, JANAP, ACP, ACP_MOD );
type LEVEL_TYPE is range 5..8;
VALID_LEVEL : array( 1..2 ) of LEVEL_TYPE := ( 5, 8 );
type PHYSICAL LINE is range 0..50;
type LOOP_SPEED is delta 0.1 range 45..9600;
VALID_SPEED : array( 1..12 ) of LOOP_SPEED := ( 45, 50, 56.9, 74.2,
    75, 150, 300, 600, 1200, 2400, 4800, 9600 );
type COMMUNITIES_SERVED is ( R, U, RU, Y, RY, UY, RUY );
type FIRST_LINK is new BOOLEAN;
MAX_NO_RIS_PER_DELIVERY : array ( 0..4 ) of INTEGER := ( 50,
    500, 1, 6, 14 );
type XTS_TYPE is new BOOLEAN;
type SOM_SEQ_TYPE is ( FULL, ABBREV );
subtype LMF is STRING( 1..2 );
subtype MEDIA is STRING( 1..1 );
type MEDIAS is ( 'A', 'T', 'C', 'Z' );
type DIRECTION_TYPE is ( BOTH, INPUT, OUTPUT );
type NO_STOP_BITS is range 1..2;
type SPEC_TERM_TYPE is ( ACCES, INTERSWITCH, TECHNICAL_CONTROL );
type PHYS_IDS;
type PHYS_PTR is access PHYS_IDS;
type PHYS_IDS is
    record
        PHYS LINE: PHYSICAL LINE;
        NEXT: PHYS_PTR := null;
    end record;
type LOGICAL_IDS;
type LOGICAL_PTR is access LOGICAL_IDS;
type LOGICAL_IDS is
    record
        LOG LINE: LOGICAL LINE;
        NEXT: LOGICAL_PTR := null;
    end record;

procedure APPEND_LINE( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
    PHYSICAL LINE );
    -- Add a physical line to the normal list of lines for a logical
    -- line.

-128-
procedure DROP_LINE( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
PHYSICAL_LINE );

-- Drop a physical line from the normal list of lines for a logical
-- line.
procedure APPEND_ALT( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
PHYSICAL_LINE );

-- Add a physical line to the alternate list of lines for a logical
-- line.
procedure DROP_ALT ( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
PHYSICAL_LINE );

-- Drop a physical line from the alternate list of lines for a
-- logical line.
procedure BEGIN_ALT( LOG_LINE: LOGICAL_LINE );

-- Begin using the alternate list of lines.
procedure END_ALT( LOG_LINE: LOGICAL_LINE );

-- Begin using the normal list of lines.
function IS_ALT( LOG_LINE: LOGICAL_LINE ) return BOOLEAN;

-- Determine if the alternate list of lines is being used.
function VALID_PHYS_LINE( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
PHYSICAL_LINE ) return BOOLEAN;

-- Determine if this physical line is in the list of lines
-- being used by this logical line.
procedure MAKE_AVAILABLE( PHYS_LINE: PHYSICAL_LINE );

-- Mark this line as available for use.
procedure NOT_AVAILABLE( PHYS_LINE: PHYSICAL_LINE );

-- Mark this line as not available for use.
function IS_AVAILABLE( LOG_LINE: LOGICAL_LINE ) return BOOLEAN;

-- Determine if this line is available for use.
procedure SET_LOGICAL( LOG_LINE: LOGICAL_LINE; FORMAT: FORMAT_TYPE;
PREFERRED_LMF: LMF; LEVEL: LEVEL_TYPE;
COMMUNITY: COMMUNITIES_SERVED; XTS: XTS_TYPE; MAX_RIS: INTEGER;
CHARACTER_SET: CHARACTER_SET_TYPE; SPEC_TERM: SPEC_TERM_TYPE );

-- Set the descriptive characteristics of a logical line
function READ_FORMAT( LOG_LINE: LOGICAL_LINE ) return FORMAT_TYPE;

-- Provide the format for this logical line.
function READ_PREFERRED_LMF( LOG_LINE: LOGICAL_LINE ) return LMF;

-- Provide the preferred line media format for this logical line.
function READ_LEVEL( LOG_LINE: LOGICAL_LINE ) return LEVEL_TYPE;

-- Provide the encoding level for this logical line.
function READ_COMMUNITY( LOG_LINE: LOGICAL_LINE ) return
COMMUNITIES_SERVED;

-- Provide the communities served by this logical line.
function READ_XTS( LOG_LINE: LOGICAL_LINE ) return XTS_TYPE;

-- Provide the XTS of this logical line.
function READ_MAX_RIS( LOG_LINE: LOGICAL_LINE ) return INTEGER;

-- Provide maximum number of routing indicators permitted in a
-- message on this logical line.
function READ_CHARACTER_SET( LOG_LINE: LOGICAL_LINE ) return
CHARACTER_SET_TYPE;

-- Provide the character set used on this logical line.
LINE_TBL_OPS

function READ_SPEC_TERM( LOG_LINE: LOGICAL_LINE ) return SPEC_TERM_TYPE;
-- Provide the SPEC_TERM of this logical line.
procedure SET_PHYSICAL( PHYS_LINE: PHYSICAL_LINE; CHNL_MODE:
CHANNEL_MODE; XMIT_MODE: TRANSMISSION_MODE; DIRECTION:
DIRECTION_TYPE; SECURITY: SECURITY_CLASSIFICATION; DESIGNATOR:
CHANNEL_DES; SOM_SEQ: SOM_SEQ_TYPE; STOP_BITS:
NO_STOP_BITS );
-- Set the descriptive characteristics of this physical line.
procedure SET_OCSN( PHYS_LINE: PHYSICAL_LINE; OCSN: CSN );
-- Set the OCSN for this physical line.
procedure INCREMENT_OCSN( PHYS_LINE: PHYSICAL_LINE );
-- Increment the OCSN for this physical line.
procedure SET_ECSN( PHYS_LINE: PHYSICAL_LINE; ECSN: CSN );
-- Set the ECSN for this physical line.
procedure SET_RCSN( PHYS_LINE: PHYSICAL_LINE; RCSN: CSN );
-- Set the RCSN for this physical line.
procedure INCREMENT_OCSN( PHYS_LINE: PHYSICAL_LINE; OCSN: CSN );
-- Increment the OCSN for this physical line.
procedure INCREMENT_ECSN( PHYS_LINE: PHYSICAL_LINE; ECSN: CSN );
-- Increment the ECSN for this physical line.
procedure INCREMENT_RCSN( PHYS_LINE: PHYSICAL_LINE; RCSN: CSN );
-- Increment the RCSN for this physical line.
function READ_CHANNEL_MODE( PHYS_LINE: PHYSICAL_LINE ) return
CHANNEL_MODE;
-- Provide the channel mode of this physical line.
function READ_SECURITY_CLASSIFICATION( PHYS_LINE: PHYSICAL_LINE ) return SECURITY_CLASSIFICATION;
-- Provide security classification of this physical line.
function READ_TRANSMISSION_MODE( PHYS_LINE: PHYSICAL_LINE ) return
TRANSMISSION_MODE;
-- Provide transmission mode of this physical line.
function READ_DIRECTION( PHYS_LINE: PHYSICAL_LINE ) return
DIRECTION_TYPE;
-- Provide transmission direction of this physical line.
function READ_DESIGNATOR( PHYS_LINE: PHYSICAL_LINE ) return
CHANNEL_DES;
-- Provide channel designator of this physical line.
function READ_SOM_SEQ( PHYS_LINE: PHYSICAL_LINE ) return SOM_SEQ_TYPE;
-- Provide the start of message sequence for this physical line.
function READ_STOP_BITS( PHYS_LINE: PHYSICAL_LINE ) return NO_STOP_BITS;
-- Provide the number of stop bits used on this physical line.
function READ_OCSN( PHYS_LINE: PHYSICAL_LINE ) return CSN;
-- Provide current value of OCSN for this physical line.
function READ_ECSN( PHYS_LINE: PHYSICAL_LINE ) return CSN;
-- Provide current value of ECSN for this physical line.
function READ_RCSN( PHYS_LINE: PHYSICAL_LINE ) return CSN;
-- Provide current value of RCSN for this physical line.
procedure SET_CURRENT_PRECEDENCE( PHYS_LINE: PHYSICAL_LINE;
CURRENT_PRECEDENCE: PRECEDENCE );
-- Record the precedence of the message passed to this physical
-- line for transmission.
function READ_CURRENT_PRECEDENCE( PHYS_LINE: PHYSICAL_LINE )
  return PRECEDENCE;
-- Provide current precedence for this physical line.
procedure SET_START_TIME( PHYS_LINE: PHYSICAL_LINE );
-- Record time that a message was passed to this physical line
-- for transmission.
function READ_START_TIME( PHYS_LINE: PHYSICAL_LINE ) return
  DATE_TIME;
-- Provide time that this physical line was last passed a
-- message to transmit.
procedure SET_INTERCEPT( LOG_LINE: LOGICAL_LINE;
  INTERCEPT_PRECEDENCE: PRECEDENCE; INTERCEPT_MEDIA: MEDIA );
-- Set intercept flags for specified criteria.
procedure RESET_INTERCEPT( LOG_LINE: LOGICAL_LINE;
  INTERCEPT_PRECEDENCE: PRECEDENCE; INTERCEPT_MEDIA: MEDIA );
-- Reset intercept flags for specified criteria.
function IS_INTERCEPT( LOG_LINE: LOGICAL_LINE;
  THIS_PRECEDENCE: PRECEDENCE; THIS_MEDIA: MEDIA ) return BOOLEAN;
-- Determine if specified criteria messages are to be intercepted.
procedure SET_KILL( PHYS_LINE: PHYSICAL_LINE );
-- Set kill flag for this physical line.
procedure RESET_KILL( PHYS_LINE: PHYSICAL_LINE );
-- Reset kill flag for this physical line.
function IS_KILL( PHYS_LINE: PHYSICAL_LINE ) return BOOLEAN;
-- Determine if kill flag is set for this physical line.
procedure SET_KILL_ON_EMPTY( PHYS_LINE: PHYSICAL_LINE );
-- Set kill when queue empty flag for this physical line.
procedure RESET_KILL_ON_EMPTY( PHYS_LINE: PHYSICAL_LINE );
-- Reset kill when queue empty flag for this physical line.
function IS_KILL_ON_EMPTY( PHYS_LINE: PHYSICAL_LINE ) return BOOLEAN;
-- Determine if kill when queue empty flag is set for this physical
-- line.

private
type LOGICAL_PORT is
  record
  INTERCEPT: array ( PRECEDENCE, MEDIAS ) of BOOLEAN;
  NAMESAKES: LOGICAL_IDS;
  FORMAT: FORMAT_TYPE := ANY;
  PREFERRED_LMF: LMF;
  LEVEL: LEVEL_TYPE := VALID_LEVEL( 2 );
  ALT_PORTS, PHYS_PORTS: PHYS_PTR := null;
  COMMUNITIES: COMMUNITIES_SERVED := R;
  XTS: XTS_TYPE := FALSE;
  MAX_RIS: INTEGER := MAX_NO_RIS_PER_DELIVERY( 0 );
  CHARACTER_SET: CHAR_SET_TYPE := ANY;
  SPEC_TERM: SPEC_TERM_TYPE := ACCES;
end record;
LINE_TBL_OPS

AVAILABLE: BOOLEAN := FALSE;
ALT_ROUTING: BOOLEAN := FALSE;
end record;
LOGICAL_TABLE: array (LOGICAL_LINE'FIRST .. LOGICAL_LINE'LAST) of LOGICAL_PORT;
type PHYSICAL_PORT(XMIT_MODE: TRANSMISSION_MODE :=
SYNCH_BLK_BY_BLK) is
record
  CHNL_MODE: CHANNEL_MODE;
  CURRENTPRECEDENCE: PRECEDENCE;
  START_TIME: DATE_TIME;
  DIRECTION: DIRECTION_TYPE := BOTH;
  SECURITY: SECURITY_CLASSIFICATION;
  SPEED: LOOP_SPEED;
  KILL, KILL_ONEMPTY,
  AVAILABLE: BOOLEAN := FALSE;
  DESIGNATOR: CHANNEL_DES;
  case XMIT_MODE is
    when ASYNCH_NORMAL | ASYNCH_STEPED =>
      STOP_BITS: NO_STOP_BITS := 1;
      SOM_SEQ: SOM_SEQ_TYPE := FULL;
      ECSN, OCSN, RCSN: CSN := 0;
    when others =>
      null;
  end case;
end record;
PHYSICAL_TABLE: array (PHYSICAL_LINE'FIRST .. PHYSICAL_LINE'LAST) of PHYSICAL_PORT;
end LINE_TBL_OPS;
3.4.2 Routing Indicator Operations

The table in the following section is used by the message processing software to determine how many copies must be made and which logical channels to route them over. These tables are initialized by the switch operator at installation time and may be changed from the operator console by the "Run Switch" software.
with GLOBAL_TYPES; use GLOBAL_TYPES;

package RIOPS is

-- NAME: RIOPS
-- PURPOSE: contains data definitions and procedures required to
decode routing indicators and determine the required
-- routing.
-- PROGRAMMER: Paul Dobbs
-- DATE: May 17, 1982
------------------------------------------------------------------
type PORT ENTRY; type ENTRY_PTR is access PORT_ENTRY;
type PORT ENTRY is
  record
    ALT ROUTING : BOOLEAN := FALSE;
    PORT : LOGICAL_LINE;
    ALT PORT : LOGICAL_LINE;
    NEXT : ENTRY_PTR;
  end record;
type PORT LIST is array(INTEGER range <>) of LOGICAL_LINE;
subtype R STRING is STRING(1..7);
type RI_LIST_ELEM; type RI_PTR is access RI_LIST_ELEM;
type RI LIST_ELEM is
  record
    RI:RI STRING;
    NEXT RI:RI_PTR;
  end record;
subtype RELAY STRING is STRING(1..4);
procedure READ_RI_TABLE(RI:RI STRING; PORTS:out PORT LIST;
SUCCESS:out BOOLEAN);
-- READ THE RI TABLE AND RETURN THE LOGICAL LINE(S) FOR A
-- SPECIFIC RI
-- SUCCESS WILL BE SET FALSE IF THE RI IS NOT FOUND

type CHECK is (NOT_FOUND,FOUND);
function CHECK_RI(RI:RI STRING) return CHECK;
-- CHECK FOR THE PRESENCE OF AN RI IN THE RI TABLE

procedure ADD_RI(RI:RI STRING; PORTS:ENTRY_PTR);
-- ADDS AN RI TO THE APPROPRIATE TABLE
-- THE PORT ENTRY LIST FOR ANY RI OTHER THAN A COLLECTIVE RI
-- WILL ONLY HAVE ONE ENTRY

procedure DELETE_RI(RI:RI STRING);
-- REMOVES AN RI FROM THE TABLES
procedure CHANGE_RI(RI:RI_STRING; LINE:LOGICAL_LINE);
-- SETS THE LOGICAL LINE FOR A NON_COLLECTIVE RI

procedure CHANGE_COLLECTIVE_RI(RI:RI_STRING; OLD_LINE,
    NEW_LINE:LOGICAL_LINE);
-- CHANGES THE PORT ENTRY FOR OLD_LINE TO NEW_LINE
-- IF OLD_LINE = Ø, ADDS A PORT_ENTRY
-- IF NEW_LINE = Ø, DELETES THE PORT_ENTRY FOR THE OLD_LINE

procedure CHANGE_ALT(RI:RI_STRING; ALT_LINE:LOGICAL_LINE;
    START:BOOLEAN:=FALSE);
-- CHANGES THE ALT ROUTING FOR A NON_COLLECTIVE RI
-- IF START = TRUE, BEGINS ALT ROUTING

procedure CHANGE_COLLECTIVE_ALT(RI:RI_STRING; LINE,
    ALT_LINE:LOGICAL_LINE; START:BOOLEAN:=FALSE);
-- CHANGES THE ALT ROUTING FOR ONE PORT ENTRY OF A COLLECTIVE
-- RI
-- IF START = TRUE, BEGINS ALT ROUTING FOR THIS PORT ENTRY

procedure BEGIN_ALT(RI:RI_STRING);
-- STARTS ALT ROUTING FOR A NON-COLLECTIVE RI

procedure BEGIN_COLLECTIVE_ALT(RI:RI_STRING; LINE:LOGICAL_LINE);
-- STARTS ALT ROUTING FOR ONE PORT ENTRY OF A COLLECTIVE RI

procedure END_ALT(RI:RI_STRING);
-- ENDS ALT ROUTING FOR A NON-COLLECTIVE RI

procedure END_COLLECTIVE_ALT(RI:RI_STRING; LINE:LOGICAL_LINE);
-- ENDS ALT ROUTING FOR A PORT ENTRY OF A COLLECTIVE RI

procedure LOAD_RI_TABLE;
-- LOADS THE RI TABLES FOR START UP

procedure SAVE_RI_TABLE;
-- SAVES THE RI_TABLE FOR RESTARTS

procedure INIT_RI_TABLE;
-- SETS UP AN EMPTY RI_TABLE
end RI_OPS;
3.5 Rationale for Hardware/Software Partitioning

Partitioning of the system functions into hardware and software was done with the primary goal of maximizing the system efficiency and a secondary goal of minimizing the amount of hardware. Particular implementations were not selected but rather each function was examined for suitability of implementation in hardware or software.

In general, most of the functions, or operations on objects, were partitioned into software while the objects themselves or object representations were partitioned into hardware. For example, translation of messages was selected for software implementation while the messages themselves are represented by bit-patterns in some form of memory (hardware).

The exceptions to this rule are the actual transmit and receive functions for both synchronous and asynchronous formats. These functions include bit stream operations and serial/parallel conversions. The processes are well defined, time consuming, and mechanical and are well suited to hardware implementation. In addition, there are many integrated circuit devices available which implement these functions with a minimum of additional hardware and software.

Since the development of the Message Switch proceeded directly from the system specification and requirements document, it evolved into a functional system which operated on a group of "objects". As such, it was independent of any preconceived hardware system. This allowed the designers to consider various hardware approaches for implementation. Three basic hardware configurations were examined: 1) a single large central processing machine, 2) a distributed (several smaller machines) and 3) a distributed hierarchical (single medium sized machine with several smaller distributed machines).

The centralized configuration was discarded for two reasons. First, the designers felt that it was unrealistic to expect Ada to be able to handle the large number of tasks which would be required (possible one thousand) in one machine. And, second, it is doubtful that a processor is available with the necessary speed and power to handle a 50 line switch, with the possible exception of a Cray.

The distributed configuration was considered in two variations, a vertical and a horizontal. For example: putting Input in one processor, Processing in another, and Output in a third, as opposed to say, ten inputs, outputs, and message processing in each of five processor units. While there are some attractive prospects to both of these variations, the disadvantages outweighed the potential benefits. The disadvantages included excessive interprocessor communication, forced sharing of tables and
status information across processor boundaries, and the possibility of multiple 16K bit per second channels overloading a single processor.

The designers agreed that the distributed hierarchical system is the best configuration for a message switch application. The logical result of the hardware/software partitioning, as related to message output is illustrated in Figure 3.5-1. The design consists of a combination of the two variations of the distributed approach, and as such, allows the designer to take advantage of the strengths of both. It was decided that Input and Output processing would be maintained on a lower level compared to the other more generalized and less time critical processes. At the same time Input/Output is to be spread horizontally over several processors to minimize the chances of overloading any single processor. Combining Input and Output in the same processor shortens the lines of communication between the two processes and improves the channel Transmit and Receive coordination. An additional advantage is availability, a single point failure would disable no more than eight channels.

Of course there are some disadvantages but they can be minimized. The system will require more processors and interprocessor communication will be needed, causing an increased number of interconnections.

Because of the reduced real time constraints of a distributed system, the first disadvantage is minimized because smaller and less expensive processors can be utilized. There is certainly increased overhead because of interprocessor communication, but this can be minimized by judicious partitioning of the functions. The other disadvantage of interprocessor communications, which is the additional hardware interconnection, can be minimized by using a medium speed serial data bus instead of a parallel bus.

Because it was not known how much support would be provided by the Ada run time package, the configuration shown in Figures 3.5-2 and 3.5-3 were developed. This method would not depend on Ada run time support, but unfortunately, becomes quite machine dependent and reduces transportability. Additional logic was added to handle serial communication protocol and to provide low level support functions that were no longer available in common memory.
Figure 3.5-1 Logical Result of Hardware/Software Partitioning
4. Detailed Hardware Design

4.1 General Configuration

Figure 4.1-1 illustrates the switch system design, consisting of a generalized computing system that is optimized for real-time communications processing. In normal operation incoming messages are received by the Line Termination Units (LTUs), which handle the line level protocol functions. The messages are assembled into segments for transmission to the active main processor. At the main processor the segments are logged, saved on reference storage, translated and routed. After routing, the messages are transferred to the appropriate LTU(s) for final validation before transmission to the appropriate stations.

Start-up of the system takes place by operator action and includes loading of the operational programs from nonvolatile media. Database tables may be loaded from nonvolatile media by operator entry. Operational programs for the LTUs are down-loaded via the serial data bus. All of the processors contain a "bootstrap" and diagnostic program in Read Only Memory.

The main processor may be any general purpose machine with the necessary speed and I/O capability. Some special adaptation will be required to facilitate interprocessor monitoring and switchover. Magnetic tape drives are included as mass storage peripherals for sequential storage applications such as journals and intercepted messages. The disk drives are intended to be used as main program storage for start-up, mass random access storage for intransit messages, reference storage and journals. The specification requires that intransit storage be of sufficient size to contain 2500 average length messages for a 50 line switch. This is 6,000,000 characters, excluding labels and linkage. Although a memory of this size is possible in direct access Read/Write Memory (such as semiconductor RAM) it is not as cost effective as using large capacity disks. Both disk and tape peripherals are backed-up with a redundant unit to prevent single failures from causing a system outage.

The purpose of the line termination units (LTU) are to provide time critical protocol analysis and oversee the data send/receive function for each of the possible fifty channels of the message switch. Each LTU contains a general purpose microcomputer which is capable of being downloaded with channel dependent software. There also exists a read-only memory (ROM) containing the bootstrap program for start-up of the serial bus and program down-loading, as well as diagnostics and fault detection firmware. The LTUs contain the necessary circuitry for electrical interfacing, controlling, and monitoring crypto and modems associated with a channel.
Figure 4.1-1 MESSAGE SWITCH HARDWARE BLOCK DIAGRAM

*REMOTE TERMINAL SBI IN BACKUP PROCESSOR WILL BE DYNAMICALLY RECONFIGURED TO A CONTROLLER WHEN THAT PROCESSOR IS ACTIVATED.*
4.2 Serial Data Base

A redundant data bus is required to prevent single failures from rendering the system inoperative. Physical constraints associated with a redundant bus virtually eliminate a parallel bus from consideration. Primarily, the large number of connections required to propagate a parallel bus from circuit board to circuit board, and the associated cabling problems are prohibitive. A serial bus eliminates these problems by requiring that only two cables be connected to each device for data bus access.

The ideal solution in the interconnect area as well as the areas of throughput and error is to use the MIL-STD-1553B serial bus. This selection has the added benefit of reducing risk because of its established technology. The serial bus is used to transfer message segments, control, request, and status information between the Main processor and the LTUs, as well as updating the Status Panel Displays and downloading the operational programs to the LTUs. The 1553B Serial Data Bus operates at a 1 MHz bit rate and is capable of transferring 480 bytes of data plus a 32 byte label in 5.760 microseconds, including overhead and polling time. This is a bus capacity of 81,081 characters (bytes of data) per second. The specification requires a 50 line switch to be capable of handling a maximum of 9,000 characters in any one second period. Therefore, as long as bus overhead stays below 8 times the maximum data handling requirement the serial data bus will be equal to the task. Standard serial bus modules are available commercially and are designed to use DMA capabilities to reduce the required processor interactions.

The interprocessor message routing is determined by the physical port number. LTU1 contains ports in the range 1-8, LTU2 range 9-16, etc. Although it is not necessary that all consecutive ports be implemented or defined, maximum use can be made of the hardware by defining the ports in groups of four. This is because four channels are supported by each interface printed circuit board (PCB).

4.3 Line Termination Unit

Each LTU shown in Figure 4.3-1 is a two or three board set consisting of a CPU Board and one or two Channel Boards. Each channel board terminates four lines (channels), for a maximum of eight per LTU. This flexibility is provided so that increased processing power is available for high speed or complex channel protocol. On the other hand, a larger quantity of low speed channels in a grouping can also be accommodated by adding the extra channel board. The main determining factors are the processor's speed and efficiency in handling the channels.
Each LTU is equipped with its own dual serial bus interface and DMA Controller for transferring data and programs directly to and from an on board dynamic random access memory (RAM). Each channel interface, which consists of serial data, control, clocks, and status lines, has an associated baud rate generator and interrupt lines. Other functional blocks are self explanatory and standard for a microprocessor system.

4.4 Design Features

Several features of this design are worthy of mention. Although it is expected that one main processor is sufficient for operational needs, a second processor is included for backup and monitoring purposes. The serial bus terminals of this processor are initially configured as a remote terminal but are capable of dynamic reconfiguration to the bus controller mode when processor switch-over occurs. Also at the time of processor switch-over the failed processor is forced to relinquish control of the mass storage peripherals to the backup processor.

The dual serial bus is redundant with automatic error detection capability. Each of the two serial busses is capable of handling the full specified message load plus overhead.

The final important feature is the treatment of operator terminals and local equipment. To prevent the proliferation of special purpose interfaces this equipment is interfaced through Line Termination Units in the same manner as normal serial communication channels.
5. Implementation of Selected Function (Output Message)

5.1 Hardware Implementation

In the system design process outlined in the Ada Integrated Methodology the hardware design and implementation are a part of the detailed design phase of system development. As the Ada Capability Study progressed, the output component was selected as the module to be designed in detail and programmed in Ada. For this reason, more attention was given to the design of the LTUs than to the main processor. In the design process, hardware was selected which has performed well in similar communications applications. It is not possible at this time to know for sure whether Ada compilers and an Ada language system will support the hardware selected.

For processing power and flexibility, an 8-bit CPU such as an INTEL 8088 is recommended. This would allow for upgrade to a 16-bit processor such as the 8086, if additional processing power is needed. Such an upgrade would increase the requirements for RAM, ROM, and driver ICs, but would simplify the serial bus interface controller because the bus operates on 16-bit words.

The following sections describe the two printed circuit boards (PCB) of the LTU in more detail.

5.2 LTU Processor PCB

This PCB, illustrated in Figure 5.2-1, comprises the elements that are common to all of the controlled serial channels, such as the serial bus interface, CPU, RAM, and ROM. The only item which is unusual in the context of a general purpose microcomputer PCB is the ID Register. This register is a set of "three-state" bus drivers which are readable by the CPU. The inputs to the drivers will be connected, through the board connector, to either +5 volts or ground. Each Processor PCB board slot will be programmed (hand-wired on the back-plane) with a unique combination of logic so that each processor can determine its own address. This address will be the one that the Serial Bus Controller will use when communicating with each processor. The Processor PCB also contains the bus driver and receivers necessary to communicate with one or two channel PCBs.

Preliminary estimates of parts count and area indicate that the ICs required will fit on a standard 77 square inch PCB, with sufficient margin to allow for the processor upgrade mentioned earlier.

5.3 LTU Channel PCB

This PCB, as illustrated in Figure 5.3-1, contains the circuitry necessary to interface to the serial channels and
the associated channel equipment (crypto and/or modems), as well as baud rate generator, and the processor bus interface circuits.

Circuitry for four serial channels is provided on each PCB. The serial channel is terminated with a programmable USART which has the capability to transmit and receive both synchronous and asynchronous data. A Register is provided to latch control outputs for crypto and modems, as well as a register to receive status. All of these are provided with electrical interfaces to perform level shifting and isolate the external lines from the logic circuits.

The Baud Rate Generator is programmable and can provide different clock rates for any of up to four asynchronous channels.
5.4 **Software Implementation**

The software implementation for this project was not complete due to the allowed time and budget. Since the message output section was coded in Ada, the intended switch initialization and operation is oriented toward that portion of the switch.

5.4.1 **Switch Operation**

The following sequence, although not implemented, is intended to show how the message output tasks are created and become ready to output actual messages through the hardware. A key feature of this design is the dynamic allocation and deallocation of output tasks in order to support database changes input by the operator while the switch is in operation.

**Power Up** Link protocol and bootstrap are stored in ROM in both processors, so the bus is capable of running on power up. However, no "output message" or "send sync/async" tasks are yet defined until the database is loaded.

**Switch Initialization** Part of the job of the switch initialization routine is to run a short diagnostic to determine the number of operational remote links. This process would thus determine the number of physical ports available in the system and print a copy on the printer. The configuration is determined by hard wiring of the connectors into which the link protocol and port printed circuit boards are inserted.

**Data Base for Output Message** The site dependent parameters (database) are entered interactively from the keyboard of a video display unit (VDU) or from a pre-prepared magnetic media such as tape or disk under the control of the "run switch" module.

The VDU command "define physical port" will prompt the user for the port number (or range of numbers) to be defined. In addition, information such as channel mode, transmission rate (baud), security level, etc. will be entered as requested. At the conclusion of each port definition sequence, the prompt: "Is this information correct (Y or N)?" will appear. If not correct the sequence will be repeated until the information is correct, at which time the following sequence will occur:

1. Run switch will lock out users of the "line table", update the necessary line table information, and unlock the table in the main processor. Line table information needed by the remote processor will be transmitted over the data link to the remote processor corresponding to the physical
2. Run switch uses the TASKS package to create a new output message task and initialize the task by issuing the physical port number. The TASKS package consists of an array of records of access types (task entry point for various output message routines) indexed by physical port number.

3. Run switch also uses the TASKS package to create and initialize the Send task (Mode I, II and IV, and V) and generate and receive control character tasks, if required, (depends on channel mode). This process is communicated over the interprocessor data link to the remote processor corresponding to the physical port being changed.

After the port is placed into active service, the channel is then ready to output message data.

An example of the calls from the "Run Switch" to the "Tasks" package in order to initialize physical line 1 for mode V operation is as follows:

TABLE(1).OUTPUT:= new OUTPUT_MESSAGE; --create the output message task.
TABLE(1).OUTPUT.INIT(1); --tell output message what physical line he is.
TABLE(1).SEND V:= new SEND MODE V; --generate mode V send task.
TABLE(1).SEND-V.INIT(1,UART_ADDRS); --tell send task what UART to use.
TABLE(1).GENCC:= new CONTROL_CHARACTER;--create the cntl char gen tsk
TABLE(1).GENCC.INIT(1); --tell cntl char tsk where to receive chars
TABLE(1).RECVCC:= new CONTROL_CHARACTER;
TABLE(1).RECVCC.INIT(1); -- what line number is this

5.4.2 Ada "Output Message" Code

This code is contained in a separate document due to its length.