AD-A131 476 DOCUMENTATION OF CONCURRENT PROGRAMS(U) GENERAL ELECTRIC CO ARLINGTON VA DATA AND INFORMATION SYSTEMS D A BOEHM-DAVIS ET AL. JUL 83 GEC/DIS/TR-83-388200-7

UNCLASSIFIED N00014-79-C-0595 F/G 9/2 NL

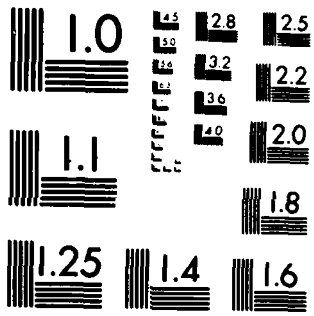
1/1

END

DATE FILMED

12 83

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

GEC/DIS/TR-83-388200-7

AD A 131476

DOCUMENTATION OF CONCURRENT PROGRAMS

DEBORAH A. BOEHM-DAVIS
ANDREW M. FREGLY

Software Management Research & Ada Development
Data & Information Systems
General Electric Company
1755 Jefferson Davis Highway
Arlington, Virginia 22202

DTIC
ELECTE
AUG 1 7 1983

Copy available to DTIC does not permit fully legible reproduction

July 1983

FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

83 08 15 004

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

(12)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-83-388200-7	2. GOVT ACCESSION NO. AD-A131476	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Documentation of Concurrent Programs		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER GEC/DIS/TR-83-388200-7
7. AUTHOR(s) Deborah A. Boehm-Davis & Andrew M. Fregly		8. CONTRACT OR GRANT NUMBER(s) N00014-79-C-0595
9. PERFORMING ORGANIZATION NAME AND ADDRESS Data & Information Systems General Electric Company 1755 Jefferson Davis Hwy., Arlington, VA 22202		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N 42; RR04209; RR0420901; NR 196-160
11. CONTROLLING OFFICE NAME AND ADDRESS Engineering Psychology Group, Code 442 Office of Naval Research Arlington, Virginia 22217		12. REPORT DATE July 1983
		13. NUMBER OF PAGES 59
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) same		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

same

18. SUPPLEMENTARY NOTES

Technical Monitor: Dr. John J. O'Hare

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

Software engineering, Software experiments, Modern programming practices, Software documentation, Petri nets, Resource diagrams, Program design languages (PDLs), Software human factors

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

Previous research on the effectiveness of documentation for sequential programs has suggested that the most effective documentation aids are those which provide clear control-flow information. The current research extends this work into the domain of concurrent processing programs to determine whether the documentation for these programs requires additional information regarding interprocess communications. In this research, programmer performance was examined on a modification task, where modifications were made Cont'd

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

S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Cont'd

to either the data structure or control flow of the program. Taken as a whole, the data suggest that the most appropriate type of documentation for concurrent processing may be different than the most appropriate type of documentation for strictly sequential processing. For modifications to concurrent processing programs, at least for simple programs and simple modifications, it is not crucial whether interprocess communications or control-flow information is highlighted in the documentation format. For more complex problems, it would appear that control-flow information is not necessary, and, in fact, may interfere with making the modification. These data are especially interesting at this time, when PDLs are becoming a de facto standard in the software industry. Further, they suggest that industry may be preparing to adopt, as a standard, a documentation format which will not necessarily provide them with the greatest possible benefit.

GEC/DIS/TR-83-388200-7

DOCUMENTATION OF CONCURRENT PROGRAMS

DEBORAH A. BOEHM-DAVIS
ANDREW M. FREGLY

Software Management Research & Ada Development
Data & Information Systems
General Electric Company
1755 Jefferson Davis Highway
Arlington, Virginia 22202

Submitted to:

Office of Naval Research
Engineering Psychology Group
Arlington, Virginia

Contract: N00014-79-C-0595
Work Unit: NR 196-160

JULY 1983

Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.

TABLE OF CONTENTS

Title	Page
INTRODUCTION	1
METHOD	5
Materials	5
Problems	5
Modifications	5
Documentation Formats	5
Supplemental Materials	6
Design	6
Participants	7
Procedure	7
RESULTS	9
Modification Time	9
Errors	10
Preferences for Documentation Format	11
Experiential Factors	13
DISCUSSION	14
ACKNOWLEDGEMENTS	16
REFERENCES	17
APPENDIX	19
TECHNICAL REPORTS DISTRIBUTION LIST	56

INTRODUCTION

A complete software package always includes documentation. Although its importance is often overlooked, documentation may be the only source of program design information. Major tasks in the software life cycle, such as design, coding, testing and maintenance, are often performed by different individuals. Lientz and Swanson (1979) found that, typically, only about half of a software system's maintenance personnel had been involved in its development. Poor documentation techniques can, therefore, dramatically increase labor costs throughout the labor intensive software life cycle by making both development and maintenance tasks more difficult.

Recent research in this area (Boehm-Davis, Sheppard, & Bailey, 1982; Sheppard, Kruesi, & Bailey, in press; Sheppard, Kruesi, & Curtis, 1981) has been directed toward determining performance on a set of software tasks as a function of the type of documentation. In these studies, programmer performance was examined on comprehension, coding, debugging, and modification tasks as a function of the type of documentation provided. The documentation formats were constructed from the factorial combination of three types of symbology with three types of spatial arrangement. These formats were chosen because they represent the primary dimensions for categorizing the way in which available documentation aids configure the information they present to programmers (Jones, 1979). The three types of symbology in which information was presented consisted of normal English, abbreviated English (such as program design language), and ideograms. The spatial arrangements of the information used in these experiments were sequential, branching, and hierarchical. While each of the four tasks pursued in this research produced slightly different results, there was a general trend towards the superiority of succinct symbology and a branching spatial arrangement in each.

The current research extends the previous investigations on purely sequential programs into the domain of concurrent programming by examining performance on a modification task. Concurrent processing refers to the simultaneous processing of two (or more) portions of the same program. Concurrent processing may be carried out by separate processors in a single computer, separate processors in several computers (distributed processing), or it may be simulated by time-sharing within one processor of a computer. The use of concurrent processing in a program presents a problem in representing those processes in the documentation. Most current documentation formats were designed for sequential program representation, and may not be suitable for the representation of parallel processing. It is especially important to represent this parallelism because, when a task is split into parallel parts, two or more of these paths may need to access the same resources. The documentation should, therefore, provide explicit information on the relationships between processes. If more than one process requires access to the same piece of information, protection of the data may be required to assure its integrity. Thus, programs using concurrent processing must be constructed and documented carefully to ensure orderly access to and sharing of resources.

The investigation of documentation for concurrent processing is especially important since this form of processing is generally considered to be more complex than strictly sequential processing and it is used extensively in embedded computer systems which can monitor and control a number of hardware interfaces simultaneously. Examples of embedded applications include systems for missile guidance, aircraft flight control, and multiplexing of communication channels. The current research will investigate the usefulness of different forms of documentation for this kind of processing.

The task chosen for this experiment was a modification task. Recent reports have asserted that almost 70% of costs associated with software are sustained after the product is delivered. These costs generally are spent in modifying the original program due to changing requirements and correcting errors, and these figures suggest that even small improvements in program maintainability could be translated into substantial time and cost savings. For this reason, it is important to investigate modification performance.

Also, making a modification to an existing program requires several kinds of software skills: an understanding of how the program works; the ability to generate the code required to make changes; and the ability to debug these changes. Thus, it is important to study the modification task; it encompasses more general skills that are required for other software-related tasks.

The previous research suggested that the display of control flow was important in the documentation of sequential programs. While the display of control flow should remain important in documenting concurrent processing, it may be equally important to document the resource sharing among processes. The forms of documentation used in this experiment highlight these different types of information. While all of the documentation formats contain both control-flow and resource-sharing information, the two types of information are differentially emphasized. The first form of documentation is a standard program design language (PDL). The emphasis in PDLs is on the control flow rather than on the resource sharing of a program and the PDLs use abbreviated English in a sequential arrangement. The second form of documentation is a resource diagram, where the emphasis is on providing information about the sharing of resources rather than on control flow. Resource diagrams use abbreviated English in the communication circles and natural language in the process boxes; their spatial arrangement is most similar to the branching arrangement used in our earlier research. The third form of documentation combines both types of information by using Petri nets. Petri nets allow an equal emphasis on control flow and resource sharing. The nodes in the diagram show which resources are required for a task while the constrained language descriptions contain control-flow information. The Petri nets also use a spatial arrangement most similar to our branching arrangement.

The structure of the problem solutions was also manipulated in this research. Different design methodologies currently in use take different approaches to structuring programs. While some methodologies tend to focus on data structures in decomposing problems, others focus on functional decomposition. This may have an impact on the effectiveness of different documentation formats. The research described here examined the effectiveness of different documentation formats using problems which were structured to represent solutions which might be produced by commonly-used design methodologies.

METHOD

Materials

Problems. Three experimental problems and one practice problem were created for use in this experiment. The experimental problems were a message distribution system, an air traffic display, and a text search problem. The practice problem was a message encryption system. The algorithms used to solve the problems were chosen such that they each represented approximately the same overall level of control-flow complexity (as indicated by the McCabe (1976) metric). Each problem was coded in three ways. One version coded the problem such that it had a complex data structure and a simple control flow; one version coded the problem such that it had a simple data structure and a complex control flow; and for one version, the data structure and control flow each carried an intermediate level of complexity.

Modifications. Two modifications were constructed for each problem. One involved a change in the data structure of the problem; the other involved a change in the control flow of the problem. For example, the data-structure modification for the message distribution program (shown in the appendix) required the programmers to change the length of the message. The control-flow modification for the same problem required programmers to change the algorithm so that when a message was entered with a particular message code, all of the readers would receive the message.

Documentation formats. Three documentation formats were created for use in this experiment: Petri nets, resource diagrams, and PDLs. Examples of each of these forms of documentation are shown for all of the problems in the appendix. In the Petri nets (based on ideas in Peterson, 1981), each large box represents a process in the system. The circles represent conditions which must be satisfied before processing can continue. Information listed on the lines between circles represent actions that are being carried out or information that is being passed between processes. In the resource diagrams (based on ideas in Shaw, 1974), the boxes represent processes. The circles represent information which is being passed between processes, and the arrows indicate the direction in which information is being passed. The PDLs use standard notation, except for the use of "send" and "accept" which were the terms used to represent the passing and receiving of communications between and from processes.

Supplemental Materials. Each program was accompanied by four supplemental materials: a program overview, a data dictionary, a program listing, and a listing of the expected output from the program. The program overview contained the requirements, a general description of the program design, and the modification to be performed for each program. The data dictionary contained the variable names, an English description of the variables, and the data type for each variable. The program listing was a paper printout of the FORTRAN code which was identical to the code presented on the CRT screen. The listing of the expected output provided the programmers with the output expected from a correct run of the program; this allowed them to determine where they had gone wrong if their modification to the program did not run correctly.

Design

The experimental design used in this experiment was a 3x3x3x2 split-plot partially confounded design (based on Davies, 1956; Winer, 1971). The within-subject factors were type of documentation (Petri net, resource diagram, PDL), problem (text search, air traffic display, message distribution), and problem structure (complex data structure, complex control flow, intermediate). Type of modification (data structure, control flow) was a between-subjects variable. Each programmer modified three of the twenty-seven possible combinations of documentation, problem, and problem structure; each programmer made three modifications of the same type. For example, a programmer might modify the data-structure version of the text search program using a Petri net, the control-flow version of the air traffic display program using a resource diagram, and the intermediate version of the message distribution program using a PDL. The order in which the programmers were observed under each treatment condition was randomized independently for each programmer.

Participants

The participants in this experiment were 72 professional programmers from four different locations. All were General Electric Company employees. The programmers averaged 8.4 years of programming experience and were familiar with an average of 5.7 programming languages. All of the programmers had previous experience with FORTRAN.

Procedure

Prior to the experiment, the participants were given a one-hour training session in which they were shown examples of each type of documentation format. The experimenter also described the procedure for using the text editor to modify the programs during this session.

Experimental sessions were conducted at CRT terminals on a VAX 11/780. Each participant modified all three of the programs, which were written in FORTRAN-77, using only one of the documentation formats for each. The participants were first asked to enter the changes from the practice problem which was used during the training session to familiarize them with the operation of the experimental system and its editor. Following the practice program, the three experimental programs were presented.

For each program, the participants were asked to first indicate, on the documentation format, the locations in the program where changes needed to be made and then to actually make the modifications using the editor. An interactive data collection system prompted the participants throughout the session. The system recorded each call for an editor command (e.g. ADD, CHANGE, LIST, or DELETE). From these, the overall time to modify and debug the programs was calculated by summing the times from the individual editing sessions; the number of errors made was also calculated. The time required for compiling, linking, and executing the programs was not included in these measures. The programmers were required to continue working on a program until it was completed successfully. The programmers were allowed to take breaks between programs.

Following the experiment, the programmers completed a questionnaire about their previous programming experience. The information requested included number of years of experience and number of programming languages known. The participants were also asked to choose which documentation format they liked most and least, and to rate how much they relied on each documentation format.

RESULTS

Modification Time

The participants required an average of 23 minutes to modify each program. This represents the amount of time studying the program, deciding on the appropriate changes to make the modification, and using the text editor (i.e., the total time spent at the terminal less the time for compiling linking, and executing the program).

MODIFICATION	PROBLEM	DOCUMENTATION FORMAT			TOTAL	
		RESOURCE	PDL	PETRI		
CONTROL FLOW	MESSAGE DISTRIBUTION	19.8	22.1	21.8	21.2	26.0
	AIR TRAFFIC	21.3	25.3	26.8	24.5	
	TEXT SEARCH	28.9	30.1	37.7	32.2	
DATA STRUCTURE	MESSAGE DISTRIBUTION	13.0	12.2	14.9	13.4	20.6
	AIR TRAFFIC	21.0	23.3	23.9	22.7	
	TEXT SEARCH	20.9	22.8	33.1	25.6	
	TOTAL	20.9	22.7	26.4	23.3	

Table 1. Mean Time to Complete Modification Task (in Minutes)

Table 1 shows the mean times for each combination of documentation format, program, and type of modification. An analysis of variance showed that, overall, it took programmers less time to make a data-structure modification (21 minutes) than it did to make a control-flow modification (26 minutes) ($\underline{F}$ (2,64) = 12.64, $p < .001$). This analysis also showed that, overall, resource diagrams required the least amount of time (21 minutes), PDLs required an intermediate amount of time (23 minutes), and Petri nets required the greatest amount of time (26 minutes) ($\underline{F}$(2,95) = 7.31, $p < .001$). A significant interaction was also found between problem and documentation format ($\underline{F}$(4,95) = 2.74, $p < .05$). An examination of the data suggests that for the message distribution and air traffic display problems, there were no significant differences in modification times for resource diagrams versus PDLs or for PDLs versus Petri nets. There does appear to be a significant difference between resource diagrams and Petri nets for both problems, however. For the text search problem, the differences between pairs of documentation formats all appear to be significant.

There were also large differences in the amount of time required to modify the programs (control flow and data structure). The message distribution program required the least amount of time to modify (17 minutes), the air traffic display program required an intermediate amount of time (24 minutes), and the text search program required the greatest amount of time (29 minutes). The analysis of variance supported this conclusion ($F(2,95) = 32.30$, $p < .001$). This pattern of results mirrors the complexity ratings of the programs, as measured by the McCabe metric. While the programs were chosen to be roughly equal in overall complexity, there were some differences among their ratings, which followed the pattern of the time data; the message distribution program had an overall complexity rating of 14, the air traffic display program had an average complexity rating of 15, and the text search program had an average complexity rating of 23.

There was no effect of the structure of the programs (simple control-flow with a complex data structure, intermediate control flow and data structure, or simple data-structure with complex control-flow) on modification time ($F(2,95) < 1$), and it did not interact with any of the other variables.

Errors

For programs that did not compile or run successfully on the first submission, the programmers' editing activities for subsequent submissions were analyzed to determine the number of errors. Table 2 shows the mean number of errors for each combination of documentation format and type of modification. The number of errors was low; in addition, the majority of the errors (63%) were syntax errors rather than semantic errors. (For this analysis, misspellings of variable names, starting a line in the wrong column, and other such errors were categorized as syntax errors.) Due to the low number of semantic errors, no further analysis of these data was carried out.

MODIFICATION	PROBLEM	DOCUMENTATION FORMAT			TOTAL
		RESOURCE	PDL	PETRI	
CONTROL FLOW	MESSAGE DISTRIBUTION	.8	.9	.7	.8
	AIR TRAFFIC	1.2	1.3	.8	1.1
	TEXT SEARCH	1.1	1.4	1.7	1.4
DATA STRUCTURE	MESSAGE DISTRIBUTION	.1	0	.1	.1
	AIR TRAFFIC	.4	1.1	.6	.7
	TEXT SEARCH	.4	.7	.6	.6
	TOTAL	.7	.9	.8	.8

Table 2. Mean Number of Errors

Preferences for Documentation Format

Across the three problems, the programmers received each type of documentation format. On the questionnaire, they were asked to state which documentation format was easiest to use and which was hardest to use. They were also asked to rate how much they relied on each version of documentation format on a seven-point scale (from 0 = not at all to 6 = constantly throughout). Tables 3 and 4 show the number of people choosing each documentation format as easiest or hardest to use as a function of type of modification made. In the control-flow group, two programmers failed to indicate which format had been easiest to use; a third programmer failed to indicate which format had been hardest to use. Overall, seventy-one percent of the programmers chose the PDL format as the easiest to use; 18% chose the Petri net, and 14% chose the resource diagram. The programmers were also asked if they had previously used any of the documentation formats. Eighty-three percent of the programmers making a control-flow modification indicated that they had previously used a PDL; only 53% of the programmers making a data-structure modification had previously used a PDL. Three of the programmers indicated that they had previously used a form of resource diagram; four of the programmers had previously used a form of Petri net. Table 5 shows the mean rating of how much they relied on documentation format for each type of modification. For both types of modifications, the programmers stated they relied most heavily on the PDLs, and less so on the resource diagrams and Petri nets.

MODIFICATION	DOCUMENTATION FORMAT		
	RESOURCE	PDL	PETRI
CONTROL FLOW	5	23	6
DATA STRUCTURE	6	27	3

Table 3. Number of Times Documentation Chosen as Easiest to Use

MODIFICATION	DOCUMENTATION FORMAT		
	RESOURCE	PDL	PETRI
CONTROL FLOW	11	5	19
DATA STRUCTURE	11	5	20

Table 4. Number of Times Documentation Chosen as Hardest to Use

MODIFICATION	DOCUMENTATION FORMAT		
	RESOURCE	PDL	PETRI
CONTROL FLOW	2.4	3.6	2.8
DATA STRUCTURE	2.0	3.3	1.9

Table 5. Mean Ratings of Reliance Upon Each Documentation

Experiential Factors

The participants were asked the number of years they had been programming and the number of programming languages they knew. No correlation was found between years of programming experience and modification time. A low negative correlation ($r = -0.23$, $p < .05$) was found between number of programming languages known and modification time.

DISCUSSION

Substantial differences in completion time were observed among the three types of documentation formats. For both kinds of modification (control flow or data structure), the resource diagrams led to the best performance while Petri nets led to the poorest performance. This suggested that, unlike sequential processes where control-flow information was required, concurrent processing requires information about interprocess communications. Because data structures are often used to pass information between processes, the resource diagrams, which highlight information about communications between processes, also highlight data structures. Both kinds of modifications required locating the particular data structures that needed to be changed; this probably accounts for the fact that it was easier to locate and make modifications when resource diagrams were used. Two things should be noted, though. First, the data suggest that the differences among documentation formats are not very pronounced for all cases; the text search program provided the most striking differences. Second, the modifications used in this experiment were simple and did not require many control-flow changes; this will not always be the case with modifications. This suggests that, at least for simple programs and simple modifications, it is not crucial whether interprocess communications or control-flow information is highlighted in the documentation format. For more complex problems, the longer times required by the Petri nets and PDLs suggest that when modifications are made, detailed control-flow information is not necessary, and, in fact, may interfere with making the modification.

Differences were also observed among the three problem types used in this experiment. The message distribution problem was associated with the shortest times, the text search problem resulted in the longest times, and the air traffic display problem was in-between. This result parallels our past experiences in finding differences across problems. While the programs were roughly equated in terms of a common measure of complexity, they did have slightly different complexity ratings, as measured by the McCabe metric. The amount of time required to make modifications was found to be longer for the problems with a higher complexity metric, suggesting that control- flow complexity may indeed provide a good measure of psychological complexity.

Diversity of experience, in terms of the number of languages used, was a better predictor of performance than years of experience. This result replicates results from our earlier research (Sheppard, Kruesi, & Bailey, in press; Sheppard, Kruesi, & Curtis, 1981; Sheppard, Milliman, & Curtis, 1979) and highlights the importance of ensuring that programmers have an opportunity to gain broad applications experience as part of their professional development.

The participants' choices for the easiest to use documentation format and their previous familiarity with one of the documentation formats lead to an interesting observation. Although, overall, 68% of the programmers had used PDLs before this experiment and 71% of them chose it as the easiest to use, the time required to make the modifications with the PDLs was in between the other documentation formats, for the two types of task modification.

Taken as a whole, the data suggest that the most appropriate type of documentation for concurrent processing (resource diagram) is different than the most appropriate type of documentation for strictly sequential processing (PDL). For modifications to concurrent processing programs, at least for simple programs and simple modifications, it is not crucial whether interprocess communications or control-flow information is highlighted in the documentation format. For more complex problems, it would appear that detailed control-flow information is not necessary, and, in fact, may interfere with making the modification. These data are especially interesting at this time, when PDLs are becoming a de facto standard in the software industry. Further, they suggest that industry may be preparing to adopt, as a standard, a documentation format which will not necessarily provide them with the greatest possible benefit.

ACKNOWLEDGEMENTS

The authors would like to thank Sue Hannan of GE in Lanham, Ron Boehmke of GE in Valley Forge, Bill Carrens and Jackie Pickard of GE in Springfield, and Roger Collins, John Ivers, Dave Morris, and Lou Oliver of GE in Arlington for providing participants and facilities; Dr. John O'Hare for advice; Don Wittig for graphic support in preparing the documentation formats, and Tom McDonald for preparing the supplemental materials and statistical analyses.

REFERENCES

Boehm-Davis, D. A., Sheppard, S. B., & Bailey, J. W. An empirical evaluation of language-tailored PDLs. In _Proceedings of the 26th Annual Meeting of the Human Factors Society_. Santa Monica, CA: The Human Factors Society, 1982, 984-988.

Davies, O. L. _The design and analysis of industrial experiments_. London: Oliver & Boyd, 1956.

Jones, C. A survey of programming design and specification techniques. In _Proceedings of the IEEE Conference on Specifications of Reliable Software_. New York: IEEE, 1979.

Lientz, B. P. & Swanson, E. G. . The use of productivity aids in system development and maintenance (_Technical Report 79-1_). Los Angeles, CA: UCLA, Graduate School of Management, 1979.

McCabe, T. J. A complexity measure. _IEEE Transactions on Software Engineering_, 1976, _2_, 308-320.

Peterson, J. L. _Petri net theory and the modeling of systems_. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Shaw, A.C. _The logical design of operating systems_. Englewood Cliffs, NJ: Prentice-Hall, 1974.

Sheppard, S. B., Kruesi, E., & Bailey, J. W. An empirical evaluation of software documentation formats. In J. Thomas, & M. Schneider (Eds.), _Human Factors in Computer Systems_. Norwood, NJ: Ablex Publishing Corp., in press.

Sheppard, S. B., Kruesi, E., & Curtis, B. The effects of symbology and spatial arrangement on the comprehension of software specifications. In _Proceedings of the Fifth International Conference on Software Engineering_, San Diego, CA: IEEE, 1981.

Sheppard, S. B., Milliman, P., & Curtis, B. Experimental evaluation of on-line program construction (Tech. Rep. TR-79-388100-6). Arlington, VA: General Electric Co., 1979.

Winer, B. J. Statistical principles in experimental design. New York: McGraw-Hill, 1971.

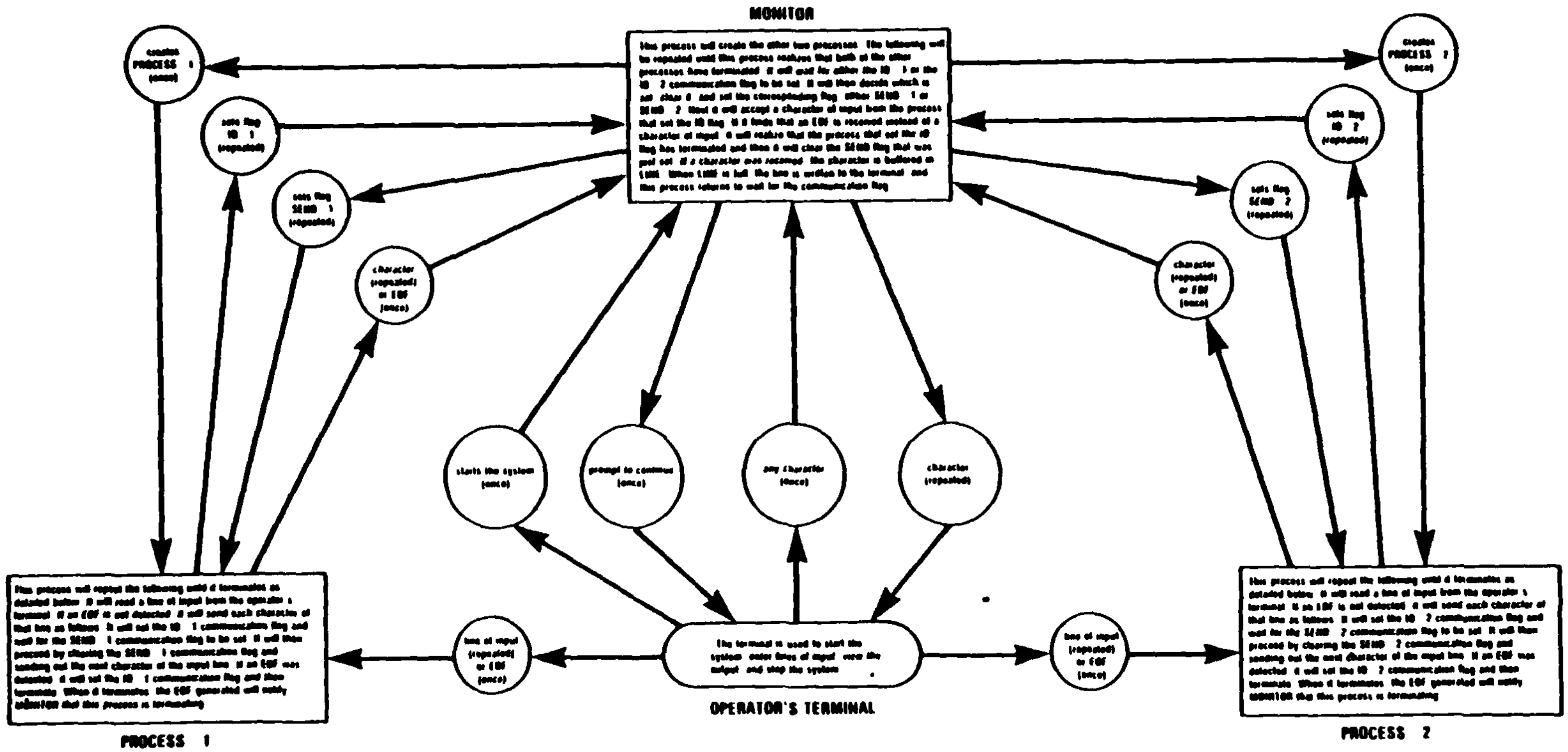
APPENDIX - DOCUMENTATION FORMATS

RESOURCE DIAGRAMS

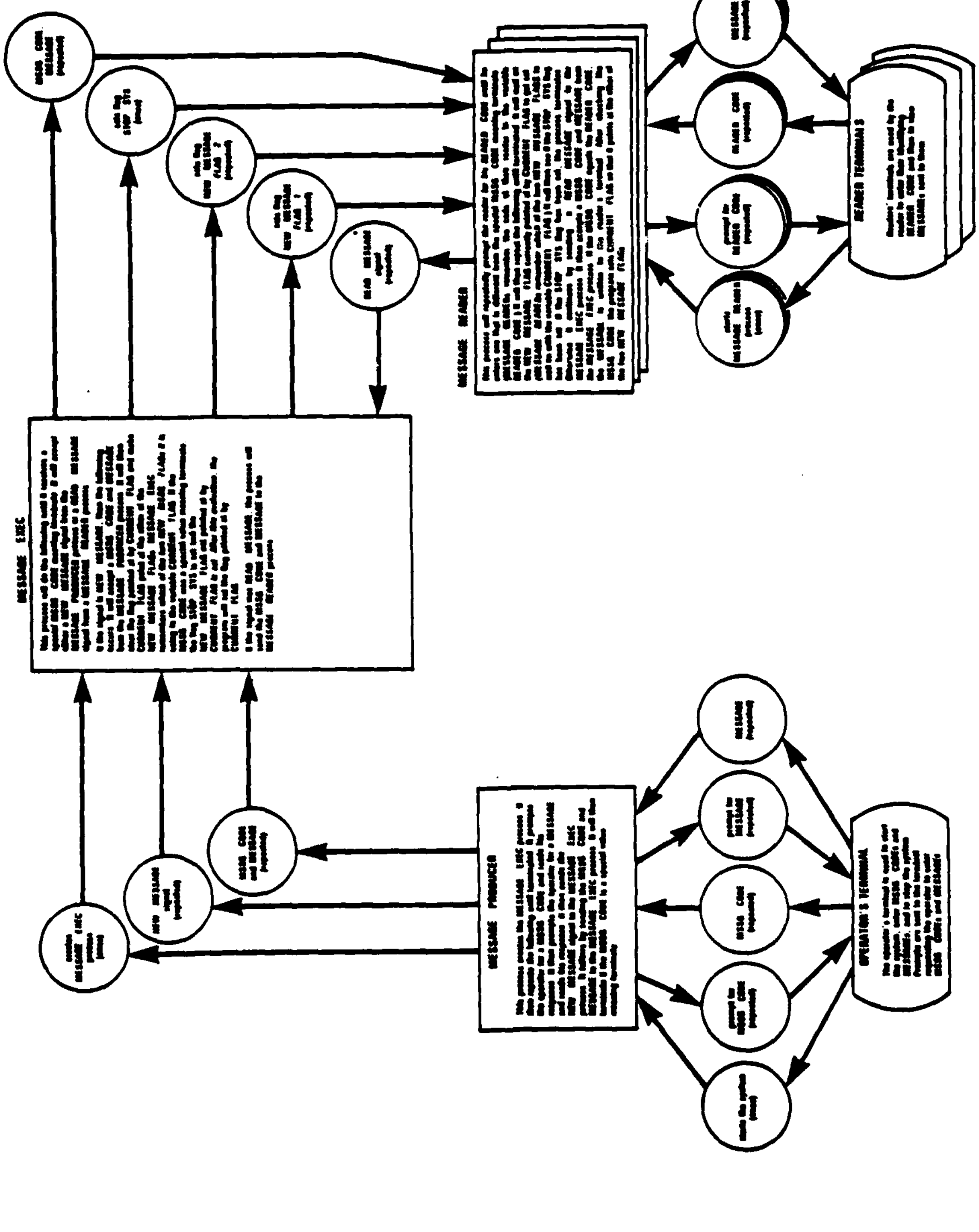
PROGRAM DESIGN LANGUAGES (PDLs)

PETRI NETS

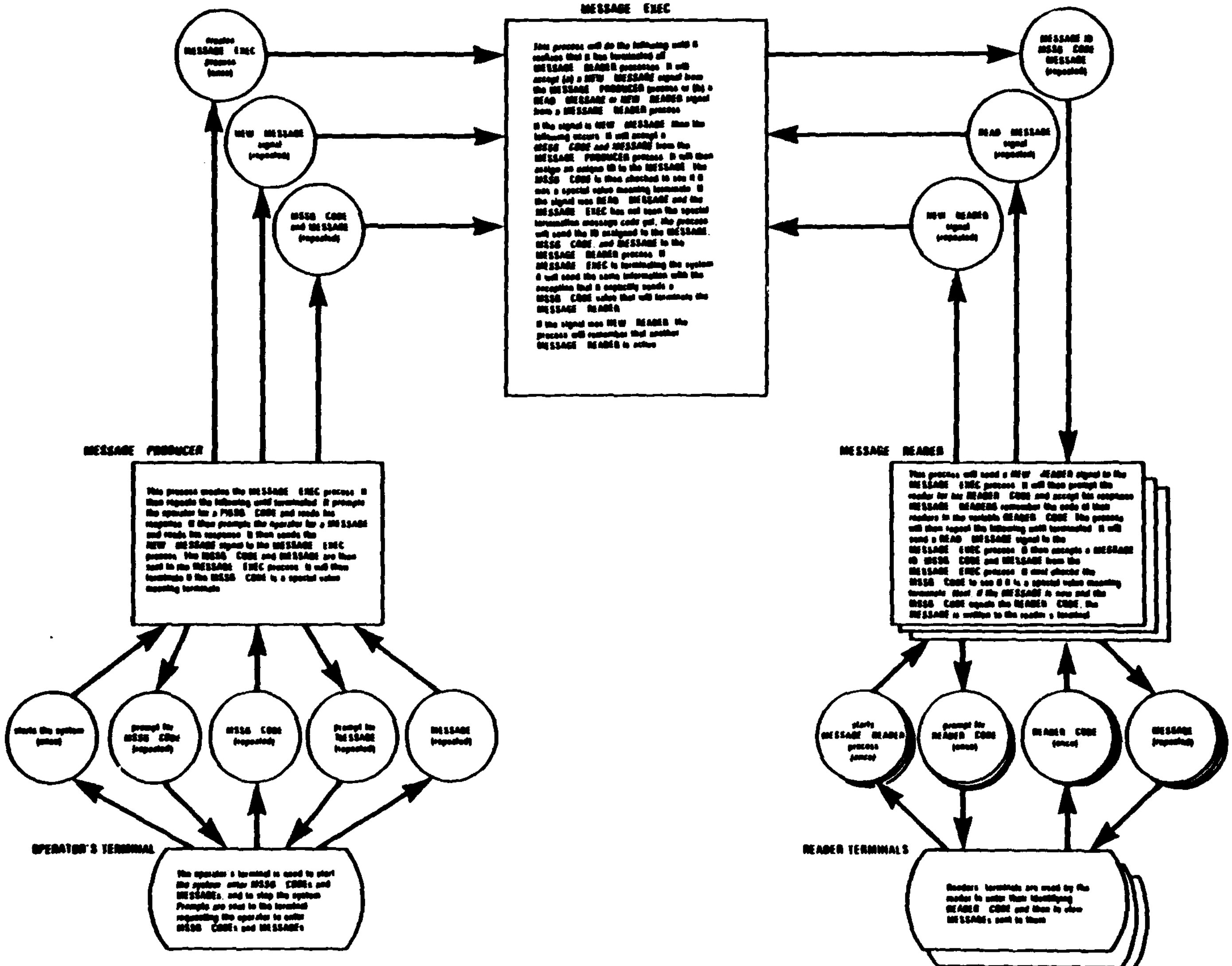
RESOURCE DIAGRAMS


MONITOR
PROCESS 1
PROCESS 2
OPERATOR'S TERMINAL
The terminal is used to start the system enter lines of input view the output and stop the system
creates PROCESS 1 (once)
creates PROCESS 2 (once)
sets flag ID 1 (repeated)
sets flag ID 2 (repeated)
sets flag SEND 1 (repeated)
sets flag SEND 2 (repeated)
character (repeated) or EOF (once)
character (repeated) or EOF (once)
starts the system (once)
prompt to continue (once)
any character (once)
character (repeated)
line of input (repeated) or EOF (once)
line of input (repeated) or EOF (once)

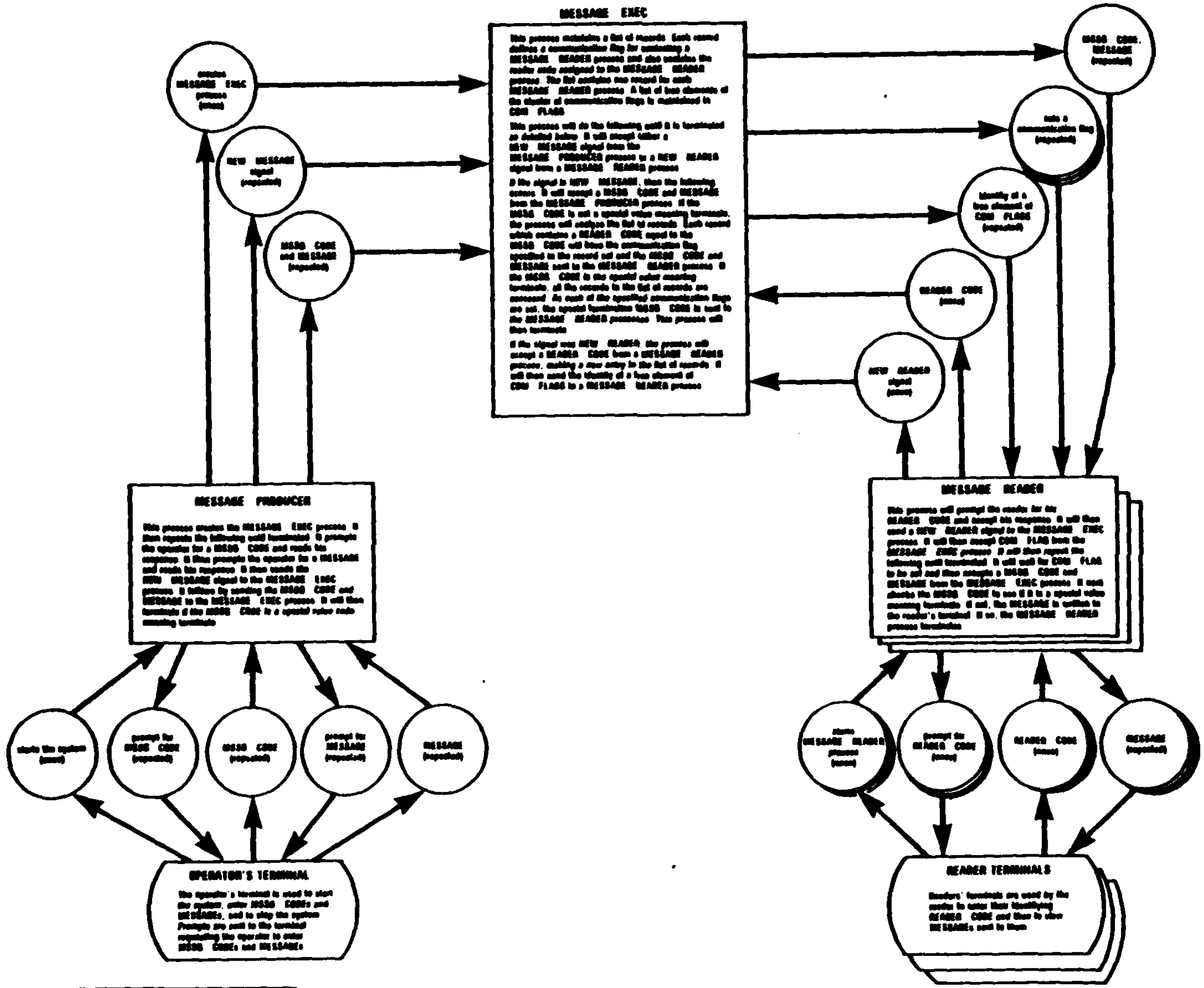
EXAMPLE



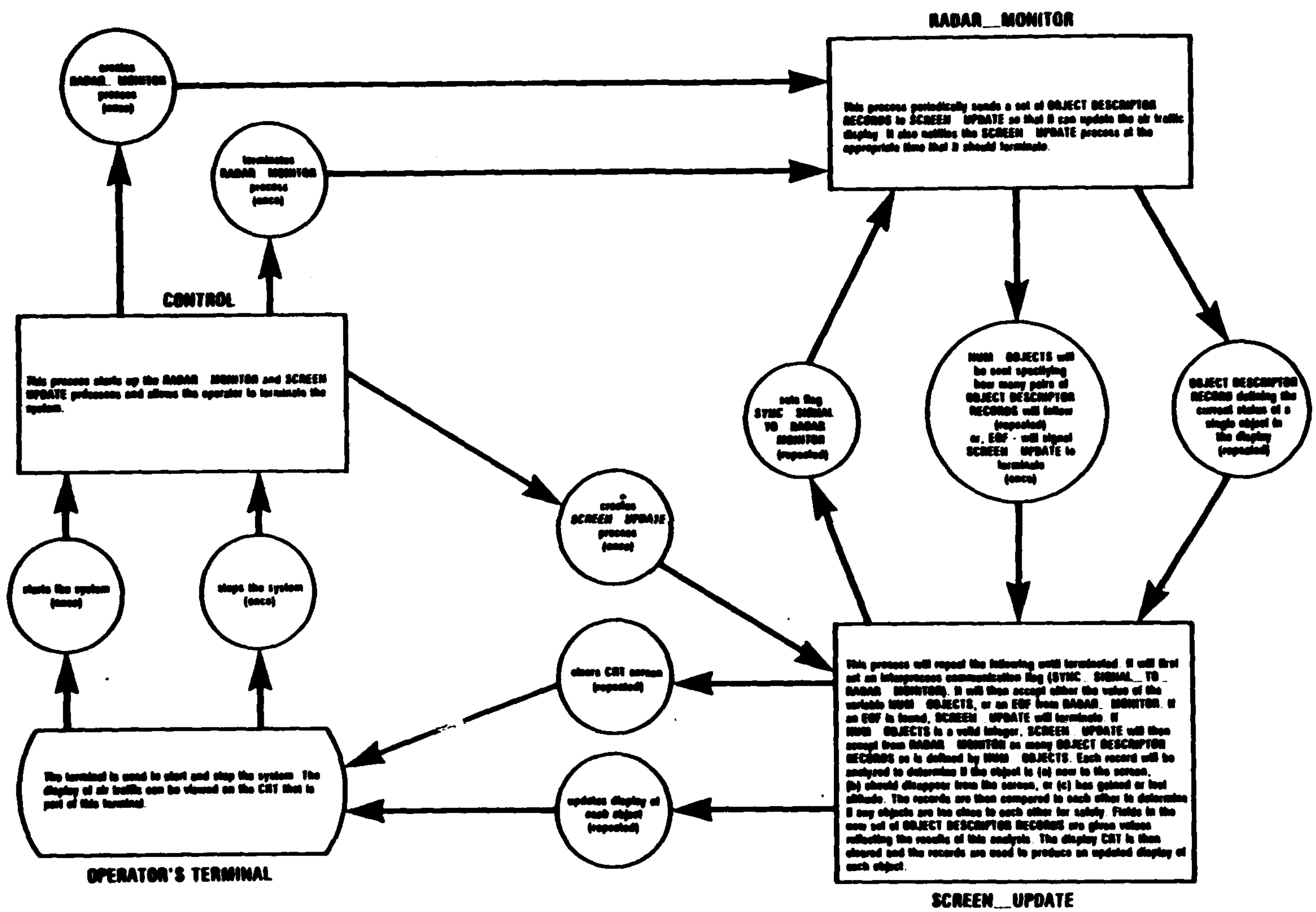
MESSAGE DISTRIBUTION(C)



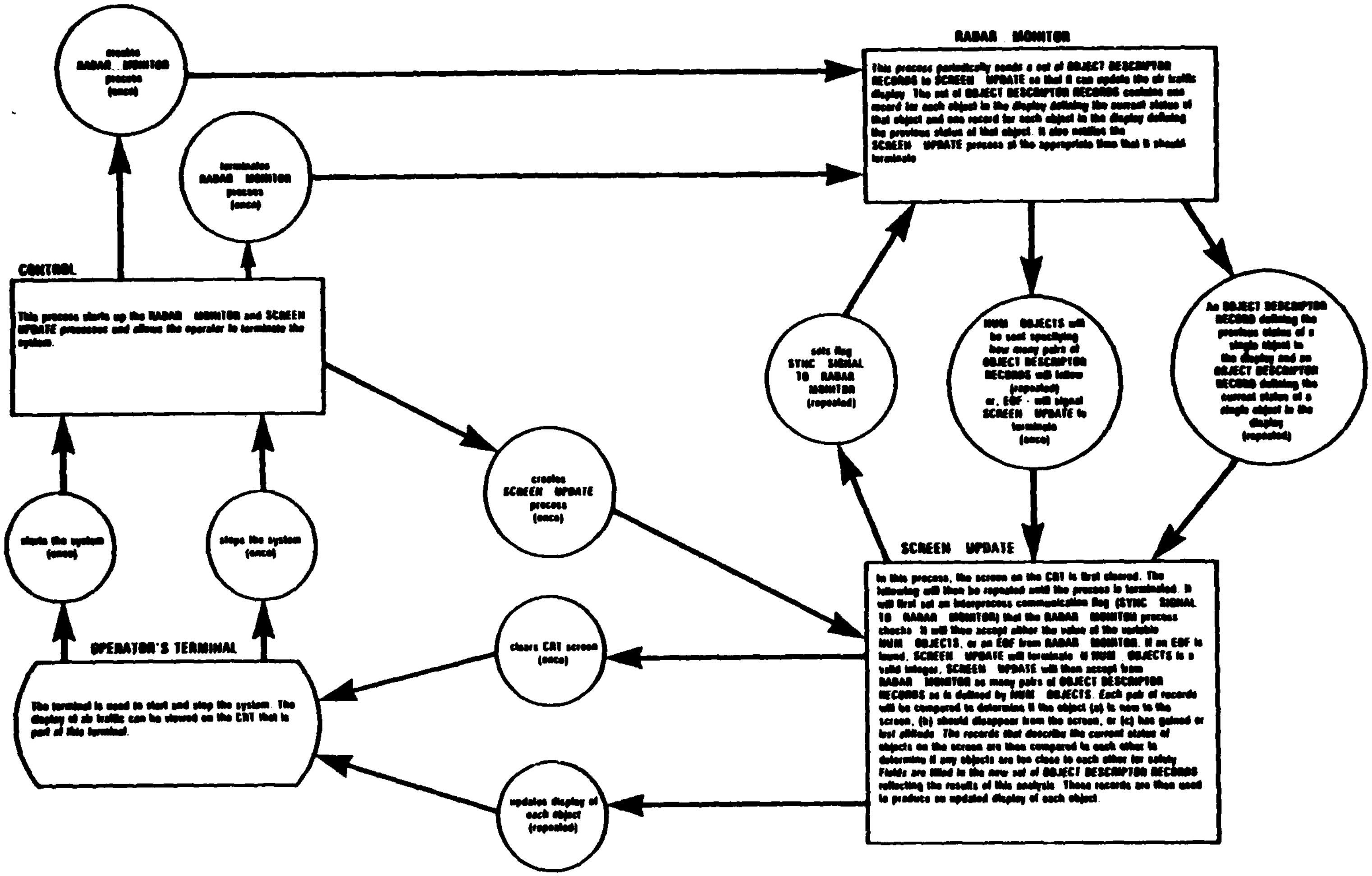
MESSAGE DISTRIBUTION(I)



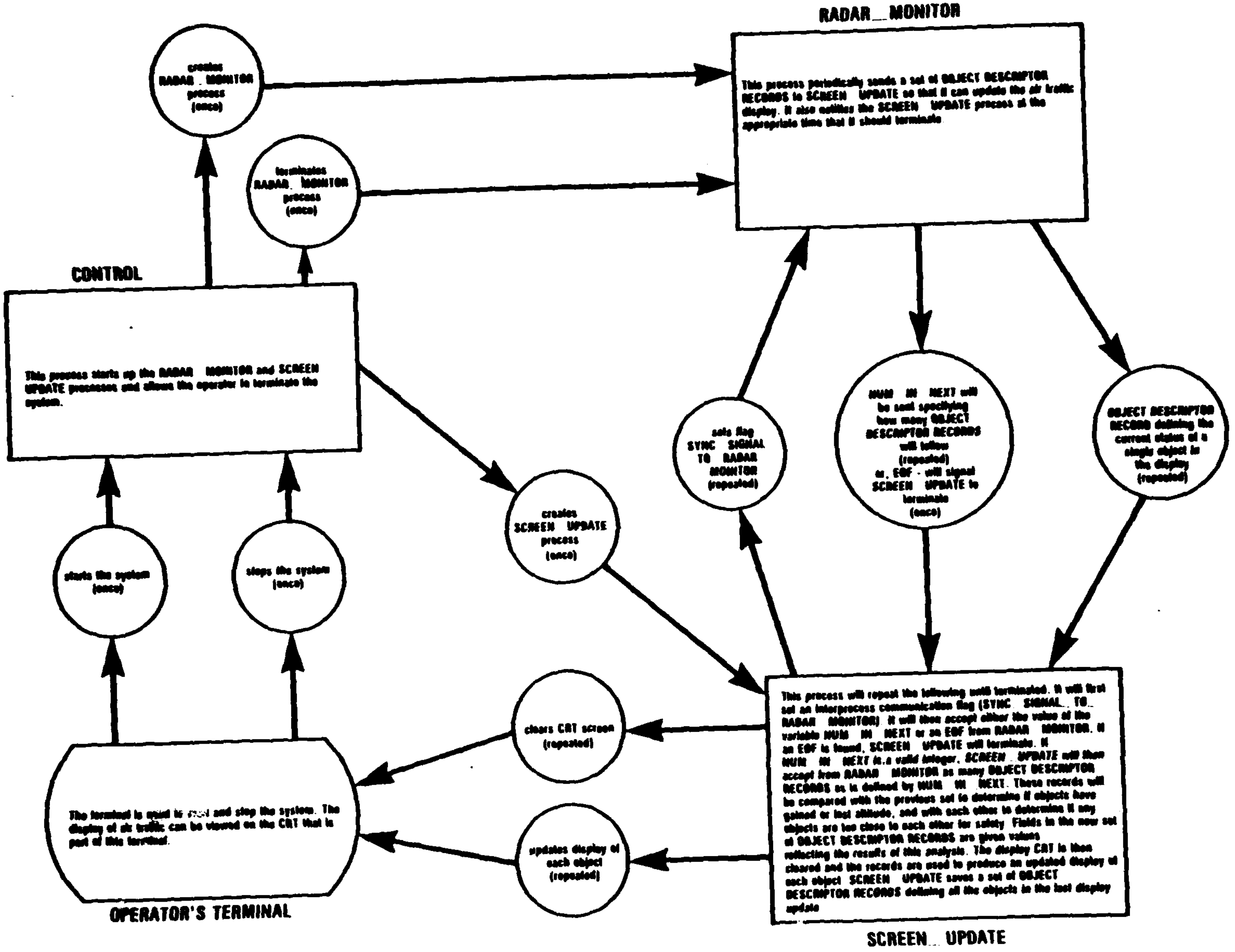
MESSAGE DISTRIBUTION(D)



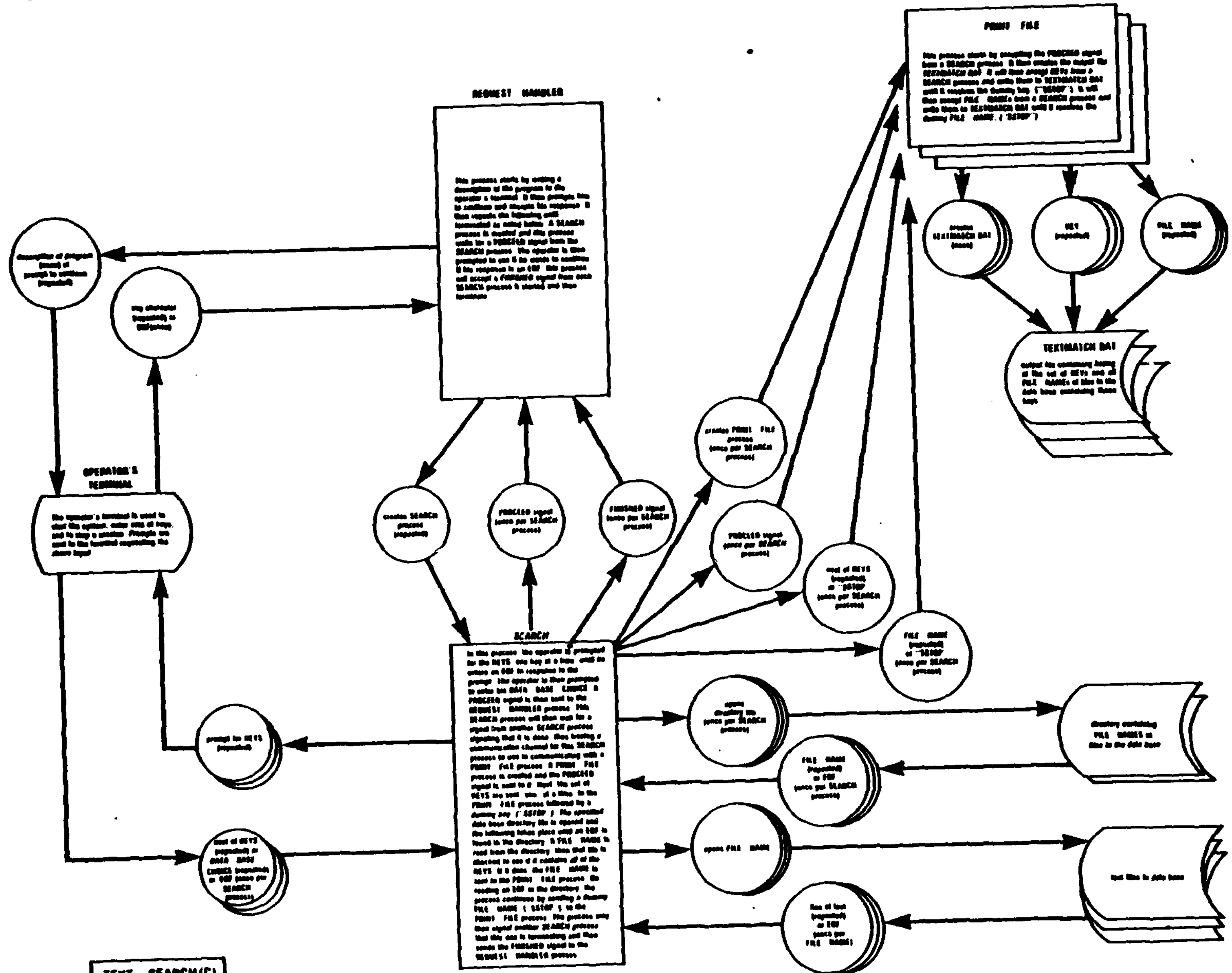
AIR TRAFFIC DISPLAY (C)



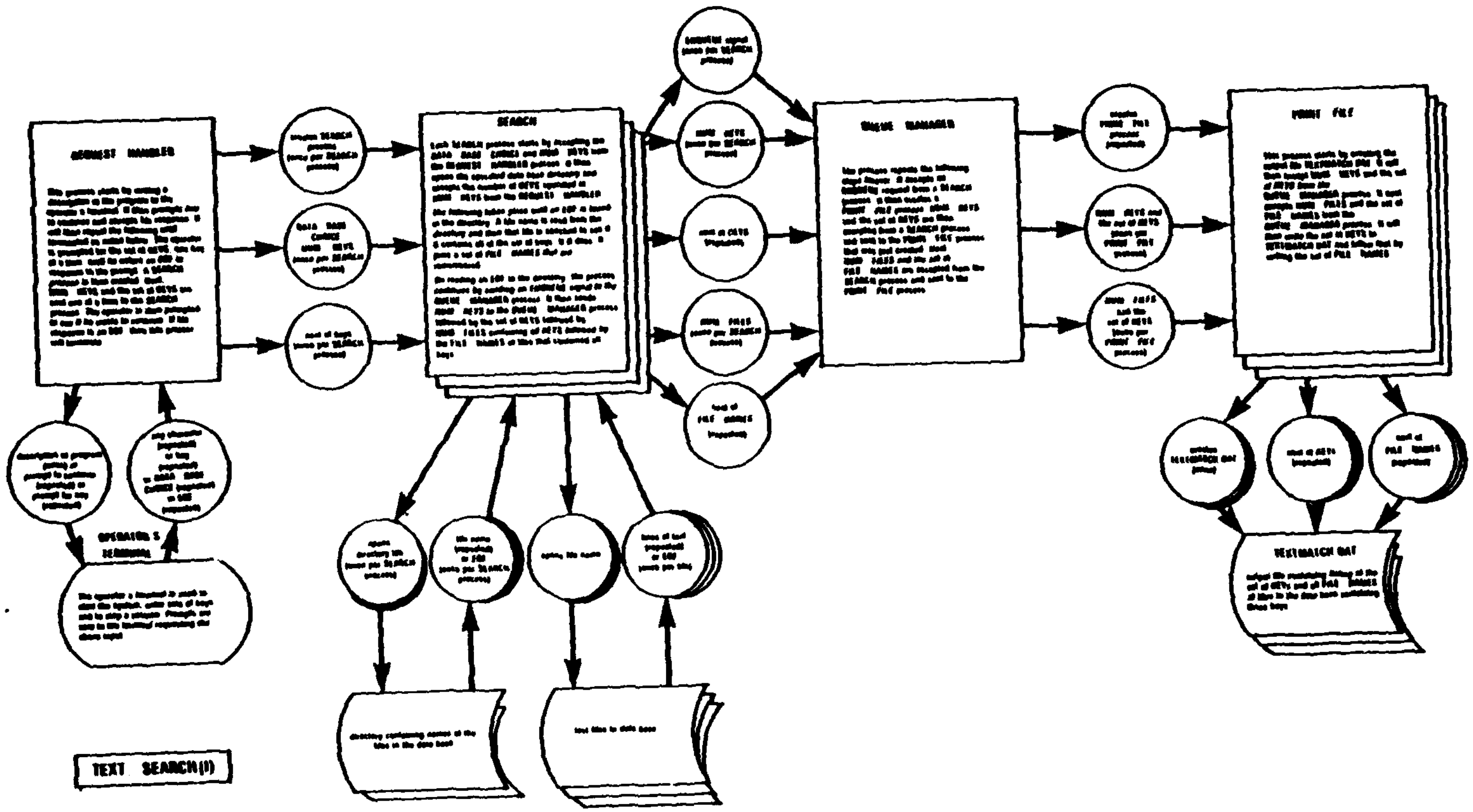
AIR TRAFFIC DISPLAY (I)



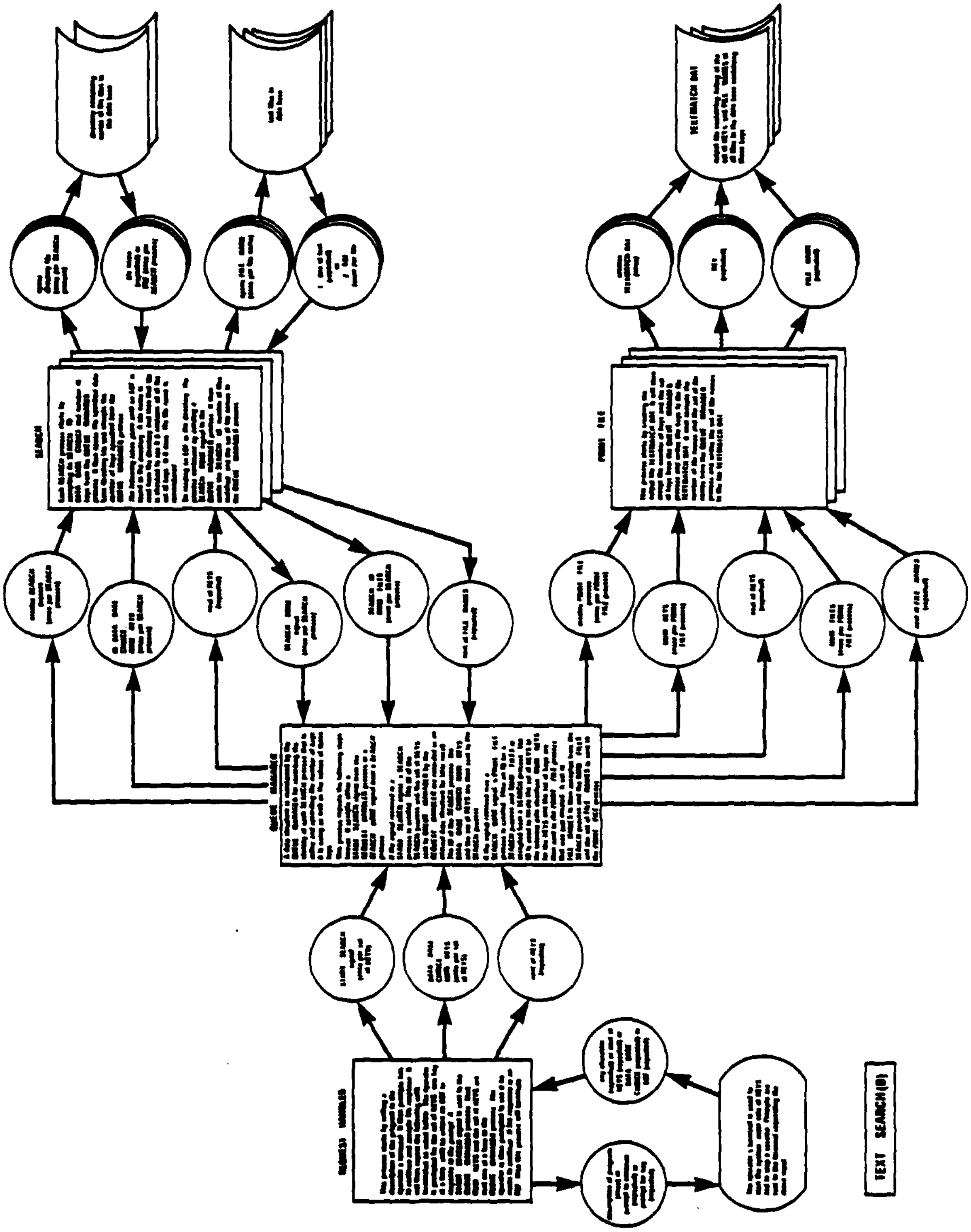
AIR TRAFFIC DISPLAY (D)



TEXT SEARCH(C)



TEXT SEARCH(I)



TEXT SEARCH(8)

PROGRAM DESIGN LANGUAGES (PDLs)

```
program EXAMPLE
declare
  IO_1, SEND_1, IO_2, SEND_2   COMMUNICATION_FLAG

  task PROCESS_1
  declare
    IN_LINE   STRING(1 80)
    I    INTEGER
  begin
    do forever
      read (IN_LINE) from terminal
      if (<end of file read>) then
        exit do
      end if
      do for I = 1 to <location of last non-blank character in IN_LINE>
        SET_FLG(IO_1)
        WAIT_FLG(SEND_1)
        CLEAR_FLG(SEND_1)
        send (IN_LINE(I  I)) to MONITOR
      end do
    end do
    SET_FLG(IO_1)
    <when PROCESS_1 terminates, the end of file this generates will notify
     MONITOR that PROCESS_1 is terminating>
  end PROCESS_1

  task PROCESS_2
  declare
    IN_LINE   STRING(1 80)
    I    INTEGER
  begin
    do forever
      read (IN_LINE) from terminal
      if (<end of file read>) then
        exit do
      end if
      do for I = 1 to <location of last non-blank character in IN_LINE>
        SET_FLG(IO_2)
        WAIT_FLG(SEND_2)
        CLEAR_FLG(SEND_2)
        send (IN_LINE(I  I)) to MONITOR
      end do
    end do
    SET_FLG(IO_2)
    <when PROCESS_2 terminates, the end of file this generates will notify
     MONITOR that PROCESS_2 is terminating>
  end PROCESS_2
  task MONITOR
  declare
    ONE_ID, TWO_ID   INTEGER
    IO_1_READY, IO_2_READY   FLAG_STATUS
    PROC_1_ALIVE, PROC_2_ALIVE   LOGICAL = TRUE
    IN_CHAR   CHARACTER
  begin
    prompt (<operator to continue>) to terminal
    CREATE(PROCESS_1)
    CREATE(PROCESS_2)
    do while (PROC_1_ALIVE or PROC_2_ALIVE)
      WAIT_FOR_OR_OF_FLAGS
      READ_FLG(IO_1, IO_1_READY)
      if (IO_1_READY) then
        CLEAR_FLG(IO_1)
        SET_FLG(SEND_1)
        accept (IN_CHAR) from PROCESS_1
        if (<not eof>) then
          write (IN_CHAR) to terminal
        else
          PROC_1_ALIVE = false
          CLEAR_FLG(SEND_1)
        end if
      end if
      READ_FLG(IO_2, IO_2_READY)
      if (IO_2_READY) then
        CLEAR_FLG(IO_2)
        SET_FLG(SEND_2)
        accept (IN_CHAR) from PROCESS_2
        if (<not eof>) then
          write (IN_CHAR) to terminal
        else
          PROC_2_ALIVE = false
          CLEAR_FLG(SEND_2)
        end if
      end if
    end do
  end MONITOR

begin
  start MONITOR
end
```

EXAMPLE

```
program MESSAGE_DISTRIBUTION
declare
  type SIGNAL is (NEW_MESSAGE, READ_MESSAGE)
  NEW_MESSAGE_FLAG_1, NEW_MESSAGE_FLAG_2, STOP_SYS   COMMUNICATION_FLAG

task MESSAGE_PRODUCER
declare
  EXEC_ID    INTEGER
  MSSG_CODE    STRING(1  5)
  MESSAGE    STRING(1  72)
begin
  CREATE(MESSAGE_EXEC, EXEC_ID)
  do while ({not stopped by operator})
    prompt ({operator for MSSG_CODE}) to terminal
    prompt ({operator for MESSAGE}) to terminal
    send (NEW_MESSAGE) to MESSAGE_EXEC
    send (MSSG_CODE, MESSAGE) to MESSAGE_EXEC
  end do
end MESSAGE_PRODUCER

task MESSAGE_EXEC
declare
  REQUEST    SIGNAL
  CURRENT_FLAG    COMMUNICATION_FLAG = NEW_MESSAGE_FLAG_1
  MSSG_CODE    STRING(1  5)
  MESSAGE    STRING(1  72)
begin
  do while ({value of MSSG_CODE is not special value meaning terminate})
    accept (REQUEST) from MESSAGE_PRODUCER or MESSAGE_READER
    if (REQUEST = NEW_MESSAGE) then
      accept (MSSG_CODE, MESSAGE) from MESSAGE_PRODUCER
      CLEAR_FLG(CURRENT_FLAG)
      CURRENT_FLAG = {alternates between NEW_MESSAGE_FLAG_1 and
        NEW_MESSAGE_FLAG_2}
      if ({value of MSSG_CODE is special value meaning terminate}) then
        SET_FLG(STOP_SYS)
        SET_FLG({whichever of the two NEW_MESSAGE flags is not equal to
          CURRENT_FLAG})
      end if
      SET_FLG(CURRENT_FLAG)
    else if (REQUEST = READ_MESSAGE) then
      send (MSSG_CODE, MESSAGE) to MESSAGE_READER
    end if
  end do
end MESSAGE_EXEC
task MESSAGE_READER
declare
  CURRENT_FLAG    COMMUNICATION_FLAG = NEW_MESSAGE_FLAG_1
  TERMINATED    FLAG_STATUS
  READER_CODE, MSSG_CODE    STRING(1  5)
  MESSAGE    STRING(1  72)
begin
  do forever
    prompt ({operator for his READER_CODE}) to terminal
    if ({READER_CODE  not equal to special termination value} then
      exit do
    else
      write({error message to operator}) to terminal
    end if
  end do
  do while (not TERMINATED)
    WAIT_FLG(CURRENT_FLAG)
    READ_FLG(STOP_SYS, TERMINATED)
    if (not TERMINATED) then
      send (READ_MESSAGE) to MESSAGE_EXEC
      accept (MSSG_CODE, MESSAGE) from MESSAGE_EXEC
      if (MSSG_CODE = READER_CODE) then
        write (MESSAGE) to terminal
      end if
      CURRENT_FLAG = {alternates between NEW_MESSAGE_FLAG_1 and
        NEW_MESSAGE_FLAG_2}
    end if
  end do
end MESSAGE_READER



begin
  start MESSAGE_PRODUCER
  {operating system will allow people to get into the distribution system by
   running the MESSAGE_READER task}
end MESSAGE_DISTRIBUTION
```

MESSAGE DISTRIBUTION (C)

```
program MESSAGE_DISTRIBUTION
declare
  type SIGNAL is (NEW_MESSAGE, READ_MESSAGE, NEW_READER

task MESSAGE_PRODUCER
declare
  EXEC_ID   INTEGER
  MSSG_CODE   STRING(1  5)
  MESSAGE   STRING(1  72)
begin
  CREATE(MESSAGE_EXEC, EXEC_ID)
  do while ({not stopped by operator}
    prompt ({operator for MSSG_CODE}  to terminal
    prompt ({operator for MESSAGE}) to terminal
    send (NEW_MESSAGE) to MESSAGE_EXEC
    send (MSSG_CODE, MESSAGE) to MESSAGE_EXEC
  end do
end MESSAGE_PRODUCER

task MESSAGE_EXEC
declare
  REQUEST   SIGNAL
  ID   INTEGER
  MSSG_CODE   STRING(1  5)
  MESSAGE   STRING(1  72)
begin
  do while ({not all MESSAGE_READERs have been terminated})
    accept (REQUEST) from MESSAGE_PRODUCER or MESSAGE_READER
    if (REQUEST = NEW_MESSAGE) then
      accept (MSSG_CODE, MESSAGE) from MESSAGE_PRODUCER
      ID = {new message identifier number}
      {see if MESSAGE_PRODUCER wants system terminated by checking MSSG_CODE
         value}
    else if (REQUEST = READ_MESSAGE  then
      if ({not terminating MESSAGE_READER processes}  then
        send (ID, MSSG_CODE, MESSAGE) to MESSAGE_READER
      else
        send (ID, {special termination MSSG_CODE}, MESSAGE) to MESSAGE_READER
      end if
    else if (REQUEST = NEW_READER) then
      {remember that another MESSAGE_READER is active}
    end if
  end do
end MESSAGE_EXEC

task MESSAGE_READER
declare
  ID   INTEGER
  READER_CODE, MSSG_CODE   STRING(1  5)
  MESSAGE   STRING(1  72)
begin
  send (NEW_READER) to MESSAGE_EXEC
  prompt ({operator for his READER_CODE}) to terminal
  do while ({termination has not been requested by MESSAGE_EXEC}
    send (READ_MESSAGE) to MESSAGE_EXEC
    accept (ID, MSSG_CODE, MESSAGE) from MESSAGE_EXEC
    {see if termination requested by checking MSSG_CODE value}
    if ({new message and MSSG_CODE = READER_CODE}) then
      write (MESSAGE) to terminal
    end if
  end do
end MESSAGE_READER

begin
  start MESSAGE_PRODUCER
  {operating system will allow people to get into the distribution system by
   running the MESSAGE_READER task}
end MESSAGE_DISTRIBUTION
```

MESSAGE DISTRIBUTION (I)

```
program MESSAGE_DISTRIBUTION
declare
  type SIGNAL is  NEW_MESSAGE, NEW_READER

task MESSAGE_PRODUCER
declare
  MESSAGE   STRING(1..72)
  EXEC_ID   INTEGER
  MSSG_CODE   STRING(1..5)

begin
  CREATE(MESSAGE_EXEC, EXEC_ID)
  do while ((not stopped by operator)
    prompt (<operator for MSSG_CODE>)  to terminal
    prompt (<operator for MESSAGE>)  to terminal
    send (NEW_MESSAGE) to MESSAGE_EXEC
    send (MSSG_CODE, MESSAGE) to MESSAGE_EXEC
  end do
end MESSAGE_PRODUCER

task MESSAGE_EXEC
declare
  COM_FLGS(10)    COMMUNICATION_FLAG
  READER_CODES(10)    STRING(1..5)
  MESSAGE   STRING(1..72)
  REQUEST   SIGNAL
  NUM_READERS   INTEGER
begin
  do forever
   accept (REQUEST) from MESSAGE_PRODUCER or MESSAGE_READER
    if (REQUEST = NEW_MESSAGE) then
      accept (MSSG_CODE, MESSAGE) from MESSAGE_PRODUCER
      if ((MSSG_CODE /= special termination value)  then
        do for I = 1 to NUM_READERS
          if (MSSG_CODE = READER_CODES(I))   then
            SET_FLG(COM_FLGS(I))
            send (MSSG_CODE, MESSAGE) to MESSAGE_READER
          end if
        end do
      else
        do for I = 1 to NUM_READERS
          SET_FLG(COM_FLGS(I))
          send (<special termination MSSG_CODE>, MESSAGE)  to MESSAGE_READER
        end do
        exit do
      end if
    else if (REQUEST = NEW_READER) then
      NUM_READERS = NUM_READERS+1
      accept (READER_CODES(NUM_READERS)) from MESSAGE_READER
      send (<next unused element of COM_FLGS>)  to MESSAGE_READER
    end if
  end do
end MESSAGE_EXEC

task MESSAGE_READER
declare
  COM_FLG   COMMUNICATION_FLAG
  READER_CODE, MESSAGE_CODE   STRING(1..5)
  MESSAGE   STRING(1..72)
begin
  prompt (<operator for his READER_CODE>)  to terminal
  send (NEW_READER) to MESSAGE_EXEC
  send (READER_CODE) to MESSAGE_EXEC
  accept (COM_FLG) from MESSAGE_EXEC
  do forever
    WAIT_ON_FLG(COM_FLG)
    accept (MSSG_CODE, MESSAGE) from MESSAGE_EXEC
    if (MSSG_CODE /= <special termination value>) then
      write (MESSAGE) to terminal
    else
      exit do
    end if
  end do
end MESSAGE_READER



begin
  start MESSAGE_PRODUCER
  <operating system will allow people to get into the distribution system by
   running the MESSAGE_READER task>
end MESSAGE_DISTRIBUTION
```

MESSAGE DISTRIBUTION (D)

```
program AIR_TRAFFIC_DISPLAY
declare
  type OBJECT_DESCRIPTOR_RECORD is record
    ID : INTEGER
    ALTITUDE : INTEGER
    ROW : INTEGER
    COLUMN : INTEGER
    ALTITUDE_CHANGE_INDICATOR   INTEGER
    HAZARD_INDICATOR : INTEGER
    OLD_ALT : INTEGER
  end record
  SYNC_SIGNAL_TO_RADAR_MONITOR : COMMUNICATION_FLAG

task CONTROL
  {starts up the other two processes in the system and allows the operator to
   terminate the system.}
end CONTROL

task RADAR_MONITOR
  {periodically sends a set of OBJECT_DESCRIPTOR_RECORDs to SCREEN_UPDATE so
   that it can update the air traffic display and also notifies the SCREEN_
   UPDATE process at the time it should terminate that it should terminate }
end RADAR_MONITOR

task SCREEN_UPDATE
declare
  OBJECTS(20) : OBJECT_DESCRIPTOR_RECORD
  NUM_OBJECTS   INTEGER
begin
  do forever
    SET_FLG(SYNC_SIGNAL_TO_RADAR_MONITOR)
    accept (NUM_OBJECTS) from RADAR_MONITOR
    if ({end of file found instead of NUM_OBJECTS}) then
      exit do
    end if
    do for I = 1 to NUM_OBJECTS
      accept (OBJECTS(I)) from RADAR_MONITOR
      if ({object disappeared from screen}) then
        {clear image of object from screen}
      end if
    end do
    do for I = 1 to NUM_OBJECTS
      if ({new object on screen}) then
        {initialize record OBJECTS(I)}
      else
        {save indicator of altitude change of object in record OBJECTS(I)}
      end if
    end do
    {check whether any objects are too close to each other, saving an indicator
     of the safety of each object in the OBJECTS records}
    {erase the screen on the display CRT}
    {for each object described by OBJECTS, update the object display on the
     display CRT}
  end do
end SCREEN_UPDATE

begin
  start CONTROL
end
```

AIR TRAFFIC DISPLAY (C)

```
program AIR_TRAFFIC_DISPLAY
declare
  type OBJECT_DESCRIPTOR_RECORD is record
    ID : INTEGER
    ALTITUDE : INTEGER
    ROW : INTEGER
    COLUMN : INTEGER
    ALTITUDE_CHANGE_INDICATOR : INTEGER
    HAZARD_INDICATOR : INTEGER
  end record
  SYNC_SIGNAL_TO_RADAR_MONITOR : COMMUNICATION_FLAG

task CONTROL
  {starts up the other two processes in the system and allows the operator to
   terminate the system.}
end CONTROL

task RADAR_MONITOR
  {periodically sends a set of OBJECT_DESCRIPTOR_RECORDs to SCREEN_UPDATE so
   that it can update the air traffic display - also notifies the SCREEN_
   UPDATE process at the appropriate time that it should terminate}
end RADAR_MONITOR

task SCREEN_UPDATE
declare
  OLD_OBJECT, NEW_OBJECT(20) : OBJECT_DESCRIPTOR_RECORD
  NUM_OBJECTS : INTEGER
begin
  {erase the screen on the display CRT}
  do forever
    SET_FLG(SYNC_SIGNAL_TO_RADAR_MONITOR)
    accept (NUM_OBJECTS) from RADAR_MONITOR
    if ({end of file found instead of NUM_OBJECTS}) then
      exit do
    end if
    do for I = 1 to NUM_OBJECTS
      accept (OLD_OBJECT, NEW_OBJECT(I)) from RADAR_MONITOR
      if ({new object on screen}) then
        {initialize record NEW_OBJECT(I)}
      else if ({object disappeared from screen}) then
        {clear image of object from screen}
      else
        {save indicator of altitude change of object in record NEW_OBJECT(I)}
      end if
    end do
    {check whether any objects are too close to each other, saving an indicator
     of the safety of each object in the NEW_OBJECTs records}
    {for each object by described NEW_OBJECTs, update the object display on the
     display CRT}
  end do
end SCREEN_UPDATE

begin
  start CONTROL
end
```

AIR TRAFFIC DISPLAY (I)

```
program AIR_TRAFFIC_DISPLAY
declare
  type OBJECT_DESCRIPTOR_RECORD is record
    ID : INTEGER
    ALTITUDE : INTEGER
    ROW : INTEGER
    COLUMN : INTEGER
    ALTITUDE_CHANGE_INDICATOR : INTEGER
    HAZARD_INDICATOR : INTEGER
  end record
  SYNC_SIGNAL_TO_RADAR_MONITOR : COMMUNICATION_FLAG

task CONTROL
  <starts up the other two processes in the system and allows the operator to
   terminate the system.>
end CONTROL

task RADAR_MONITOR
  <periodically sends a set of OBJECT_DESCRIPTOR_RECORDs to SCREEN_UPDATE so
   that it can update the air traffic display and also notifies the SCREEN_
   UPDATE process at the time it should terminate that it should terminate.>
end RADAR_MONITOR

task SCREEN_UPDATE
declare
  CURRENT_OBJECTS(20),NEXT_OBJECTS(20) : OBJECT_DESCRIPTOR_RECORD
  NUM_IN_NEXT : INTEGER
begin
  do forever
    SET_FLG(SYNC_SIGNAL_TO_RADAR_MONITOR)
    accept (NUM_IN_NEXT) from RADAR_MONITOR
    if (<end of file found instead of NUM_IN_NEXT>) then
      exit do
    end if
    do for I = 1 to NUM_IN_NEXT
      accept (NEXT_OBJECTS(I)) from RADAR_MONITOR
    end do
    <for each object described by NEXT_OBJECTS, see if the altitude has
     changed compared to the same object described in CURRENT_OBJECTS and
     save indicator of altitude change of object in record NEXT_OBJECT(I)>
    <check whether any objects are too close to each other, saving an indicator
     of the safety of each object in the NEXT_OBJECTs records>
    <erase the screen on the display CRT>
    <for each object described NEXT_OBJECTS, update the object display on the
     display CRT>
    CURRENT_OBJECTS = NEXT_OBJECTS
  end do
end SCREEN_UPDATE

begin
  start CONTROL
end
```

AIR TRAFFIC DISPLAY (D)

```
program TEXT_SEARCH is
declare
  type SIGNAL is (PROCEED, FINISHED, SEARCH_DONE)

task REQUEST_HANDLER
declare
  I, NUM_KEYS, SEARCH_ID  INTEGER
  KEYS(5)   STRING(1  80)
begin
  write (<description of program>) to terminal
  prompt (<operator to continue>) to terminal
  do forever
    CREATE(SEARCH, SEARCH_ID)
    accept (PROCEED) from SEARCH
    prompt (<operator to continue program>) to terminal
    if (<end of file received>) then
      exit do
    end if
  end do
  <for every SEARCH created, accept (FINISHED) from SEARCH>
end REQUEST_HANDLER

task SEARCH
declare
  NUM_KEYS, KEY_LENGTH(5), I, PRINT_ID, DATA_BASE_CHOICE  INTEGER
  KEYS(5)   STRING(1  80)
  FILE_NAME   STRING(1  40)
begin
  NUM_KEYS = 0
  do forever
    prompt (<operator for KEYS(NUM_KEYS+1)>) to terminal
    if (<end of file received>) then
      exit do
    end if
    NUM_KEYS=NUM_KEYS+1
  end do
  prompt (<operator to enter his DATA_BASE_CHOICE>) to terminal
  send (PROCEED) to REQUEST_HANDLER
  <wait for signal from another SEARCH signaling that it is done, this
   freeing a line for this SEARCH to use in communicating with its
   PRINT_FILE process>
  CREATE (PRINT_FILE, PRINT_ID)
  send (PROCEED) to PRINT_FILE
  do for I = 1, NUM_KEYS
    send (KEYS(I)) to PRINT_FILE
    KEY_LENGTH(I) = LAST_CHAR_LOC(KEYS(I))
  end do
  send ("$STOP") to PRINT_FILE
  <open specified data base directory file>
  do forever
    read (FILE_NAME) from directory
    if (<end of file received>) then
      exit do
    end if
    if (ALL_KEYS_IN_FILE(FILE_NAME, NUM_KEYS, KEYS, KEY_LENGTH)) then
      send (FILE_NAME) to PRINT_FILE
    end if
  end do
  send ("$STOP") to PRINT_FILE
  <if necessary, notify next SEARCH that this one is terminating>
  send (FINISHED) to REQUEST_HANDLER
end SEARCH

task PRINT_FILE
declare
  KEYS   STRING(1  80) = null
  FILE_NAME   STRING(1  40) = null
begin
  accept (PROCEED) from SEARCH
  <create output file "TEXT_MATCH.DAT">
  do while (KEY(1  5) /= "$STOP")
    accept (KEY) from SEARCH
    write (KEY) to "TEXT_MATCH.DAT"
  end do
  do while (FILE_NAME(1, 5) /= "$STOP")
    accept (FILE_NAME) from SEARCH
    write (FILE_NAME) to "TEXT_MATCH.DAT"
  end do
end PRINT_FILE

begin
  start REQUEST_HANDLER
end TEXT_SEARCH
```

TEXT SEARCH (C)

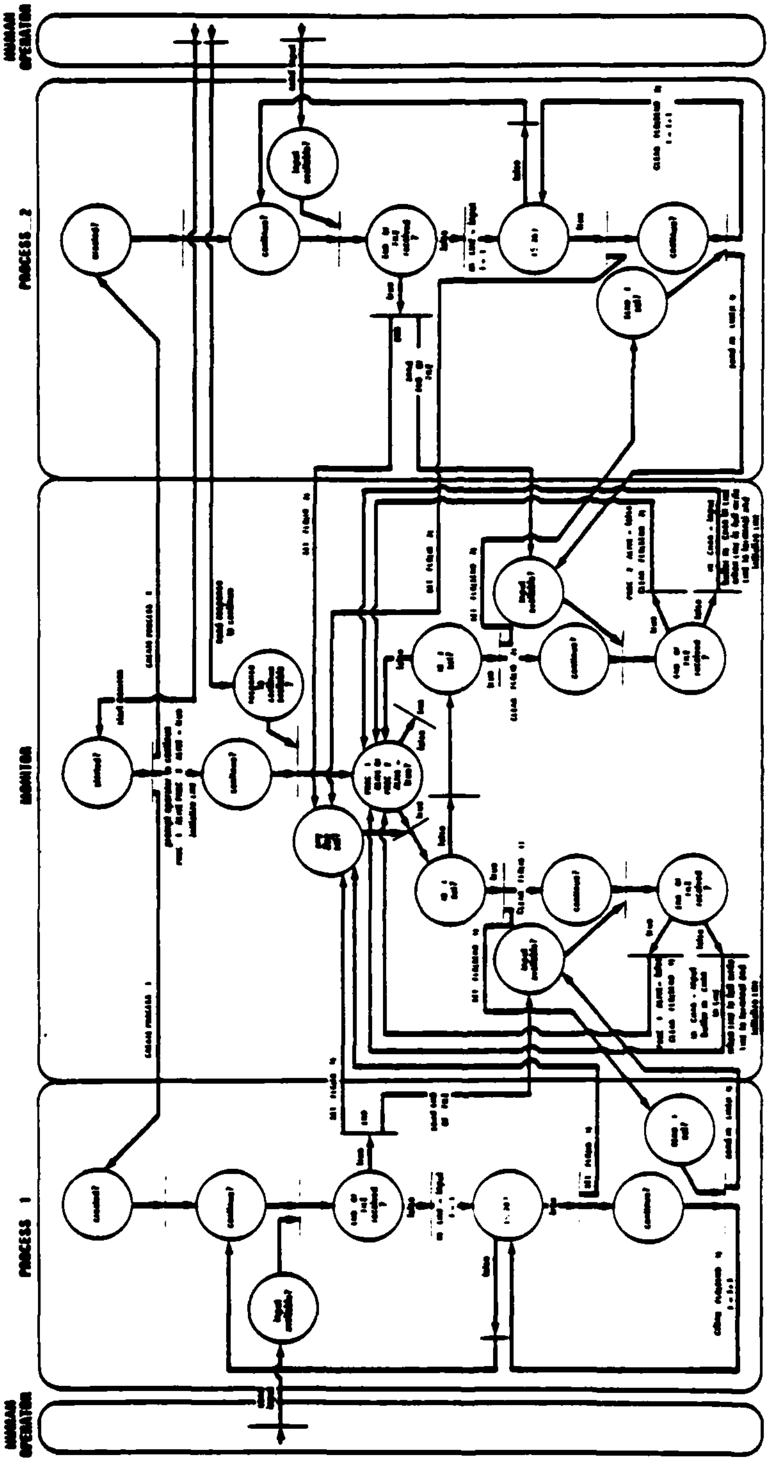
```
program TEXT_SEARCH
declare
  type SIGNAL is (ENQUEUE)

task REQUEST_HANDLER
declare
  I, NUM_KEYS, SEARCH_ID, DATA_BASE_CHOICE   INTEGER
  KEYS(5)   STRING(1  80)
begin
  write (<description of program>  to terminal
  prompt (<operator to continue>) to terminal
  do forever
    NUM_KEYS = 0
    do forever
      prompt (<operator for KEYS(NUM_KEYS+1)>  to terminal
      if (<end of file received>) then
        exit do
      else
        NUM_KEYS=NUM_KEYS+1
      end if
    end do
    prompt (<operator to enter his DATA_BASE_CHOICE>  to terminal
    CREATE(SEARCH, SEARCH_ID
    send (DATA_BASE_CHOICE, NUM_KEYS) to SEARCH
    <send (KEYS(1)  KEYS(NUM_KEYS)) to SEARCH>
    prompt (<operator to continue program>) to terminal
    if (<end of file received>) then
      exit do
    end if
  end do
end REQUEST_HANDLER

task QUEUE_MANAGER
declare
  REQUEST   SIGNAL
  PRINT_ID, NUM_KEYS, NUM_FILES, I, KEY_LENGTH   INTEGER
  KEY, FILE_NAME   STRING(1  80)
begin
  do forever
    accept (REQUEST) from SEARCH
    if (REQUEST = ENQUEUE) then
      CREATE(PRINT_FILE, PRINT_ID)
      <accept NUM_KEYS, and the set of KEYs from a SEARCH process and
       send them to the PRINT_FILE process that was just created>
      <accept NUM_FILES, and the set of FILE_NAMEs from the SEARCH process
       and send them to the PRINT_FILE process>
    end if
  end do
end QUEUE_MANAGER
task SEARCH
declare
  NUM_KEYS, KEY_LENGTH(5), I, NUM_FILES, DATA_BASE_CHOICE   INTEGER
  KEYS(5), FILE_NAME(100)   STRING(1  80)
begin
  NUM_FILES = 0
  accept (DATA_BASE_CHOICE, NUM_KEYS) from REQUEST_HANDLER
  <open specified data base directory file>
  <accept (KEYS(1)  KEYS(NUM_KEYS)) from REQUEST_HANDLER>
  do forever
    read (FILE_NAME(NUM_FILES+1)) from directory
    if (<end of file received>) then
      exit do
    end if
    if (ALL_KEYS_IN_FILE(FILE_NAME, NUM_KEYS, KEYS, (key length)) then
      NUM_FILES=NUM_FILES+1
    end if
  end do
  send (ENQUEUE) to QUEUE_MANAGER
  send (NUM_KEYS) to QUEUE_MANAGER
  <send (KEYS(1))  KEYS(NUM_KEYS)) to QUEUE_MANAGER>
  send (NUM_FILES) to QUEUE_MANAGER
  <send (FILE_NAMES(1)  FILE_NAMES(NUM_FILES)) to QUEUE_MANAGER>

end SEARCH

task PRINT_FILE
declare
  NUM_KEYS, NUM_FILES, I, KEY_LENGTHS(5)   INTEGER
  KEYS(5), FILE_NAME(100)   STRING(1  80)
begin
  <create output file "TEXT_MATCH DAT">
  <accept NUM_KEYS, and the set of KEYS from QUEUE_MANAGER>
  <accept NUM_FILES, and the set of FILE_NAMES from QUEUE_MANAGER>
  <write  KEYS(1)  KEYS(NUM_KEYS) to "TEXT_MATCH DAT">
  <write (FILE_NAME(1)  FILE_NAME(NUM_FILES) to "TEXT_MATCH DAT">
end PRINT_FILE

begin
  start REQUEST_HANDLER
end TEXT_SEARCH
```

TEXT SEARCH (I)

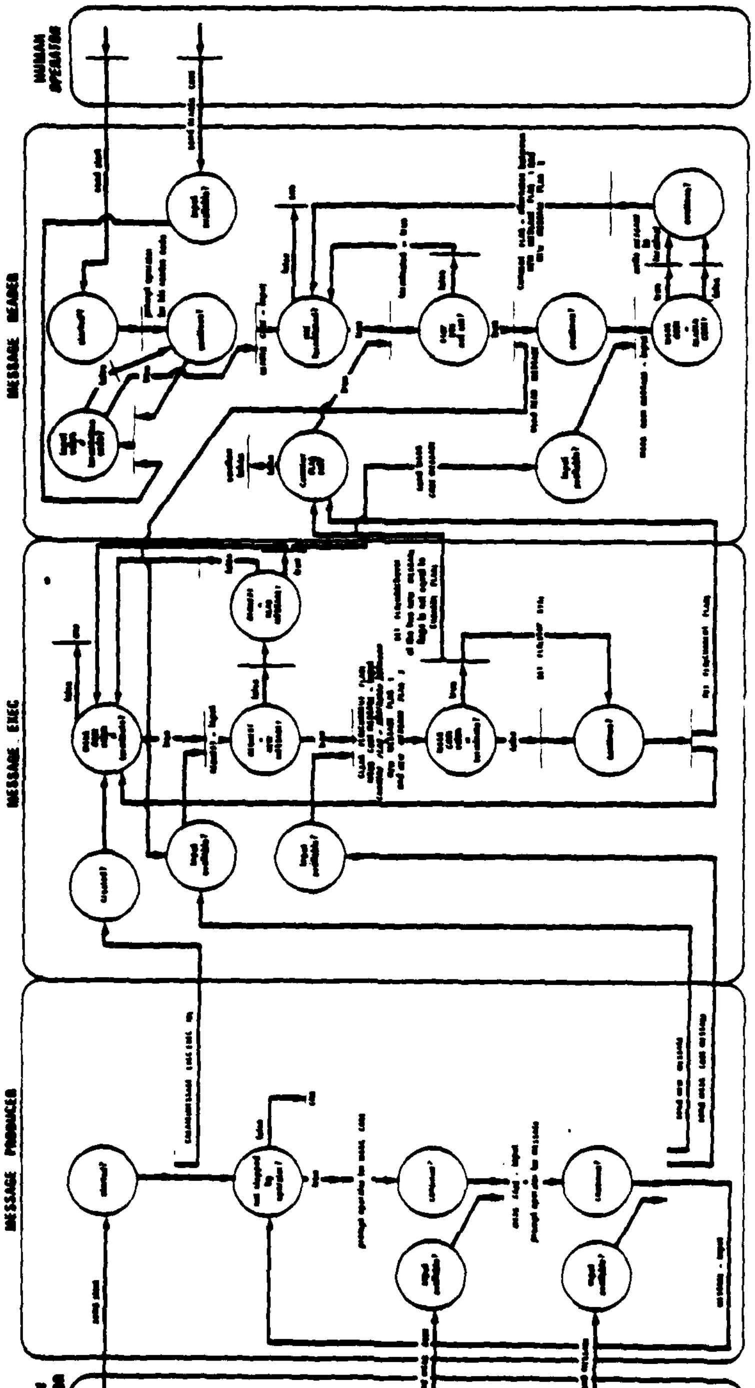
```
program TEXT_SEARCH
declare
  type SIGNAL is (START_SEARCH, SEARCH_DONE)

task REQUEST_HANDLER
declare
  I, NUM_KEYS, DATA_BASE_CHOICE : INTEGER
  KEYS(5) : STRING(1..80)
begin
  write (<description of program>) to terminal
  prompt (<operator to continue>) to terminal
  do forever
    NUM_KEYS = 0
    do forever
      prompt (<operator for KEYS(NUM_KEYS+1)>) to terminal
      if (<end of file received>) then
        exit do
      else
        NUM_KEYS=NUM_KEYS+1
      end if
    end do
    prompt(<operator to enter his DATA_BASE_CHOICE>) to terminal
    send (START_SEARCH) to QUEUE_MANAGER
    send (DATA_BASE_CHOICE, NUM_KEYS) to QUEUE_MANAGER
    <send (KEYS(1)...KEYS(NUM_KEYS)) to QUEUE_MANAGER>
    prompt (<operator to continue program>) to terminal
    if (<end of file received>) then
      exit do
    end if
  end do
end REQUEST_HANDLER

task QUEUE_MANAGER
declare
  REQUEST : SIGNAL
  SEARCH_ID, PRINT_ID, INDEX, NUM_KEYS(10), NUM_FILES, I : INTEGER
  DATA_BASE_CHOICE : INTEGER
  KEYS(10, 5), FILE_NAME : STRING(1..80)
begin
  do forever
    accept (REQUEST) from REQUEST_HANDLER or SEARCH
    if (REQUEST = START_SEARCH) then
      CREATE(SEARCH, SEARCH_ID)
      INSERT_SEARCH_ID(SEARCH_ID, INDEX)
      <accept DATA_BASE_CHOICE, NUM_KEYS, and the set of KEYS from
       REQUEST_HANDLER and send SEARCH_ID, DATA_BASE_CHOICE, NUM_KEYS and
       the set of KEYS to the SEARCH process that was just created >
      NUM_KEYS(INDEX) = <number of keys>
      (KEYS(INDEX, 1...NUM_KEYS) = <keys just received>
    else if (REQUEST = SEARCH_DONE) then
      CREATE(PRINT_FILE, PRINT_ID)
      accept (SEARCH_ID, NUM_FILES) from SEARCH
      GET_SEARCH_INDEX(SEARCH_ID, INDEX)
      <send NUM_KEYS(INDEX), and the set of KEYS pointed at by INDEX to the
       PRINT_FILE process that was just created>
      <accept the set of FILE_NAMEs from the SEARCH process and send NUM_FILES
       and the set of FILE_NAMEs to the PRINT_FILE process>
    end if
  end do
end QUEUE_MANAGER

task SEARCH
declare
  NUM_KEYS, I, NUM_FILES, SEARCH_ID, DATA_BASE_CHOICE : INTEGER
  KEYS(5), FILE_NAMES(100), FILE_NAME : STRING(1..80)
begin
  <accept SEARCH_ID, DATA_BASE_CHOICE, NUM_KEYS and the set of KEYS
   from QUEUEMANAGER>
  <open specified data base directory file>
  NUM_FILES = 0
  do forever
    read (FILE_NAME) from directory
    if (<end of file received>) then
      exit do
    end if
    if (ALL_KEYS_IN_FILE(FILE_NAME, NUM_KEYS, KEYS, KEY_LENGTHS)) then
      NUM_FILES=NUM_FILES+1
      FILE_NAMES(NUM_FILES)=FILE_NAME
    end if
  end do
  <send SEARCH_DONE, SEARCH_ID, NUM_FILES, and the set of FILE_NAMES
   to QUEUE_MANAGER>
end SEARCH

task PRINT_FILE
declare
  NUM_KEYS, NUM_FILES, I : INTEGER
  KEYS(5), FILE_NAME : STRING(1..80)
begin
  <create output file "TEXT_MATCH.DAT">
  <accept NUM_KEYS and the set of KEYS from QUEUE_MANAGER and write
   the set of KEYS to "TEXT_MATCH.DAT" >
  <accept NUM_FILES and the set of FILE_NAMEs from QUEUE_MANAGER and
   write the set of FILE_NAMEs to "TEXT_MATCH.DAT"
end PRINT_FILE

begin
  start REQUEST_HANDLER
end TEXT_SEARCH
```

TEXT SEARCH (D)

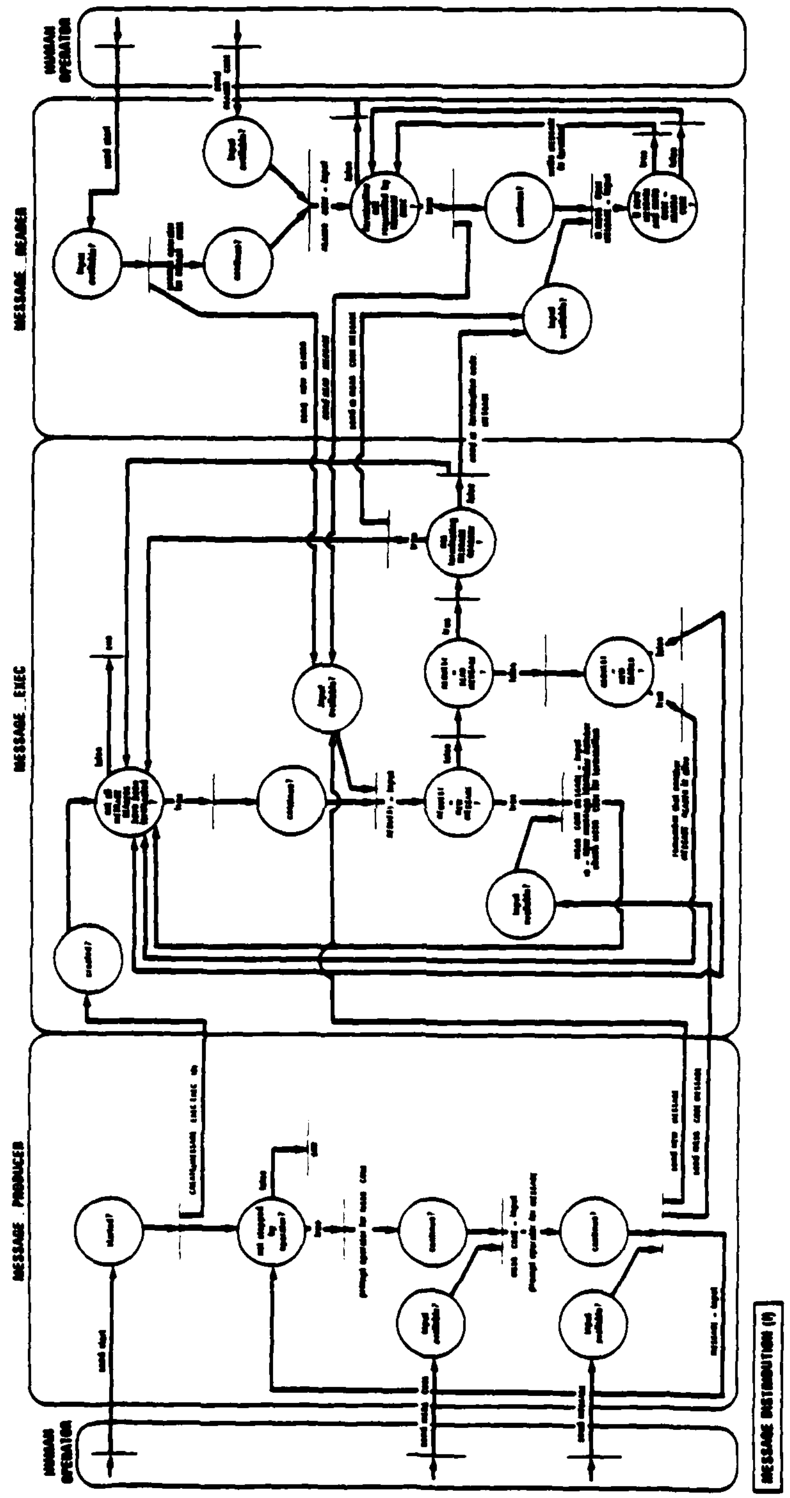
PETRI NETS

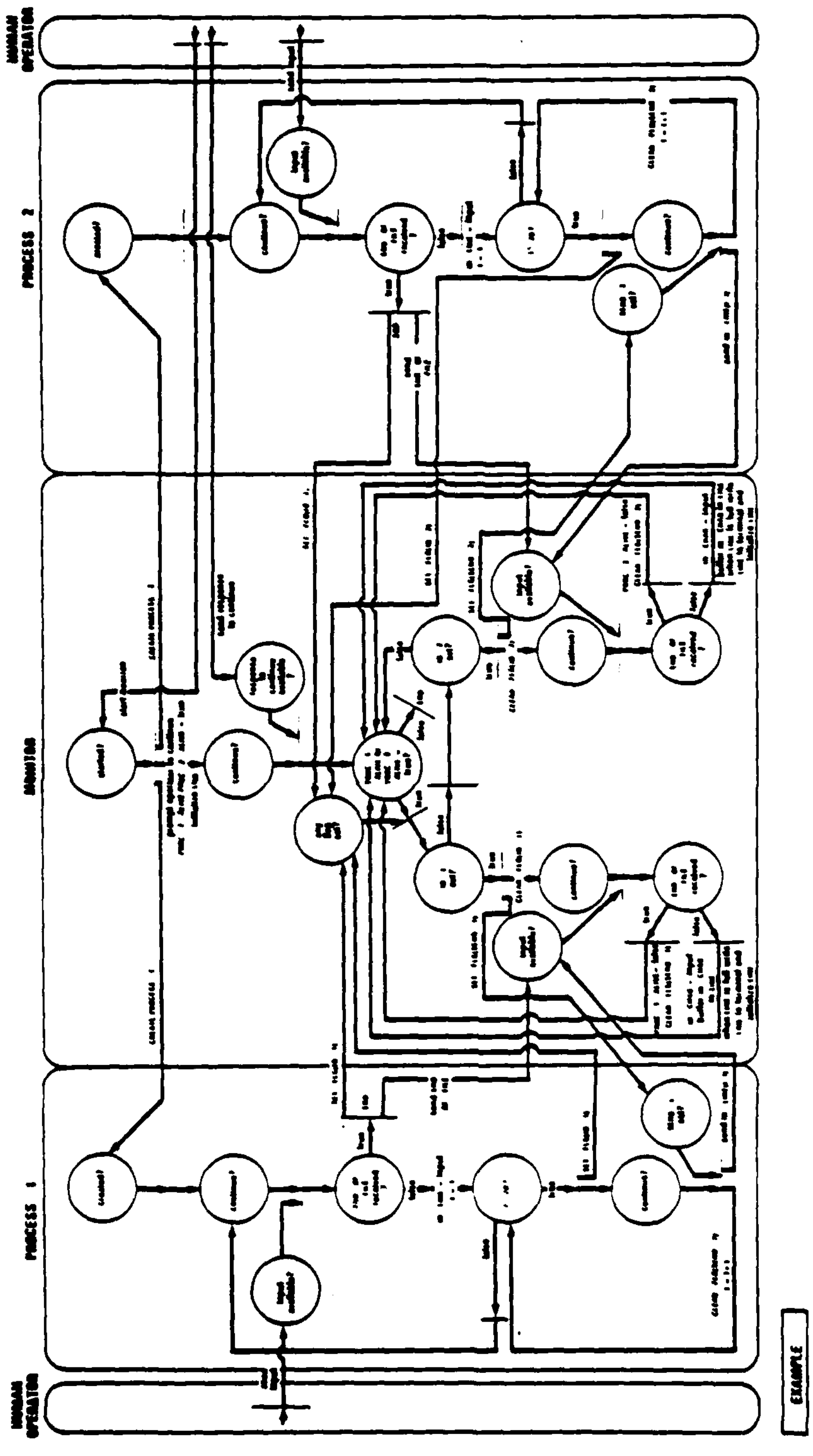


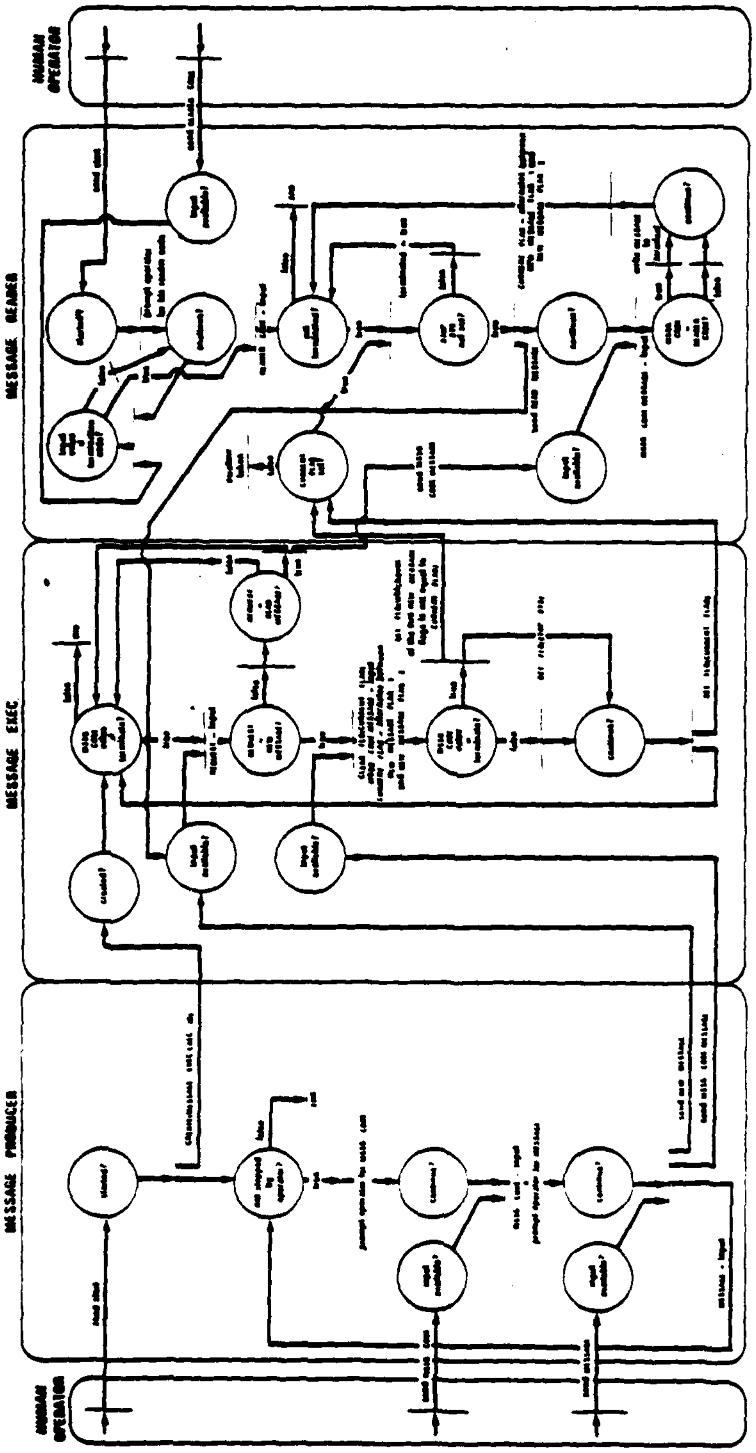
EXAMPLE



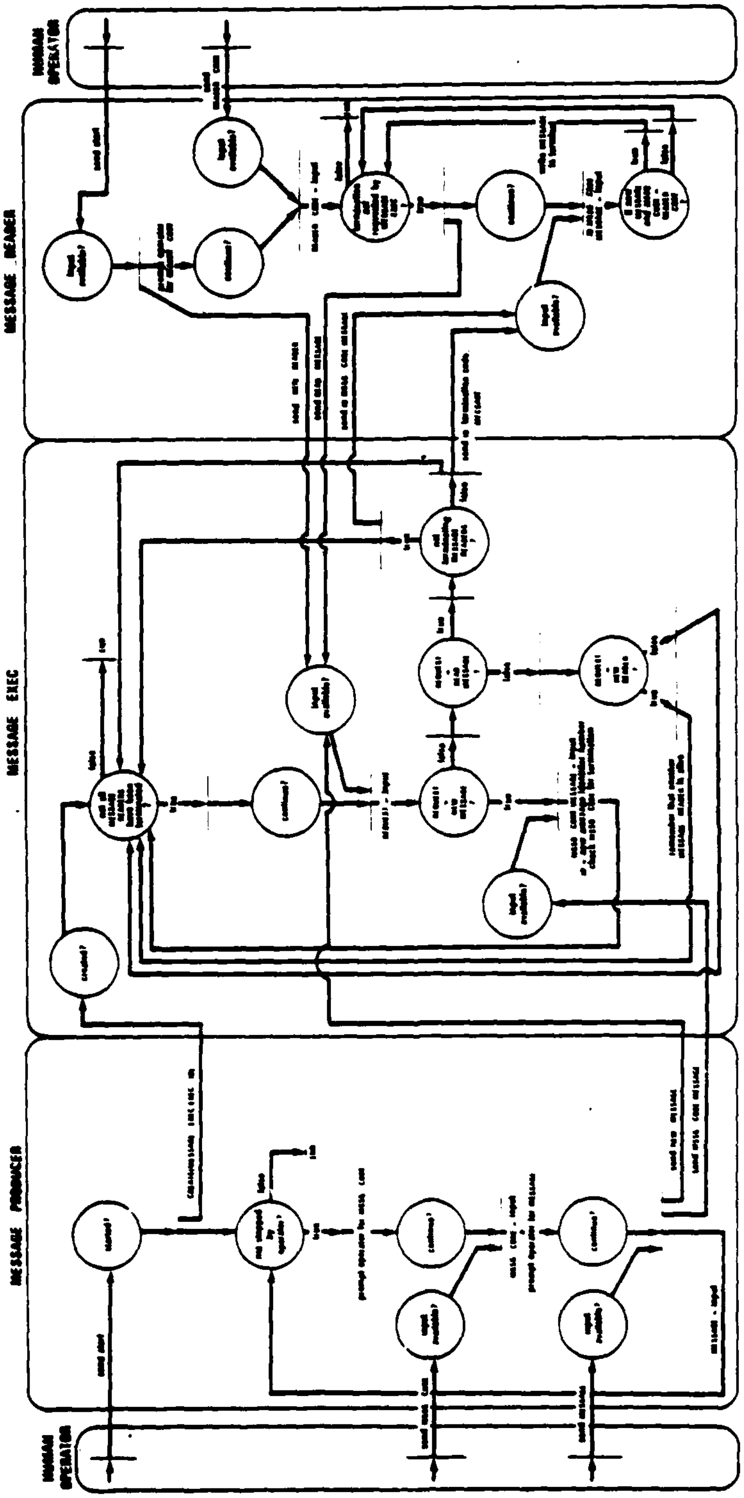
MESSAGE DISTRIBUTION (C)


HUMAN OPERATOR
MESSAGE_PRODUCER
MESSAGE_EXEC
MESSAGE_READER
HUMAN OPERATOR
MESSAGE DISTRIBUTION (9)

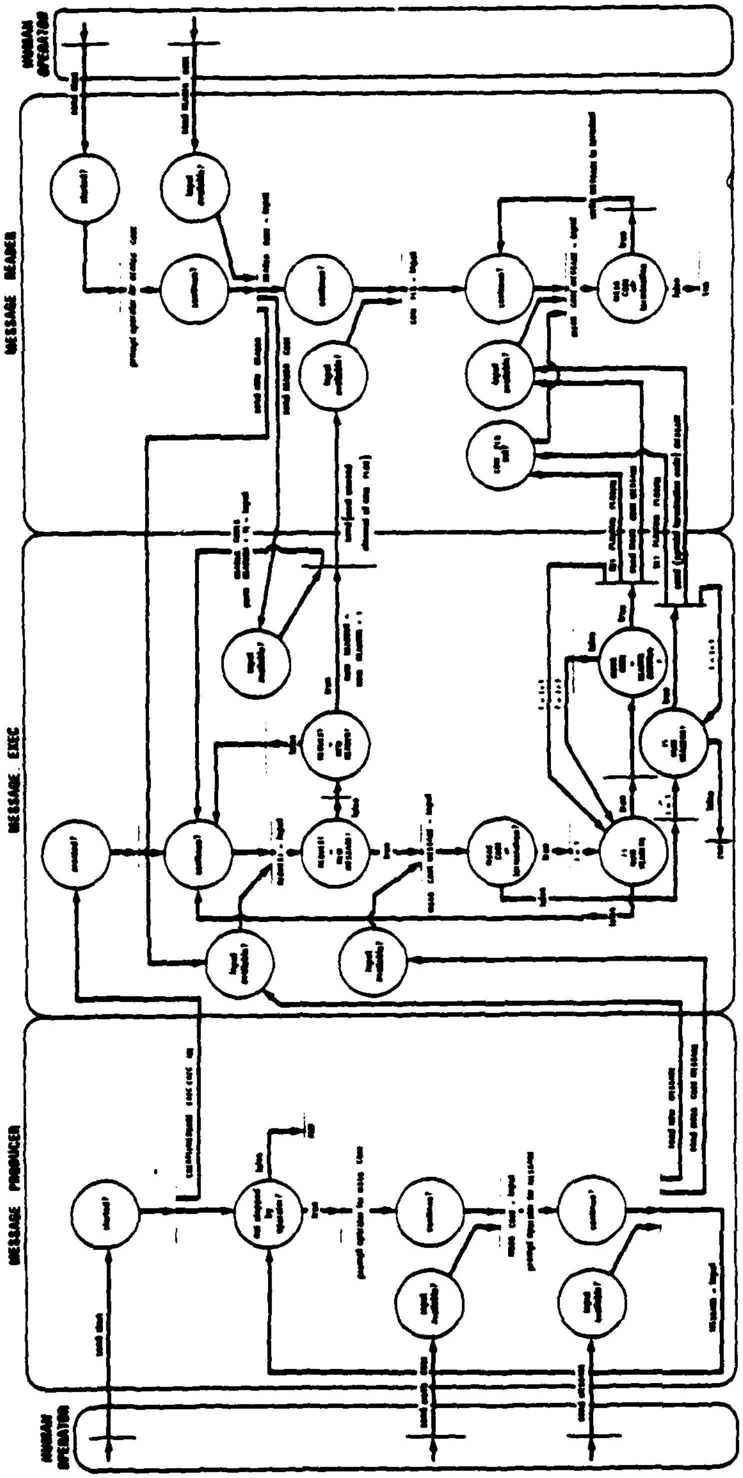

HUMAN OPERATOR
PROCESS 1
MONITOR
PROCESS 2
HUMAN OPERATOR
EXAMPLE



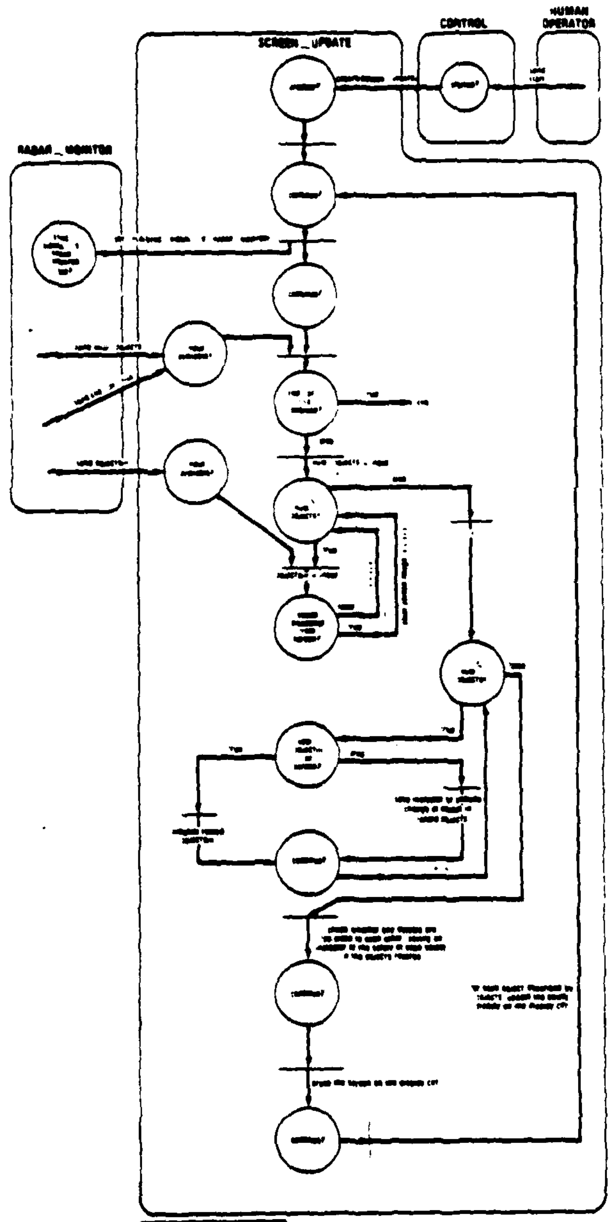
MESSAGE DISTRIBUTION (C)



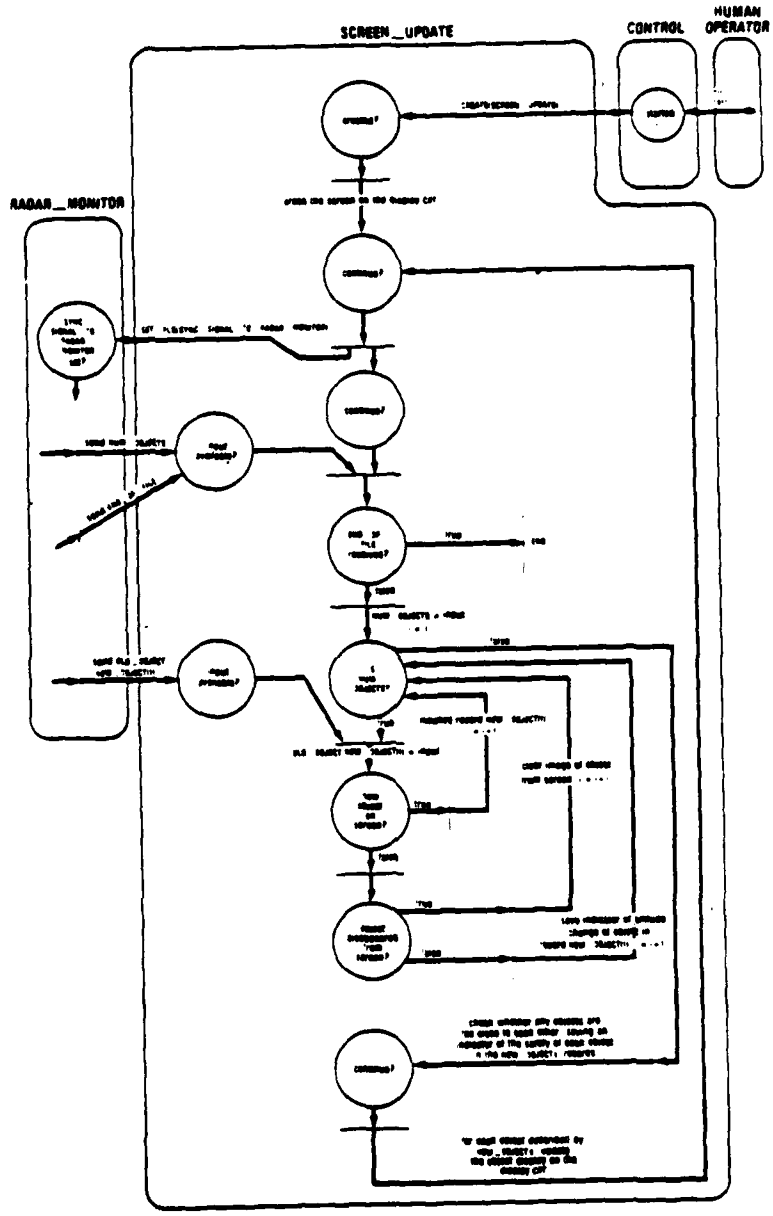
MESSAGE DISTRIBUTION (1)



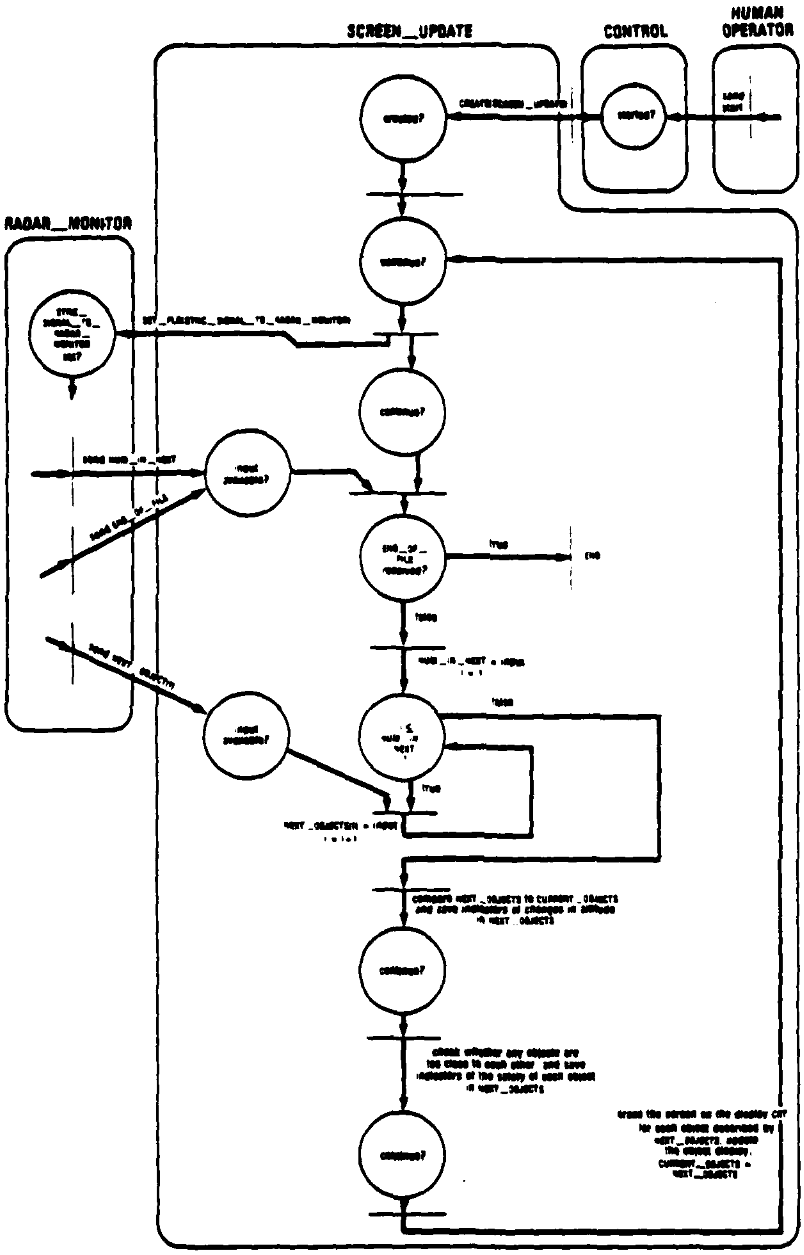
MESSAGE DISTRIBUTION (B)



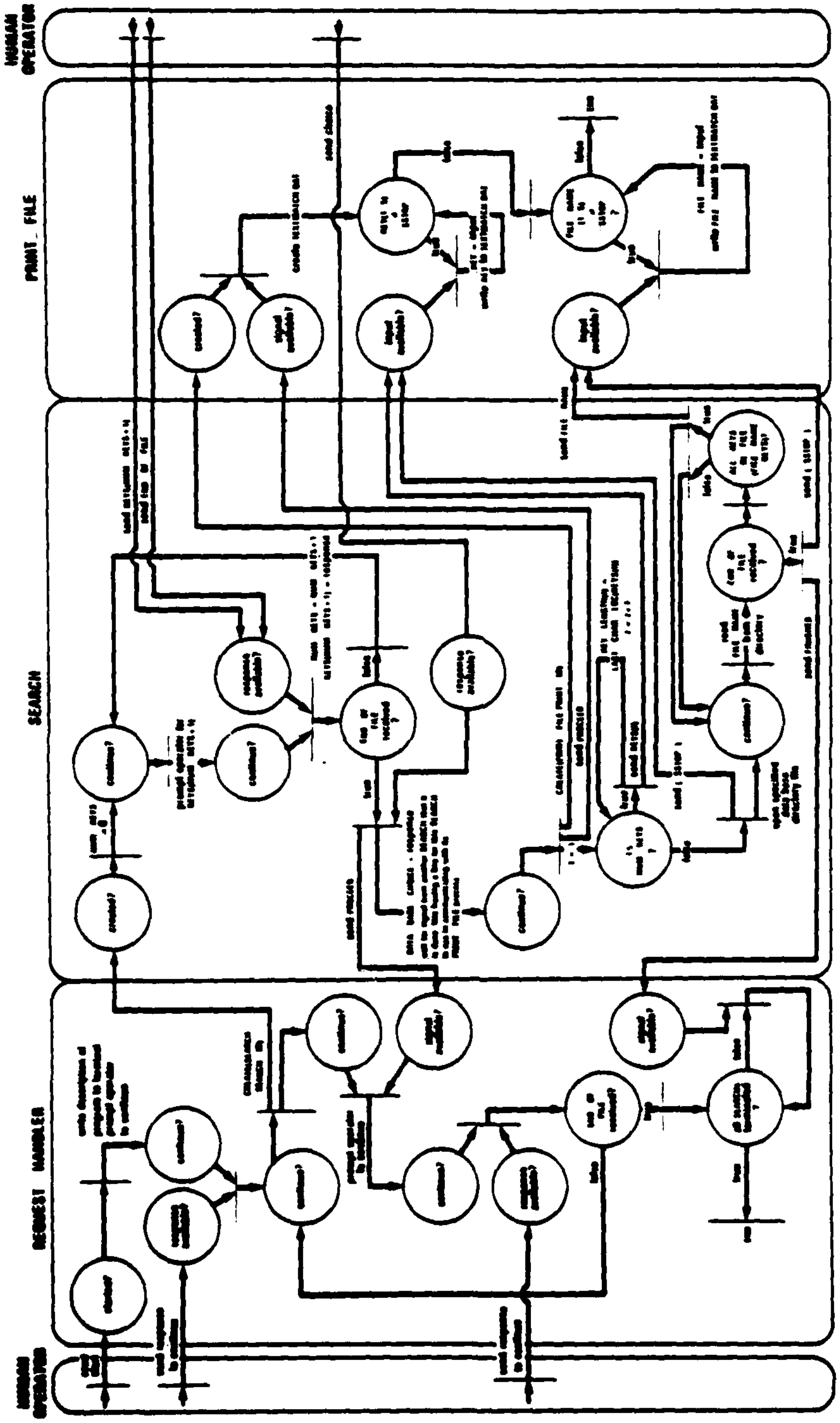
AIR TRAFFIC DISPLAY(C)



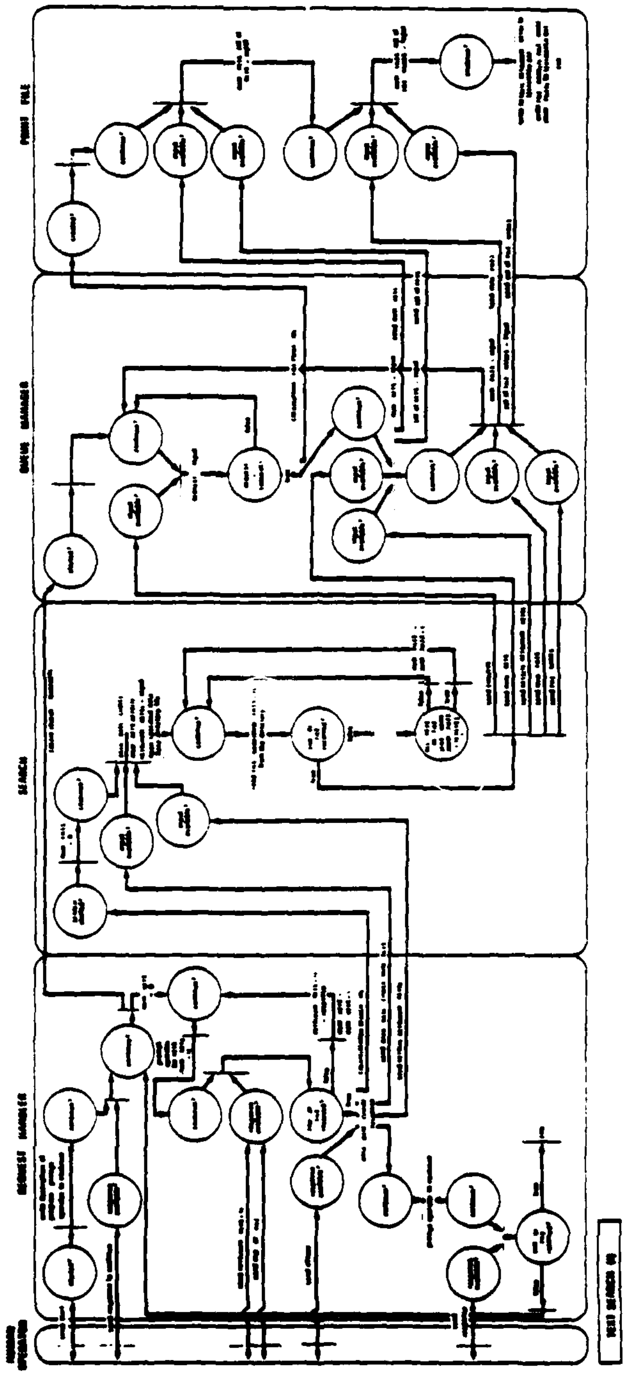
AIR TRAFFIC DISPLAY (I)

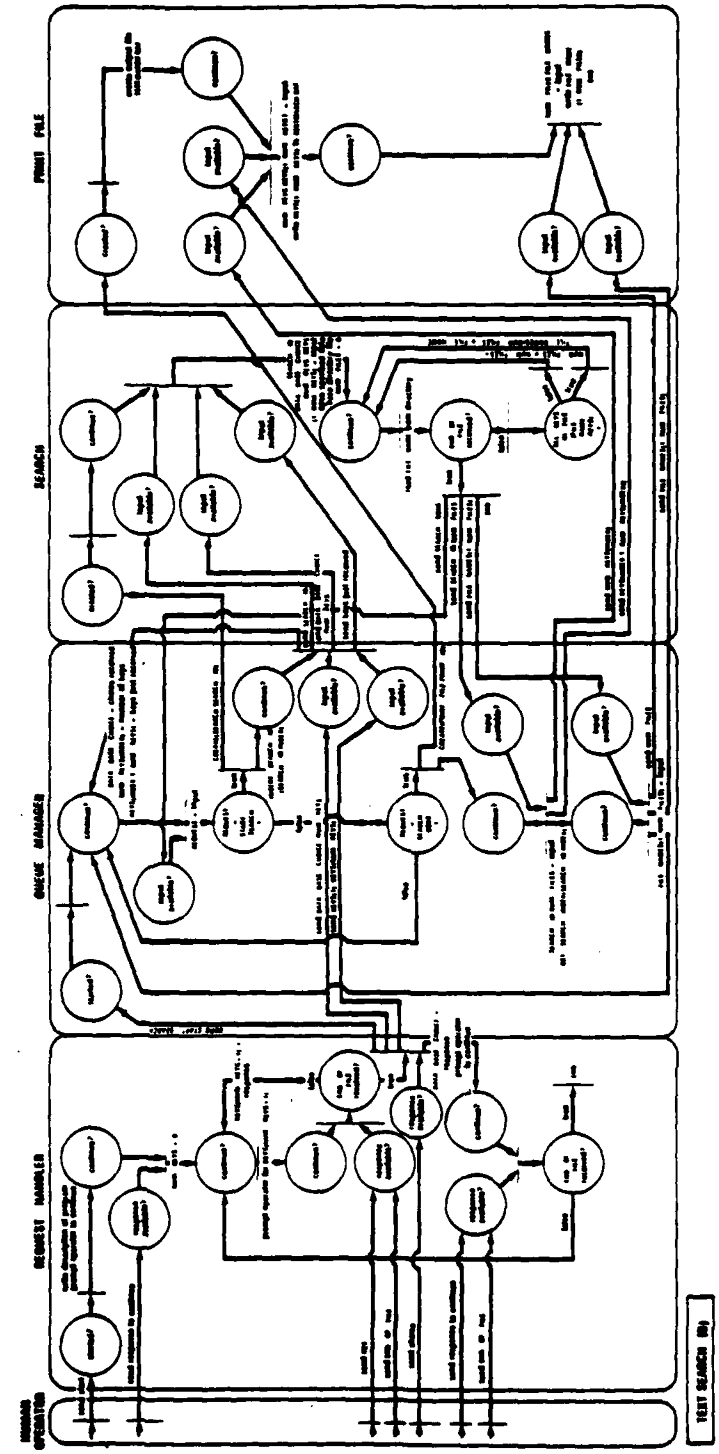


AIR TRAFFIC DISPLAY(D)



TEXT SEARCH (C)




HUMAN OPERATOR
REQUEST HANDLER
QUEUE MANAGER
SEARCH
PRINT FILE
TEXT SEARCH (b)

TECHNICAL REPORTS DISTRIBUTION LIST

OFFICE OF NAVAL RESEARCH

Engineering Psychology Group

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CAPT Paul R. Chatelier
Office of the Deputy Under
Secretary of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, DC 20301

Dr. Dennis Leedom
Office of the Deputy Under
Secretary of Defense (C^3I)
Pentagon
Washington, DC 20301

Department of the Navy

Engineering Psychology Group
Office of Naval Research
Code 442 EP
Arlington, VA 22217 (2 cys.)

Aviation & Aerospace Technology
Programs
Code 210
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Communication & Computer
Technology Programs
Code 240
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Information Sciences Division
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. J. S. Lawson
Naval Electronic Systems Command
NELEX-06T
Washington, DC 20360

Department of the Navy

Tactical Development & Evaluation
Support Programs
Code 230
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Manpower, Personnel & Training
Programs
Code 270
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Special Assistant for Marine Corps
Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

CDR James Offutt, Officer-in-Charge
ONR Detachment
1030 East Green Street
Pasadena, CA 91106

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, DC 20375

Dr. Robert E. Conley
Office of Chief of Naval Operations
Command and Control
OP-094H
Washington, DC 20350

Department of the Navy

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, DC 20350

Dr. Alfred F. Smode
Training Analysis and Evaluation
Group
Orlando, FL 32813

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Dr. L. Chmura
Naval Research Laboratory
Code 7592
Computer Sciences & Systems
Washington, DC 20375

Chief, C^3 Division
Development Center
MCDEC
Quantico, VA 22134

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 334A
Washington, DC 20361

Commander
Naval Air Systems Command
Crew Station Design
NAVAIR 5313
Washington, DC 20361

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 81323
Washington, DC 20360

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research Lab
Naval Submarine Base
Groton, CT 06340

Department of the Navy

Combat Control Systems Department
Code 35
Naval Underwater Systems Center
Newport, RI 02840

Human Factors Department
Code N-71
Naval Training Equipment Center
Orlando, FL 32813

CDR Norman E. Lane
Code N-7A
Naval Training Equipment Center
Orlando, FL 32813

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, DC 20380

HQS, U. S. Marine Corps
ATTN: CCA40 (Major Pennell)
Washington, DC 20380

Commanding Officer
MCTSSA
Marine Corps Base
Camp Pendleton, CA 92055

Human Factors Technology Admin.
Office of Naval Technology
Code MAT 0722
800 North Quincy Street
Arlington, VA 22217

Mr. Lawrence Lindley
Naval Avionics Center
Code 821
6000 East 21st Street
Indianapolis, IN 46218

Mr. Philip Andrews
Naval Sea Systems Command
NAVSEA 03416
Washington, DC 20362

Larry Olmstead
Naval Surface Weapons Center
NSWC/DL
Code N-32
Dahlgren, VA 22448

Department of the Navy

Mr. Ronald Leask
Naval Underwater Systems Center
Code 3251
Smith Street
New London, CT 06320

Navy Personnel Research and
Development Center
Planning & Appraisal Division
San Diego, CA 92152

Mr. Stephen Merriman
Human Factors Engineering Div.
Naval Air Development Center
Warminster, PA 18974

Mr. Jeffrey Grossman
Human Factors Branch
Code 3152
Naval Weapons Center
China Lake, CA 93555

Dean of Academic Departments
U. S. Naval Academy
Annapolis, MD 21402

Dr. S. Schiflett
Human Factors Section
Systems Engineering Test
Directorate
U.S. Naval Air Test Center
Patuxent River, MD 20670

CDR C. Hutchins
Code 55
Naval Postgraduate School
Monterey, CA 93940

Office of the Chief of Naval
Operations (OP-115)
ATTN: Dr. Robert Carroll
Washington, DC 20350

Mr. Marshall R. Potter
Project Management Support Branch
System Effectiveness & Component
Engineering Division
Code NAVELEX 8143
Washington, DC 20360

Department of the Navy

Commanding Officer
Naval Health Research Center
San Diego, CA 92135

Commander, Naval Air Force,
U. S. Pacific Fleet
ATTN: Dr. James McGrath
Naval Air Station, North Island
San Diego, CA 92135

Dr. Robert Blanchard
Navy Personnel Research and
Development Center
Command and Support Systems
San Diego, CA 92152

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93042

Mr. John Impagliazzo
Code 101
Naval Underwater Systems Center
Newport, RI 02840

Mr. Harry Crisp
Code N-51
Combat Systems Department
Naval Surface Weapons Center
Dahlgren, VA 22448

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, DC 20310

Dr. Edgar M. Johnson
Technical Director
U. S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Director, Organizations and
Systems Research Laboratory
U. S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Department of the Army

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

Department of the Air Force

U. S. Air Force Office of Scientific Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

AFHRL/LRS TDC
ATTN: Susan Ewing
Wright-Patterson AFB, OH 45433

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks AFB, TX 78235

Foreign Addresses

Director, Human Factors Wing
Defence & Civil Institute of Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314 (12 cys.)

Dr. Clint Kelly
Director, System Sciences Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Other Government Agencies

Dr. M. Montemerlo
Human Factors & Simulation
Technology, RTE-6
NASA HQS
Washington, DC 20546

Other Organizations

Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311

Dr. Robert T. Hennessy
NAS - National Research Council (COHF)
2101 Constitution Avenue, N.W.
Washington, DC 20418

Dr. Robert C. Williges
Department of Industrial Engineering and OR
Virginia Polytechnic Institute and State University
130 Whittemore Halle
Blacksburg, VA 24061

Mr. Edward M. Connelly
Performance Measurement Associates, Inc.
410 Pine Street, S.E.
Suite 300
Vienna, VA 22180

Dr. J. O. Chinnis
Decision Science Consortium
Suite 721
7700 Leesburg Pike
Falls Church, VA 22043

Dr. Richard Pew
Bolt, Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238

Psychological Documents (3 cys.)
ATTN: Dr. J. G. Darley
N-565 Elliott Hall
University of Minnesota
Minneapolis, MN 55455

8