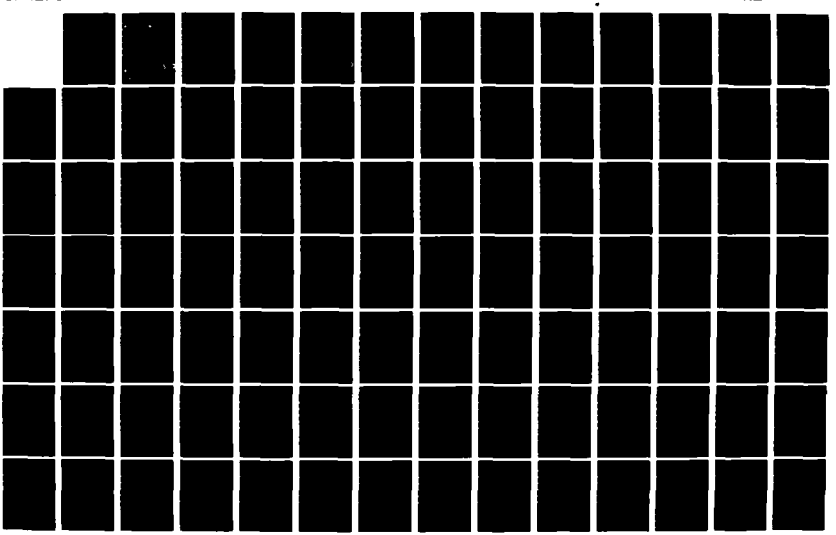
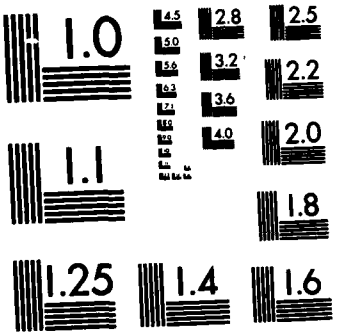
AD-A124 712 AUTOMATED TOOLS FOR TEST AND ANALYSIS OF RADAR WARNING 1/2
RECEIVER SOFTWARE(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. J R ROBERTSON
UNCLASSIFIED DEC 82 AFIT/GCS/EE/82D-28 F/G 17/9 NL





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

AUTOMATED TOOLS FOR TEST AND ANALYSIS OF RADAR WARNING RECEIVER SOFTWARE

THESIS

AFIT/GCS/EE/82D-28 JOEL R. ROBERTSON
CIV USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/82D-28	2. GOVT ACCESSION NO. AD-A124 712	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Automated Tools for Test and Analysis of Radar Warning Receiver Software		5. TYPE OF REPORT & PERIOD COVERED MS THESIS 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joel R. Robertson, Civilian, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS WR/ALC-MMRRVA Robins AFB, Georgia 31098		12. REPORT DATE December 1982 13. NUMBER OF PAGES 156
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		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)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17
FREDERIC C. LYNCH, Major, USAF
Director of Public Affairs

4 JAN 1983

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AIC)
Wright-Patterson AFB OH 45433

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

Computer Programming
Computer Programs
Computer Program Verification
Debugging (Computers)

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

See reverse

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract

The Data Extraction and Analysis System (DEAS) was designed and implemented for the Electronic Warfare Avionics Integrated Support facility at the Warner Robins Air Logistics Center. The DEAS is designed to be used with the ALR-46 Integrated Support System (ISS) to assist testing and evaluation of operational flight software. The system is written in DEC standard Pascal.

AFIT/GCS/EE/82D-28

AUTOMATED TOOLS FOR TEST AND ANALYSIS OF RADAR WARNING RECEIVER SOFTWARE

THESIS

AFIT/GCS/EE/82D-28 JOEL R. ROBERTSON
CIV USAF

Approved for public release; distribution unlimited

AFIT/GCS/EE/82D-28

AUTOMATED TOOLS
FOR TEST AND ANALYSIS OF
RADAR WARNING RECEIVER SOFTWARE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment for the Degree of
Master of Science in Computer Engineering

by

Joel R. Robertson, BSEE
CIV USAF
Graduate Electrical Engineering

December 1982

PREFACE

The ALR-46 Data Extraction and Analysis System was developed for the Electronic Warfare Avionics Support Facility to assist in testing of the ALR-46 operational flight program.

I would like to express my appreciation to my thesis committee , and in particular Dr. Gary B. Lamont my thesis advisor, for their assistance and guidance during this thesis. Finally, I would like to extend my sincerest thanks to my wife, Tami, for her encouragement and assistance in typing this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The Data Extraction and Analysis System (DEAS) was designed and implemented for the Electronic Warfare Avionics Integrated Support facility at the Warner Robins Air Logistics Center. The DEAS is designed to be used with the ALR-46 Integrated Support System (ISS) to assist testing and evaluation of operational flight software. The system is written in DEC standard Pascal.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND.

The primary function of an electronic warfare (EW) system is to detect radiation, identify and evaluate threats, and generate appropriate responses (Ref 13:3). One type of electronic warfare system is the Radar Warning Receiver (RWR). The response produced by a RWR is to inform the pilot when his aircraft is being illuminated by a hostile threat. The two types of RWR's are analog and digital (Ref 8). Analog RWR's were developed first and gave only signal direction and relative signal strength information. The digital RWR's contain an embedded computer and identify the threat by comparing measured signal parameters to those stored in a data base. The digital RWR would then display the type and position of the threat on a cockpit CRT display. The program that the RWR executes is called the Operational Flight Program (OFP) and the data base is called the Emitter Identification Data (EID).

1.1.1 ALR-46 Radar Warning Receiver - The Electronic Warfare Avionics Integrated Support Facility (EWAISF) at Robins AFB provides hardware and software support for the Air Force Electronic Warfare systems. One of the largest systems maintained by the EWAISF is the ALR-46. The ALR-46 is a 2 to 18 Ghz digital radar warning receiver (RWR). It is capable of analyzing and identifying up to 16 emitters concurrently. The ALR-46 is the most widely used RWR in the Air Force inventory (Ref 23:193). The ALR-46, shown in figure 1, consists of the following (Ref 16:6,13): four amplifier/detectors located at 45, 135, 225, and 315 degrees, an omni-directional low band receiver, the signal processor, up to two azimuth indicators, one "threat display and control unit" for each azimuth indicator, and interfaces to aircraft avionic systems.

Two hardware versions of the ALR-46 are currently maintained by the EWAISF, the original version and an updated version. The original version of the ALR-46 has a five board CPU and draws the CRT display symbols with software. The updated version has a single board CPU and has a dedicated graphics processor for drawing the symbols on the CRT. The different hardware configurations require different versions of software. The Tactical Air Command (TAC) and the Strategic Air Command (SAC) have different mission requirements and therefore use different versions of software for each hardware configuration. Therefore, each of the two hardware configurations have two versions of the

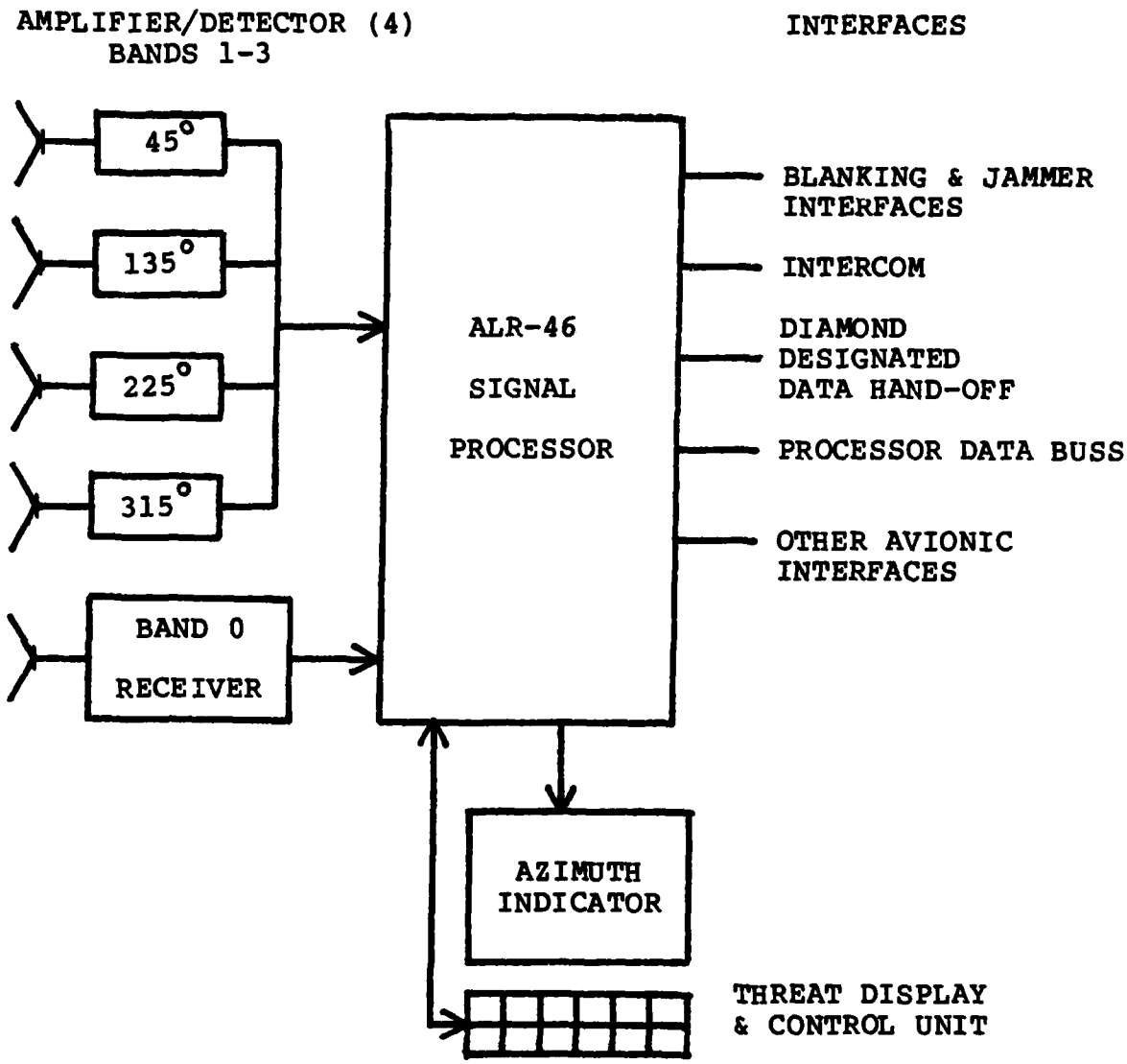


Figure 1. ALR-46 Radar Warning Receiver

operational flight software for a total of four software versions.

1.1.2 Program Maintenance - Routine changes to the flight software are made during a Block Change Cycle (Ref 14: Sec 4, 7). A Block Change cycle is a scheduling method to collect, develop, implement, and test computer program changes in a fixed time frame (Ref 14: Sec 1, 4).

Program changes are collected to be processed concurrently in each change cycle. Program changes are prioritized and placed in a queue up to the block cycle cutoff date. Changes received after the cutoff date are held for the next change cycle. The software for each of the four versions is updated once each year. The primary reasons for updating the software are: to update the EID to include new intelligence data or to reflect changing mission requirements, to add new capabilities to the OFP, and to corect errors found by either the user or by maintenance personnel.

The ALR-46 Integration Support System (ISS), provides a facility for testing and evaluating the performance of the ALR-46. Data can be extracted from the signal processor without interfering with the program execution. The ISS allows operation of the signal processor with up to 32K of external RAM or with it's internal RAM/PROM. The use of external RAM allows program modification or instrumentation without "burning" new PROMS (Ref 2: Sec 3,1).

1.2 PROBLEM.

The software delivered with the ISS enables the user to load operational flight software into the ALR-46 signal processor RAM, to extract data from the 32K external RAM memory without interfering with OFP execution, and to validate proper operation of the data extraction subsystem

(Ref 1: Sec 2,1). Software has been developed (Ref 22) in response to user requirements to duplicate the ALR-46 CRT on a Tektronix 4027 color graphics terminal and to display extracted data on a video terminal. However, since no other software tools have been developed for the ALR-46 ISS, flight software is still tested and debugged with the traditional co-resident debug program. Testing reqirements need to be established, and the necessary software tools need to be developed to exploit the capabilities of the new systems as fully as possible.

1.3 SCOPE.

This study has three goals. The first goal is to establish the requirements for automated tools for test and analysis of ALR-46 flight software. The second goal is to identify software tools already available. The final goal is to develop a system of test tools for use with the ALR-46 ISS. An overall system design of the tools will be done before coding will begin. The tools will then be coded and tested. Those tools which are not currently available but outside the scope of this study will be recommended for future investigation. The completed software will be delivered to the sponsor (EWAISF).

1.4 STANDARDS.

Top-down procedures of software development will be adhered to. This means that program development will proceed from program requirements to functional specification, to design, to coding, and to validation and verification.

The top-down approach leads to structured programs in which the main program can be decomposed into smaller subprograms. The design of subprogram "stubs" allow testing to begin as code is developed (Ref 20:29,31).

Both FORTRAN and Pascal are available at the sponsor's facility (EWAISF). Even though FORTRAN is the most commonly used high level language at the EWAISF, Pascal was chosen as the language for program development. Pascal is a block orientated language and supports structured programming constructs (Ref 7). The use of Pascal will enable the development of a more easily maintained system.

1.5 APPROACH.

The first objective of this study is to determine the user's requirements for software testing. This is to be done primarily through a series of interviews with EWAISF personnel.

The second objective is to determine the software tools requirements. A detailed literature search will be done to determine which software tools are already available. A feasibility study will be done to determine which tools are within the scope of this effort. The third and major effort is to design the system of test tools. The final objective will be to code and test the software tools. An integrated test plan will be developed. Final testing and installation will be done on the ISS at Robins AFB at the prerogative of the EWAISF.

1.6 THESIS DEVELOPMENT.

This thesis will be developed in five sections. Chapter 2 covers the overall testing requirements for electronic warfare embedded computer systems in general and the ALR-46 RWR in particular. Chapter 3 describes the requirements for the automated test tools which will be developed during this study. Chapter 4 discusses the design of the automated test tools. Chapter 5 describes the implementation and the testing of the tools. The final chapter covers results and recommendations for further study. Appendix A contains an index of available software testing tools. Appendix B contains the formats required by the software tools. Appendix C contains the Structured Analysis and Design Technique (SADT) diagrams. Appendix D contains the structure charts. Appendix E contains the data dictionary listed in alphabetical order. Appendix F

contains the source listing of the software tools. Finally, Appendix G contains the test documentation for the software tools.

CHAPTER 2

SYSTEM REQUIREMENTS

2.1 INTRODUCTION.

A software requirement is a need established for a piece of software by an organization in order to achieve certain goals. The requirement-generation activity culminates in the approvals, negotiations, and commitments of resources necessary to initiate, sustain, and complete the software development. The software requirements should identify the objectives of the program, its environment, the configuration required for its operation, and the resources required for its support. The software requirements should be complete enough to allow development to proceed without major changes in the requirements (Ref 21:2-3).

This chapter will discuss the overall requirements for ARL-46 flight software testing. The environment in which software testing is to be done will be presented, and user requirements for software test tools will be discussed. Available software tools which meet these requirements will

also be described. Finally, the method used in determining which software tools to develop will be discussed.

2.2 ENVIRONMENT.

2.2.1 Introduction - The environment in which the software tools must operate is composed of three ALR-46 Integration Support Systems (ISS), an interface controller (VAX 11/780), and an edit and assembly station (ECLIPSE S/230). The interface controller is connected to the three ISSs and to the edit and assembly station by a star configuration computer network, as shown in figure 2. The network utilizes the Digital Equipment Corporation DECNET protocol.

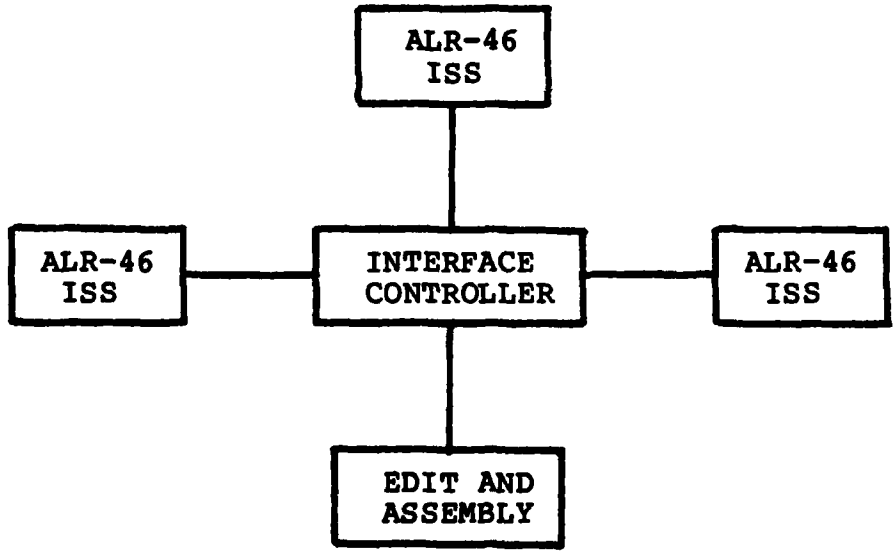


Figure 2. ALR-46 ISS network.

2.2.2 ALR-46 ISS - Each ALR-46 ISS, shown in figure 3, consists of three subsystems, the hot mock-up subsystem, the data extraction subsystem, and test equipment (Ref 2: Sec 3, 1).

2.2.2.1 Hot Mock-up Subsystem - The hot mock-up subsystem, shown in figure 4, supplies the required power for the ALR-46. Test points, not available during airborne operation, are provided for signal monitoring. The hot mock-up subsystem provides a means to control the operation of the signal processor with its resident OFP.

The hot mock-up subsystem contains the AN/ALR-46(V) radar warning receiver (RWR), the adaptor subsystem, the system monitor panel, and the 400 hz power source (Ref 2: Sec 3, 1-4).

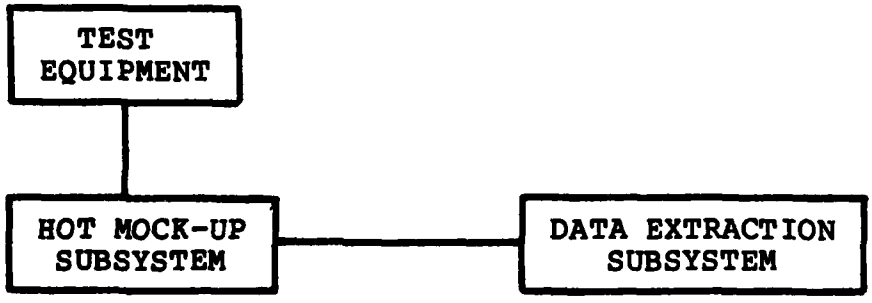


Figure 3. ALR-46 ISS.

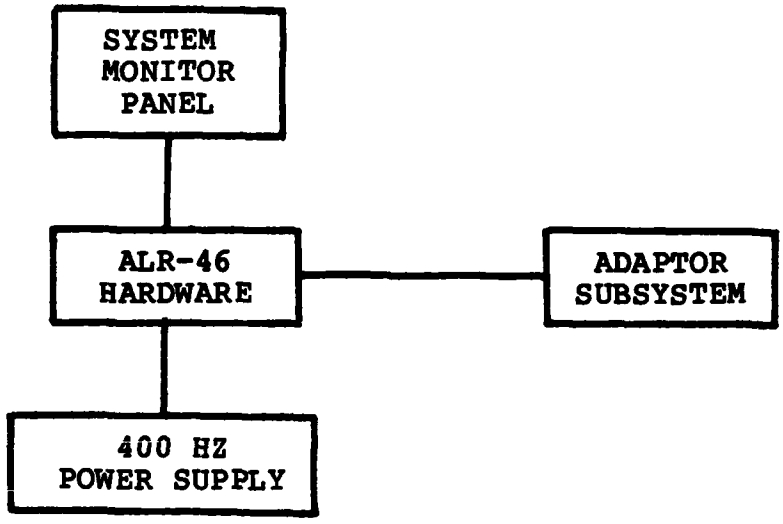


Figure 4. Hot Mock-up Subsystem

The operation of the signal processor is controlled by the adaptor subsystem. The adaptor subsystem contains the DX adaptor board, which provides the means to extract data from the ALR-46 signal processor. The adaptor subsystem also provides the basic I/O capability for the signal processor (Ref 2: Sec 3 5,10). The adaptor subsystems are slightly different for the two hardware configurations of the ALR-46 (Ref 3, 4).

The system monitor panel provides test points, used primarily to aid in detection and isolation of RWR hardware problems. The system monitor panel also contains various status indicators and controls.

2.2.2.2 Data Extraction Subsystem - The data extraction subsystem, shown in figure 5, provides the means for specifying data to be extracted and for collecting the extracted data. The data exchanges between the data extraction subsystem and the adaptor subsystem occur via a DR11-B DMA interface. The data extraction subsystem consists of a PDP-11/34 computer, a PDT-11/130 CRT, a LA120 printer, a TE16 magnetic tape unit, and two RL01 disk drives. The PDP-11/34 runs under the RSX-11M operating system (Ref 2: Sec 3, 11-12).

2.2.2.3 Test Equipment - The ISS contains test equipment for monitoring signals at the test points provided by the system monitor panel. The test equipment included are: an oscillocope, a digital multimeter, and a universal counter (Ref 2: Sec 3, 12).

2.2.3 Interface Controller - The interface controller consists of a Digital Equipment Corporation VAX 11/780 computer. The peripherals associated with the VAX 11/780 are, two REP06 disc drives, a TEU77 tape transport, a LA120 console terminal, and four VT100 CRTs.

The interface controller performs two primary functions. One function is to control the DECNET data exchanges and the other is to do off line analysis and storage of extracted data. The VAX 11/780 runs under the VMS operating system.

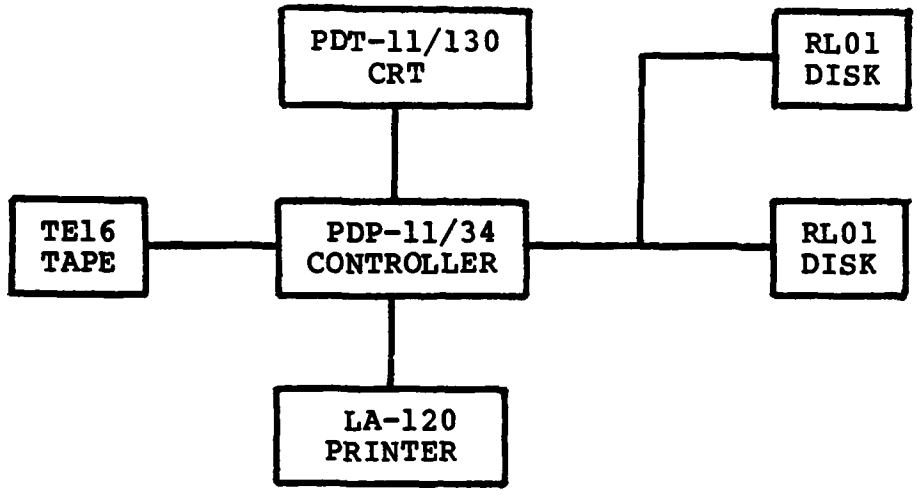


Figure 5. Data Extraction Subsystem

2.2.4 Edit And Assembly Station - The edit and assembly Station consists of a Data General Eclipse S/230 minicomputer. The peripherals associated with the Eclipse are two 20 M byte disk drives, a 190 M byte disk drive, two magnetic tape transports, a line printer, a printing terminal, and three CRTs. The Eclipse runs under the Data General Advanced Operating System (AOS). AOS is a multi-user multi-tasking operating system.

The primary function of the Eclipse is to assemble the ALR-46 flight software. The ALR-46 signal processor executes the Data General Nova instruction set. Since the Nova instruction set is a subset of the Eclipse instruction set, the ALR-46 flight software can be assembled by the AOS Macro Assembler. The assembled flight software is transferred to the interface controller which in turn routes it to the appropriate ISS.

2.3 USER REQUIREMENTS.

2.3.1 Interviews - A series of interviews have been held with EWAISF personnel in order to establish requirements for automated test and support tools. The Engineering and Reliability Branch Chief, the EW Receivers System Section Chief, the ALR-46 Unit Chief, five engineers, and one technician participated in the interviews.

Eight requirements for automated test and support tools were identified. The tools would be used for support of software development and testing. The automated tools requirements identified were:

1. Automated control of the threat generators. The software to generate threat signals is scheduled to be delivered with the threat generator system. The capability to compare the response of the ALR-46 to the threats generated has not been developed at present.

2. The capability to perform statistical analysis of the decision paths executed, such as the percentage of times one path is executed instead of another, and to determine the testing coverage.

3. The capability to determine performance characteristics, such as the execution time of programs and modules.

4. A computer simulation of the ALR-46 system. This would be used to evaluate hardware as well as software modifications.

5. The capability to generate input buffers. This would allow testing of the software without having to generate actual RF signals and for repeatability of tests.

6. Automated generation of test data.

7. The capability to test programs and modules independently.

8. The capability to assemble flight programs on the VAX 11/780. The specified function of the VAX 11/780 is offline support of the I&S. This includes mass storage, message routing, data analysis, and the production of flight software. The Eclipse S/230 is being retained for the sole function of assembly of flight software.

Two additional requirements not identified in the user interviews are: the ability to identify variable out of range conditions, and "hooks" for inclusion of previously developed software (Ref 22).

2.4 HUMAN-FACTORS REQUIREMENTS.

2.4.1 Introduction - Human factors is a discipline which attempts to take into consideration human strengths and limitations in the design of computer hardware and software (Ref 19:108-132). If human factors are not considered, a computer system may be difficult for people to operate.

Human beings vary in intelligence, education, and motivation. The general rule in considering human factors in the design of a system is to consider the needs of the intended users.

The relative importance of human factors varies with the program environment and application. Four aspects of a computer program to consider are: the number of users, the diversity of the users backgrounds, program complexity, and the consequences of user error.

2.4.2 Design Principles - Six human-factors design principles which will be discussed are: provide feedback, be consistent, minimize human memory demands, keep program simple, match the program to the operator's skill level, and sustain operator orientation.

When a user makes an action he needs to know when that action has taken effect. If an entry is made and no feedback is provided, the action may be repeated or another action may be tried, which may have unintended results.

Feedback should be immediate and appear in an expected location.

The program should be consistent. Consistency allows the user to learn the operation of one part of the program and apply that knowledge to other parts of the program.

The program should minimize human memory demands. Since computers have better memories than people, and remember things exactly, the computers memory should be relied on as much as possible. Selecting an option from a displayed menu is generally preferable to entering memorized mnemonics.

The program should be kept simple. A program which is unnecessarily complex is both difficult to learn and use.

The program interface should be matched to the operators skill level. The skill level of the intended users should be determined before the program is designed.

The possibility of a user becoming disoriented should be minimized. The user should be provided with messages telling the user where the user is and how to return to where the user came from.

2.5 AVAILABLE TOOLS.

The software tools available for software testing fall into two categories, static and dynamic.

2.5.1 Static Analysis - Static analysis of software uses methods of validation which do not require the program to actually be executed (Ref 11:83). Static analysis may use some method of simulated execution. There are three types of static analysis of programs. The first type provides general information about a program, and are not designed to find a particular kind of logical error. An example of this type would be cross reference generators. The second type uses techniques which are designed to find particular types of errors or abnormal program constructs. This class of techniques is called Static Error Analysis techniques. The third type of static analysis is Symbolic Evaluation. Several types of automated tools useful in static analysis are described.

2.5.1.1 Source Comparator - A source comparator is a computer program used to compare two versions of the same computer program source code. The program will identify changes made to the source program or establish identical configurations (Ref 18:55).

2.5.1.2 Cross-Reference Generator - A cross-reference generator is a computer program which provides cross-reference information on system components. A program can be cross-referenced with other programs, macros or parameter names. This program is useful in determining the impact of a change made in one area to other sections of the program by showing the scope of variables (Ref 18:55).

2.5.1.3 Flow-Chart Generator - A flow-chart generator is a computer program used to show the logical control-flow structure of a computer program. The logic flow is determined from the program instructions and not from the comments. The flow-charts generated by the program can be compared to the flow-charts supplied with the code to identify discrepancies (Ref 18:56).

2.5.1.4 Standards Analyzer - A standards analyzer is a computer program used to determine if predefined procedures, rules, and conventions have been followed. The standards analyzer can check for violations in conventions such as program size, comments, and structure (Ref 18:58).

2.5.2 Dynamic Analysis - Dynamic analysis primarily involves program testing (REF 10:185). There are several other techniques such as dynamic assertions and recovery control blocks.

Program testing involves executing the program with a set of sample data as input. The test output may be program output variables, intermediate values of selected variables, or timing information in "real time" systems.

2.5.2.1 Test Data Generator - A test data generator is a program which generates test data or test cases to exercise the system under test. Test data generators are useful in systems where "live" data is not available (Ref 18:56).

2.5.2.2 Data Base Analyzer - A data base analyzer is a computer program which provides information on the usage of data. It indicates whether the program inputs, uses, modifies, or outputs the data element (Ref 18:55).

2.5.2.3 Debugger - A debugger is a computer program which helps to identify and isolate program errors. It usually includes commands such as DUMP, TRACE, MODIFY, CONTENTS, BREAKPOINT, etc. (Ref 18:56, 9:426-428).

2.5.2.4 Simulator - A simulator is a computer program which provides the system under test with inputs or responses which "resembles" those which would have been provided by the device being simulated (Ref 12:3-4). A description of several of the various types of simulators follows:

2.5.2.4.1 Environment Simulator - An environment simulator is a computer program which is used to test operational programs on a host computer. The operational programs run under simulated conditions as if they were operating within the real-time constraints of a machine to which all the components of the ultimate system are attached (Ref 18:56).

2.5.2.4.2 Peripheral Simulator - A peripheral simulator is a computer program used to present functional and signal interfaces representative of a peripheral device to the target system.

2.5.2.4.3 System Simulator - A system simulator is a computerized model of the system (hardware, software, interfaces) used to predict system performance over time. A system simulator allows full control of inputs, and computer characteristics. It allows processing of intermediate outputs without destroying simulated time, and allows full test repeatability and diagnostics (Ref 18:58).

2.5.2.4.4 Instruction Simulator - An instruction simulator is a computer program used to simulate the execution characteristics of a target computer. The instruction simulator provides bit for bit fidelity with the results that would have been produced by the target computer (Ref 18:56).

2.5.2.5 Interactive Test Bed - An interactive test bed performs three functions. First, it preprocesses the module under test so that instrumentation to measure testing coverage can be inserted in the code. Second, it links the instrumented module into a test harness which can control these functions: 1) Set-up of input data, 2) Intercept of stub calls, 3) Supply of stub return data, 4) Capture of output data, and 5) Reporting of coverage. Finally, it documents the test actions so testing can be repeated (Ref 18:58).

2.6 TOOLS SELECTION.

Since the contractor is scheduled to supply software to drive the threat simulators, the development of this software will not be considered during this study. The capability for closed loop testing by generating a threat and then analyzing the response of the system cannot be addressed until the specific implementation details of the threat generators are established.

A computer simulation of the ALR-46 system has been developed and is identified in Appendix A. This simulator should be evaluated by the EWAISF before any other work is done in this area.

The production of a software package to assemble flight software on the VAX 11/780 is a large programming task but would not have sufficient theoretical content to warrant thesis level attention.

The tools selected for development meet the following requirements: to determine the testing coverage, to determine the execution time of modules, and to identify variable out of range conditions.

It has been determined that the remainder of the user requirements could not be accomplished in the time available for this study.

2.7 SUMMARY.

Many software testing tools have been developed, however most are software or application dependent. The majority of these tools were developed for use with a specific high level language, such as FORTRAN. Since the ALR-46 flight software written is Nova assembly language, the number of tools directly applicable is greatly reduced. The tools found which are directly applicable are identified in Appendix A. Further, the unique design of the ALR-46 ISS data extraction subsystem requires custom software to be written for it.

This chapter has discussed the requirements for software test tools and the types of automated tools available to meet those requirements. Finally, the process used in determining which software tools to develop during this study was described.

CHAPTER 3

SOFTWARE REQUIREMENTS

3.1 INTRODUCTION.

Chapter 2 discussed the ALR-46 flight software testing requirements, what software tools are available to meet the requirements, and which tools were selected for development in this study. This chapter will discuss the requirements for the software tools.

3.2 SOFTWARE MONITORS.

Monitors for software testing can be divided into four categories (Ref 17:405-406). The first category is composed of monitors used for enforcing or ascertaining the traversal of paths. The second category is composed of monitors used for measuring path traversal frequencies. The third type is used for detecting erroneous conditions, such as variable out of range. The fourth type is the performance monitor which is used for observing performance behaviors, such as excessive delay.

There are two approaches for ascertaining the traversal of test paths. The first method inserts code segments, called flow controlling monitors, in the test paths. A flow controlling monitor is said to be "closed" when it is set to transfer control to the test supervisor, and "open", otherwise. The use of flow controlling monitors requires modification of the software under test. The normal operation of the data extraction subsystem only allows reading of data and not insertion. Therefore, flow-controlling monitors will not be considered in this study. The second method detects the traversal of paths by analyzing the data recorded during the test run by a set of monitors called traversal markers.

The following software monitors were selected for use in the Data Extraction and Analysis system (DEAS) because they could be implemented on the ISS data extraction subsystem. The ISS data extraction subsystem allows extraction of data without insertion of probes into the code under test. The monitors to be used will determine what data will be extracted by the ISS data extraction subsystem. Performance monitors will be used to determine the execution time of modules. Traversal markers will be used to determine the path traversal frequencies. Error detecting monitors will be used to detect variable out of range errors.

3.3 SOFTWARE REQUIREMENTS.

The DEAS operates in one of five modes. The first mode, called the "performance analysis mode", measures the execution time of each subroutine call. The second mode, called the "coverage analysis mode", determines the frequency of path coverage of marked paths during a particular test. The third mode, called the "error analysis mode", detects variable out of range errors. The fourth mode, called the "trackfile analysis mode", and the fifth mode, called the "RWR mode", are from a previous study (Ref 22). The organization of the DEAS is shown in figure 6.

The Data Extraction and Analysis System (DEAS) software is composed of three tasks. The function of the first task is to specify the data to be extracted and when it is to be extracted. The second task controls the extraction of the data and saves the raw data in a disk file. This software was delivered by the contractor who developed the ISS (Ref 1). The third task analyzes and presents the data in tabular format. This will provide the test engineer with test results in a form which can be more easily decyphered than large amounts of raw data.

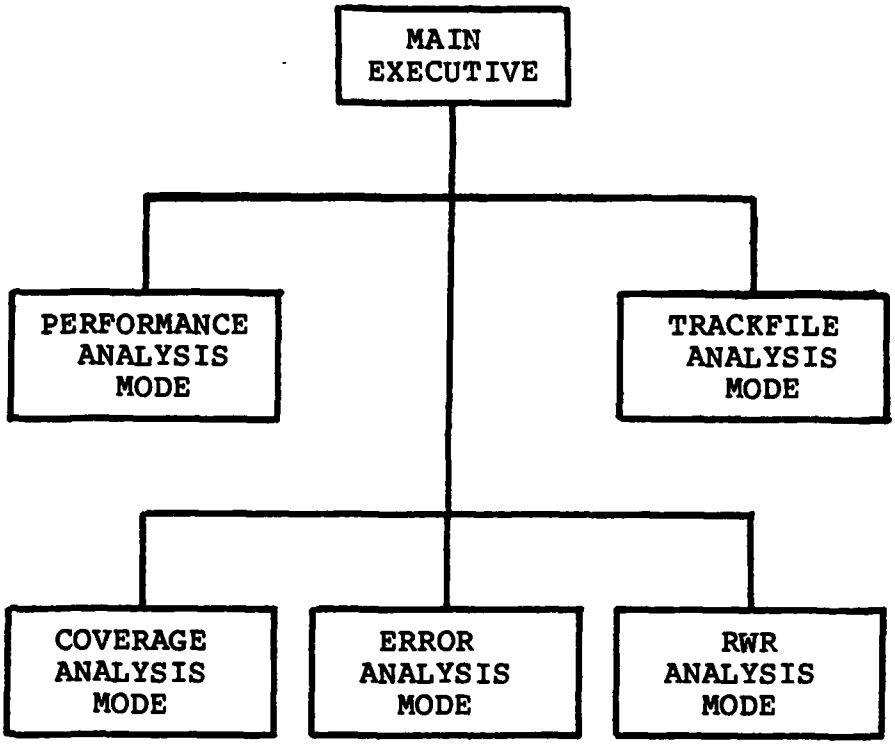


Figure 6. DEAS Function Chart

3.3.1 Performance Analysis Mode - The performance analysis mode marks the time when a subroutine is entered and when the subroutine is exited. The difference in these times is the time spent in execution. The entry and the exit point of each subroutine is entered. The output is a table of execution times for the subroutines.

3.3.2 Coverage Analysis Mode - The coverage analysis mode counts the number of times each of the selected paths are executed. The addresses of instruction to be used for path markers are entered. The output is a table of the paths and number of times executed.

3.3.3 Error Analysis Mode - The error analysis mode checks for "variable out of range" errors. The addresses of variables to be monitored are entered. The output is a table of the variables and their minimum and maximum values.

3.3.4 Trackfile Mode - The trackfile analysis mode displays the contents of the emitter track file. Provisions will be made to include the code for this function which was developed by another study (Ref 22).

3.3.5 RWR Mode - The RWR mode displays a copy of the ALR-46 CRT on a color graphics terminal. Provisions will be made to include the code for this function which was developed by another study (Ref 22).

3.4 SUMMARY.

This chapter has described the software requirements for the Data Extraction and Analysis system. The requirements were derived from the user, environmental, and other requirements described in chapter 2. Briefly the software requirements are as follows:

1. Determine module performance characteristics

2. Determine path traversial frequency

3. Detect variable out of range conditions

4. Provide for previously developed software

The next chapter will describe the system design.

CHAPTER 4

SOFTWARE DESIGN

4.1 INTRODUCTION.

The previous two chapters described the software requirements. This chapter will discuss the design methodology and describe the detailed system design for the Data Extraction and Analysis System (DEAS). The design strategy will be described followed by the techniques used in the design of the DEAS.

4.2 DESIGN STRATEGY.

There are primarily two design strategies for developing computer software. The first is top-down design. In this method the major functions of a system are identified and expressed in terms of lower level functions (Ref 24:322). The process of functional decomposition is repeated until all the subfunctions can be easily implemented. A potential problem in strict top-down design is that there may be no way to ensure that operations at one

level in the hierachy are supportable by some resource to be provided at subordinate levels (Ref 20:21). The second design strategy is bottom-up design. In this method, design is started at the bottom of the hierachy before the design at the top has been completely thought out (Ref 20:5). The "bottom-up" approach can lead to difficulty in integrating system components.

The top-down approach was chosen for the design of the Data Extraction and Analysis System. The danger of having system integration problems with the bottom-up approach out weighs any advantages it might have. The top-down hierachic decomposition leads directly to structured programs.

4.3 DESIGN TECHNIQUES.

Several techniques were considered for the design of the DEAS. The techniques considered were: Structure Charts (Ref 24:25), Data Dictionaries (Ref 24), Data Flow Diagrams (Ref 24), Structured Analysis and Design Techniques (SADT) (Ref 6), and Hierarchical Input-Process-Output (HIPO) (Ref 24). A combination of SADT, Structure Charts, and a Data Dictionary were selected for the system design. SADT shows more detail than Data Flow Diagrams. SADT allows the inclusion of control flow into the diagrams, Data Flow Diagrams do not. Structure Charts show the relationship between the modules more clearly than HIPO. A Data Dictionary is desirable for any design technique.

4.3.1 SADT Diagrams. - To apply SADT to a problem, a model is built which expresses a "complete" understanding of the nature of the problem. SADT breaks a complex subject into its component parts. SADT begins with the most general description of the system, represented by a single box, and breaks that box into a number of more detailed boxes. Each of these boxes is further broken down into more detailed boxes.

The number of detailed boxes, which any parent box is broken into, is limited to a maximum of six and a minimum of three. The upper limit prevents too much detail from being introduced at any one level. The lower limit ensures enough detail to make the decomposition worthwhile.

Each module in a SADT model is represented by a box. The relationship between modules is shown by interconnecting arrows. This box structure is shown in figure 7. When a module is broken down into submodules, the interfaces between them are shown as arrows.

The arrows on the left show input data, which are transformed into output data, shown by the arrows on the right. Controls, represented by the arrows on the top, govern the way the transformation is done. Mechanisms, represented by the arrows on the bottom, indicate the process or device which performs the activity (Ref 6: Sec 4, 5,22).

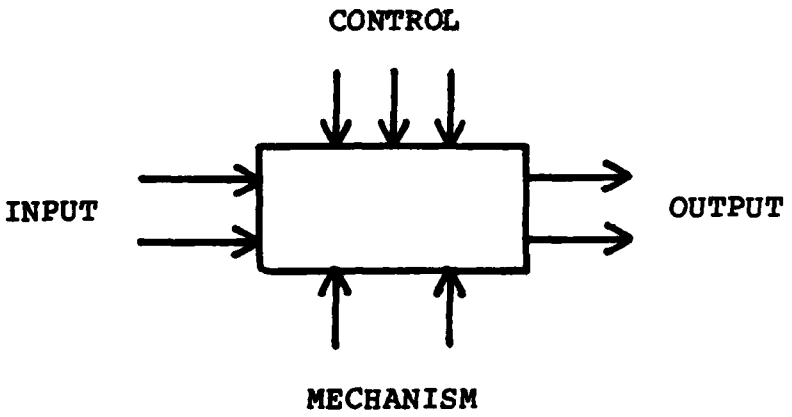


Figure 7. Box/Interface Arrow Definition.

A representative SADT diagram is shown in figure 8. The SADT diagrams developed for the DEAS are contained in Appendix C.

4.3.2 Structure Charts - Structure charts were used to develop the detailed system design. Their use along with top-down design causes the major functions to be developed first (Ref 24:141-147). Figure 9 shows the symbols used in the structure charts. A representative structure chart is shown in figure 10. The structure charts developed for the DEAS are contained in Appendix D.

4.3.3 Data Dictionary - A data dictionary defines all the terms used in system development. A data dictionary entry should contain a concise description of the term, all associated aliases, and the composition of the entry (Ref 24:150-163). The Data Dictionary for the DEAS is contained in Appendix E.

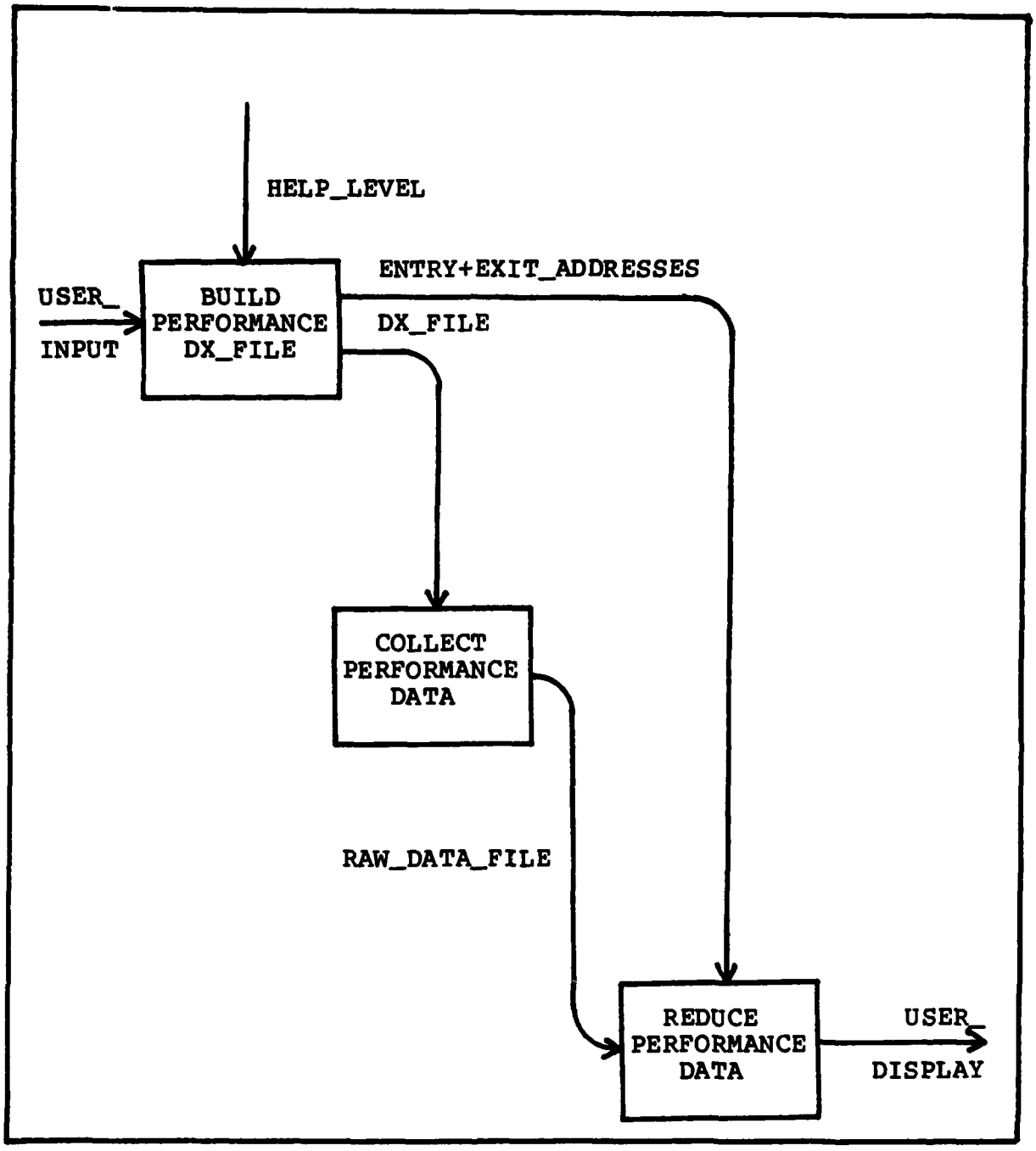


FIGURE 8. SADT Diagram.

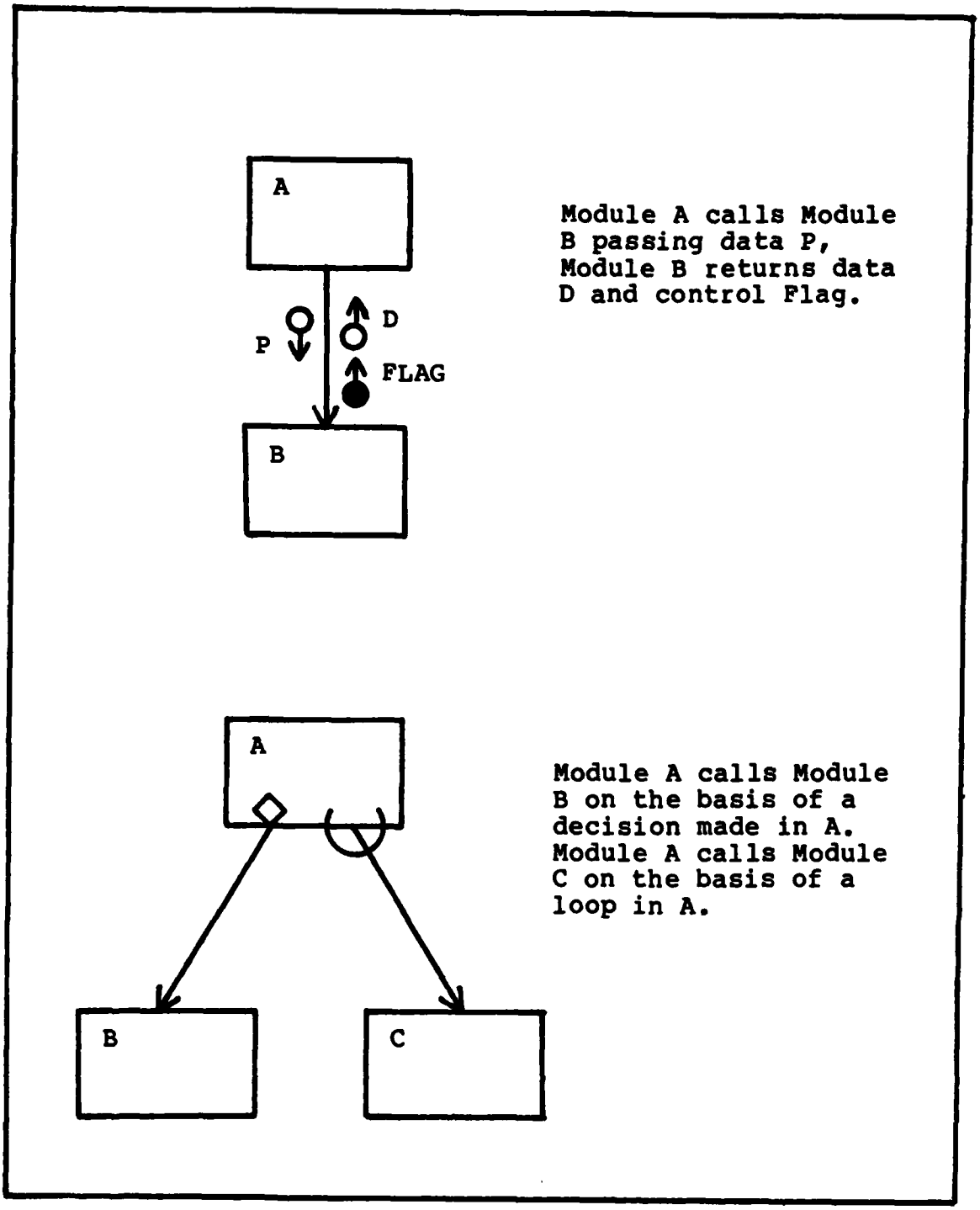


Figure 9. Structure Chart Symbols.

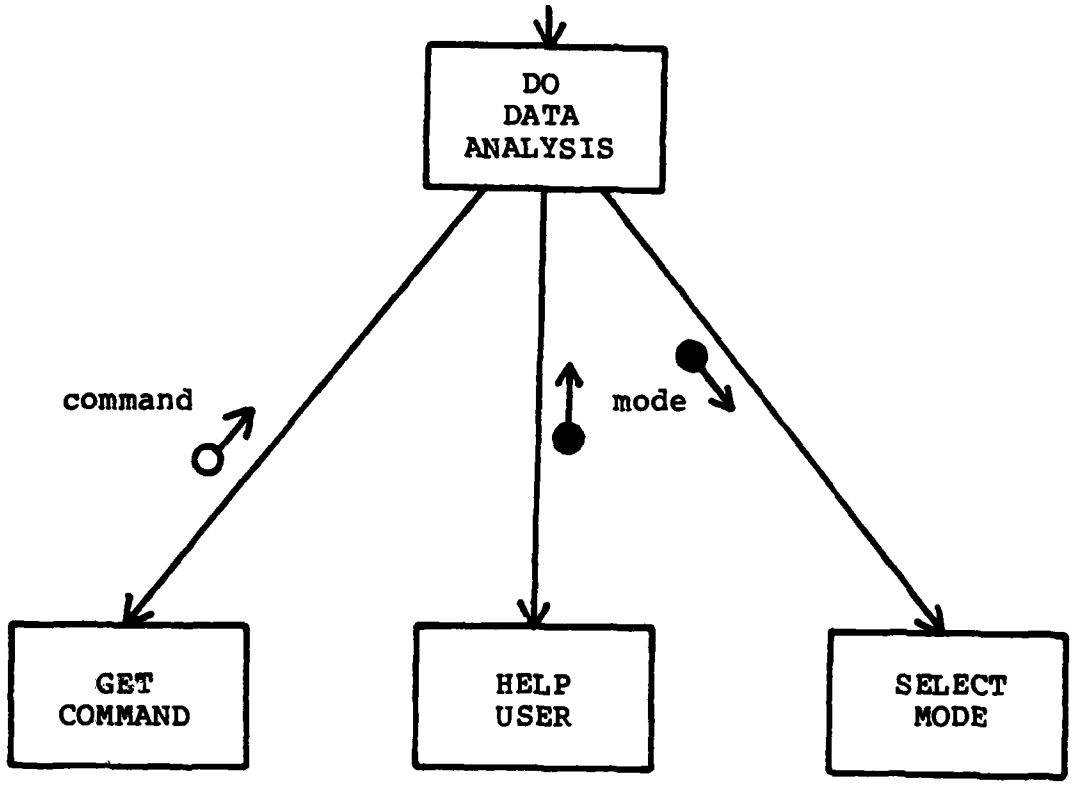


Figure 10. Structure chart.

4.4 SUMMARY.

This chapter described the design strategy and design techniques used in developing the DEAS. The use of SADT diagrams, structure charts, and the data dictionary were described. The SADT diagrams, structure charts, and data dictionary used in the design of the DEAS are contained in Appendices C, D, and E respectively. The next chapter discusses the implementation and testing of the DEAS.

CHAPTER 5

IMPLEMENTATION AND TESTING

5.1 INTRODUCTION.

The implementation of the Data Extraction and Analysis System was done in VAX/11 Pascal (Ref 7). The reasons Pascal was chosen were discussed in Chapter 1. Appendix F contains the source listing for the DEAS. The implementation was accomplished in accordance with the design set forth in Chapter 4. The data structures used will be described followed by a description of each of the functional components of the system. Finally, the testing of the system will be discussed.

5.2 IMPLEMENTATION STRATEGY.

As in design, two strategies were considered. The first is "top-down" implementation. With this strategy the higher level modules are coded first with the next lower level modules replaced with "dummy stubs" for testing. Modules are coded working down until the entire design has

been coded. The second strategy is "bottom-up" implementation. With this strategy modules are coded starting with the lowest level. For testing, the remainder of the system is replaced with a "test harness".

The top-down approach was selected because modules could be tested as they are coded without requiring a test harness for each module. The top-down approach minimizes interfacing problems by defining the interfaces in the higher level modules.

5.3 IMPLEMENTATION DETAILS.

5.3.1 Data Structures - Consideration was given to three types of data structures for temporary storage of information: the array, the linked list, and the internal file. The array is the simplest data structure. The size of an array is fixed regardless of the amount of information stored in it. The size of an array must be set to the maximum amount of data to be stored. The handling of a linked list is more complex than an array, but the storage is allocated dynamically. The internal file is more complex to handle than a linked list but can store large quantities of data. The linked list was chosen for temporary data storage because storage space can be allocated dynamically and the expected amount of storage required did not justify the extra complexity of internal files.

5.3.2 Main Executive - The main executive accepts commands from the terminal. The complete list of valid commands is shown in Table 1. A menu is printed if "help" was entered, if a valid command was entered one of the five modes of operation is started.

The user cannot cause the program to abnormally terminate by typing invalid data. User input is accepted in an array of characters (string array). Command inputs are compared against valid commands until a match is found or the entire list has been searched. Data inputs are checked for invalid characters then, if necessary, converted into the appropriate data type. Upper or lower case characters are acceptable as input since lower case characters are converted internally to upper case.

Table 1.

Valid Commands.

HELP	-	select from menu
PERFORMANCE	-	performance analysis
COVERAGE	-	coverage analysis
ERROR	-	error condition detection
TRACK	-	trackfile analysis
RWR	-	RWR color graphics display
QUIT	-	exit program

5.3.3 Subroutine Execution Time Calculation - When this mode is executed the user is instructed to enter a name, used to identify the subroutine, and its entry and exit addresses. This mode builds an "event" type data extraction file, as described in Appendix B. When the data for all the subroutines have been entered the DEAS will begin to extract the specified data.

After data collection is terminated the DEAS will calculate the execution time of each subroutine call. Two stacks are implemented for each subroutine. One stack stores the entry time for each subroutine call and the other stack stores the calculated execution time for each call. Each time an event record is read, the event address is compared with the entry address and the exit address for each subroutine. If a match is found with an entry address, the entry time is pushed on the "entry time" stack. If a match is found with the exit address, the entry time is popped off the "entry time" stack and subtracted from the exit time to obtain the execution time. The execution time is then pushed on the "execution time" stack. When all data in the "raw data file" has been analyzed, the list of subroutines will be searched and the execution time of each call will be popped from the "execution time" stack and displayed on the console.

5.3.4 Coverage Analysis - When this mode is executed the user is instructed to enter an address to serve as a path marker for each path to be monitored. The coverage analysis mode builds a data extraction file to collect "event" data. The format for this file is found in Appendix B. The extracted data is reduced by inserting it into a binary tree. A count is maintained of the number of traversals of each path. An "in-order" traversal of the tree is made and the paths traversed are printed out along with the number of traversals of each path.

5.3.5 Error Analysis - When this mode is entered the user is instructed to enter the address of each variable to be monitored. The error analysis mode builds a data extraction file to collect "location" data. The format for this file is found in Appendix B. The extracted data is reduced by inserting it into a binary tree, ordered by address value. The minimum and the maximum value attained by the variable is inserted into the binary tree. An "in-order" traversal is made of this tree also. The addresses of extracted variables, with their minimum and maximum values, are printed out in ascending order.

5.4 TESTING.

Two levels of testing were done, white box and black box. "White box testing" is done using knowledge of the internal structure of the code. "Black box testing" uses

the functional requirements and the system specification to evaluate the performance of a system.

5.4.1 White Box Testing - White box testing started when the first module was coded. "Dummy stubs" replaced the modules which had not yet been coded. The primary method of white box testing was path analysis. This testing technique attempts to execute each decision to decision path.

5.4.2 Black Box Testing - After the entire DEAS system was coded black box testing was done. Black box testing was done using equivalence class testing and boundary value analysis.

5.4.2.1 Equivalence Partitioning - Equivalence classes are identified by taking each input condition and dividing it into several groups (Ref 15:44-50). There are two types of equivalence classes. The first type is "valid equivalence classes" which represents valid inputs to the program. The second type is "invalid equivalence classes" which represent all other possible inputs. A set of guidelines for identifying equivalence classes is:

1. If an input condition specifies a range of values, choose one valid equivalence class and two invalid equivalence classes.

2. If an input condition specifies the number of values, choose one valid equivalence class and two invalid equivalence classes.

3. If an input condition specifies a set of input values and if each input value is handled differently by the program, identify a valid equivalence class for each input condition and one invalid equivalence class.

4. If an input condition specifies a "must be" condition, identify one valid equivalence class and one invalid equivalence class.

5. If it is suspected that all elements in an equivalence class are not handled identically by the program, split the equivalence class into several smaller equivalence classes.

Several examples of equivalence partitioning are shown in Table 2.

5.4.2.2 Boundary-value Analysis - Boundary conditions are conditions which are directly or, above, or below the edges of input and output equivalence classes (Ref 15:50-55). Boundary-value analysis differs from equivalence partitioning in two respects. First, boundary-value analysis requires that elements selected from an equivalence class be at the edge of the class rather than any element in the class. Second, output and input conditions are

Table 2.
Equivalence Classes.

INPUT CONDITION	VALID EQUIVALENCE CLASSES	INVALID EQUIVALENCE CLASSES
COMMAND	PERFORMANCE COVERAGE ERROR TRACK RWR QUIT HELP	ANY OTHER WORD
OCTAL ADDRESS	0..32767	> 32767 < 0 NON OCTAL DIGIT
MENU INPUT	1..6	> 6 < 1

considered, rather than only the input conditions.

5.4.3 System Testing - The specific tests used in validating proper system performance are contained in Appendix G. The test cases were chosen by applying equivalence class analysis and boundry value analysis to the system requirements. The expected response was obtained in all test cases.

5.5 SUMMARY.

This chapter described the implementation and testing of the Data Extraction and Analysis System. The design described in Chapter 4 was followed in the implementation of the DEAS. The language used in the implementation of the

DEAS was Pascal. The main data structures and the major program sections were described. The DEAS was fully tested according to the test plans in Appendix G. No known errors exist in the program.

CHAPTER 6

RESULTS AND RECOMMENDATIONS

6.1 RESULTS.

This study investigated the requirements for automated tools for test and analysis of embedded computer software. The user requirements along with external requirements were considered in establishing the requirements for the Data Extraction and Analysis System.

Once the requirements for the DEAS were established system design began. A system model was produced using SADT and the detailed system design was done using structure charts combined with a data dictionary.

The DEAS was implemented using Pascal on the VAX 11/780 computer. The highly modular, structured design minimized interfacing difficulties during implementation.

Both black box and white box testing was done. White box testing was done as the modules were developed. Path analysis was the primary technique used during white box testing. Black box testing was done after the entire system was developed. The techniques used for black box testing were equivalence partitioning and boundary value analysis.

Installation and integration of the DEAS with the ISS was not possible because of time constraints. No problems are anticipated with installation since DEC Standard Pascal was used and system calls were minimized.

6.2 RECOMMENDATIONS.

There are two user requirements, identified during this study, which require further study.

The capability to generate input buffers could loosely be thought of as a peripheral simulator. The simulator would replace the RF and A/D portion of the system. This would enable the exact duplication of conditions between tests. this would also allow the simulation of signals beyond the capabilities of the threat generators.

The Automated generation of test data would aid the engineer in selecting good test data. The generation of test data for the EID would be simpler than for the OFP, because the EID is constructed using rigid rules.

Bibliography

1. 62F026000. AN/ALR-46(V) Integration Support System Software Documentation. Buffalo NY: Comptek Research Inc., 1981.

2. 62U026003. AN/ALR-46(V) Integration Support System User Guide. Buffalo NY: Comptek Research Inc., 1981.

3. 62U026007. CM-442 Adapter Technical Manual. Buffalo NY: Comptek Research Inc., 1982.

4. 62U026008. CM-442A Adapter Technical Manual. Buffalo NY: Comptek Research Inc., 1982.

5. 62U026009. DX Interface Technical Manual. Buffalo NY: Comptek Research Inc., 1982.

6. 9022-78R. An Introduction to SADT Structured Analysis and Design Technique. Waltham, Massachusetts: SofTech, Inc., 1976.

7. AA-H485A-TE. VAX-11 PASCAL User's GUIDE. Maynard, Massachusetts: Digital Equipment Corporation, 1979.

8. Bibbens, Terry E. "EW the Unique Weapon," Signal. Falls Church, Virginia: Armed Forces Communication and Electronics Association, 18-22 (March 1981).

9. Fairley, Richard E. "ALADDIN: Assembly Language Assertion Driven Debugging Interpreter," IEEE Transactions on Software Engineering, Vol. SE-5, No. 4: 426-428 (July 1979).

10. Howden, William E. "A Survey of Dynamic Analysis Methods," Tutorial: Software Testing & Validation Techniques, edited by Edward Miller and William E. Howden. 184-206. New York, N. Y.: Institute of Electrical and Electronics Engineers, 1978.

11. Howden, William E. "A Survey of Static Analysis Methods," Tutorial: Software Testing & Validation Techniques, edited by Edward Miller and William E. Howden. 82-96. New York, N. Y.: Institute of Electrical and Electronics Engineers, 1978.

12. Mitchel, William B. Digital Simulation as an EW Software Maintence Tool. Presented at Electronic Warfare Symposium VI, Robins AFB, Georgia: March 1981

13. Mitchal, William B. and Gary W. Little. "Performance Analysis of Electronic Warfare Systems Software," Presented at the National Aerospace Electronics Conference, Dayton, Ohio: May 1978.

14. MMROI 800-01. Software Change Processing/Configuration Management for EW Systems. Robins AFB, Georgia: Warner Robins Air Logistics Center, Directorate of Material Management, Electronic Warfare Division, 1979.

15. Myers, Glenford J. The Art of Software Testing. New York, NY: John Wiley & Sons, 1979.

16. R-3636-9234. The ALR-46A System Description and Related Support Functions. Belmont, California: Dalmo Victor Corporation, 1978.

17. Ramamoorthy, C. V., et al. "Optimal Placement of Software Monitors Aiding Systematic Testing," IEEE Transactions on Software Engineering, VOL. SE-1, NO. 4: 403-411 (December 1975).

18. Reifer, Donald J. and Stephen Trattner. "A Glossary of Software Tools and Techniques," Computer, 52-59 (July 1977).

19. Simpson, Henry. "A Human-Factors Style Guide for Program Design", BYTE, 7: 108-132 (April 1982).

20. Tausworthe, Robert C. Standardized Development of Computer Software, Part I, Methods. Washington D.C.: U.S. Government Printing Office, 1976.

21. Tausworthe, Robert C. Standardized Development of Computer Software, Part II, Standards. Washington D.C.: U.S. Government Printing Office, 1978.

22. Thames, J. Wayne. The ALR-46 Computer Graphics System for the Robins AFB Electronic Warfare Division Engineering Branch Laboratory. MS Thesis. Wright Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, December 1981.

23. The International Countermeasures Handbook (Second Edition), edited by Harry F. Eustance. Palo Alto, Californa: EW Communications Inc., 1976.

24. Weinberg, Victor. Structured Analysis. New York, New York: Yourdon Press, 1980.

25. Yourdon, Edward and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Englewood Cliffs, New Jersy: Prentice Hall, Inc., 1978.

APPENDIX A

INDEX OF AVAILABLE SOFTWARE TOOLS

This appendix contains four software tools identified in this study which are directly applicable for testing the ALR-46 flight software. The name of the tool is given along with its source and description. The tools are divided into two groups, those used for static analysis and those used for dynamic analysis.

A.1 STATIC ANALYSIS TOOLS.

1. NAME: EID Tools

   SOURCE : Comtek Reasearch Inc.

   DESCRIPTION : Examines the EID database for violations of structure rules.

2. NAME: SCOM

   SOURCE: Data General Corp.

   DESCRIPTION: Utility which is part of the Advanced Operating System (AOS). Identifies the differences between two source or text files. If the program finds differences, it outputs either a message or the differences. The program will then attempt to resynchronize.

A.2 DYNAMIC ANALYSIS TOOLS.

1. NAME: ALLADIN

   SOURCE: Richard E. Fairley

   DESCRIPTION: Assembly Language Assertion Driven Debugging Interpreter.

2. NAME: RWR Simulator

SOURCE: Comtek Research Inc.

DESCRIPTION: Simulator for the ALR-46 Radar Warning Receiver. The RWR Simulator consists of three components: an Environment Simulator, a Receiver Simulator, and the operational flight software.

APPENDIX B

DATA EXTRACTION FILE FORMATS

This appendix describes the formats for the event/location file and the extracted data file. The file required by the data extraction subsystem is described first. The file produced by the data extraction subsystem is described second.

CONTENTS

B.1 EVENT/LOCATION SPECIFICATION FILE.

This file contains addresses which specify "event" locations, and "location" address limits. An "event" occurs whenever the data extraction subsystem detects a memory "fetch" on a memory address flagged for extraction. A "location" occurs whenever the data extraction subsystem detects a "memory write" to a address flagged for extraction. The format of the file is as follows (Ref 2: Sec 4, 27):

```
address of event 1
address of event 2
      .
      .
      .
address of event n
0
lower address - upper address of first contiguous data block
lower address - upper address of second contiguous data block
      .
      .
      .
lower address - upper address of final contiguous data block
```

The event addresses are entered first, one per line, and are terminated by a line containing a zero. The zero terminator must be present even if no event addresses are specified. Event addresses are specified by a right-justified, five-digit, octal number.

The data location address limits are entered following the event addresses. The data address location address limits consist of a lower address limit, followed by a space, comma, or a hyphen, then followed by the upper

address limit. One pair of address limits are entered per line. The address limits are right-justified, five-digit, octal numbers as required by the event addresses. If a data block is only one word long, both addresses are set equal to the address of the word. The same address cannot appear as both an event address and be contained a data block defined by a location pair (Ref 2: Sec 4, 27-29). An example of an acceptable event/location file follows:

```
00537
  745
01243
0
07775 10003
20105,20105
22351-23200
```

B.2 EXTRACTED DATA FILE.

The data extracted by a "location" is the address extracted and the value written to that address. The data extracted by an "event" is a -1 marker, followed by its address, followed by the high and low 16 bits of the clock. The formats of the two data extraction sequences are (Ref 5: Sec 3, 9):

1. Location

   ADDRESS
   DATA

2. Event

   -1 MARKER

ADDRESS
CLOCK HIGH (MSBs)
CLOCK LOW (LSBs)

The 32-bit clock free runs at 1MHz, providing a 1 usec LSD (Ref 5: Sec 3, 6).

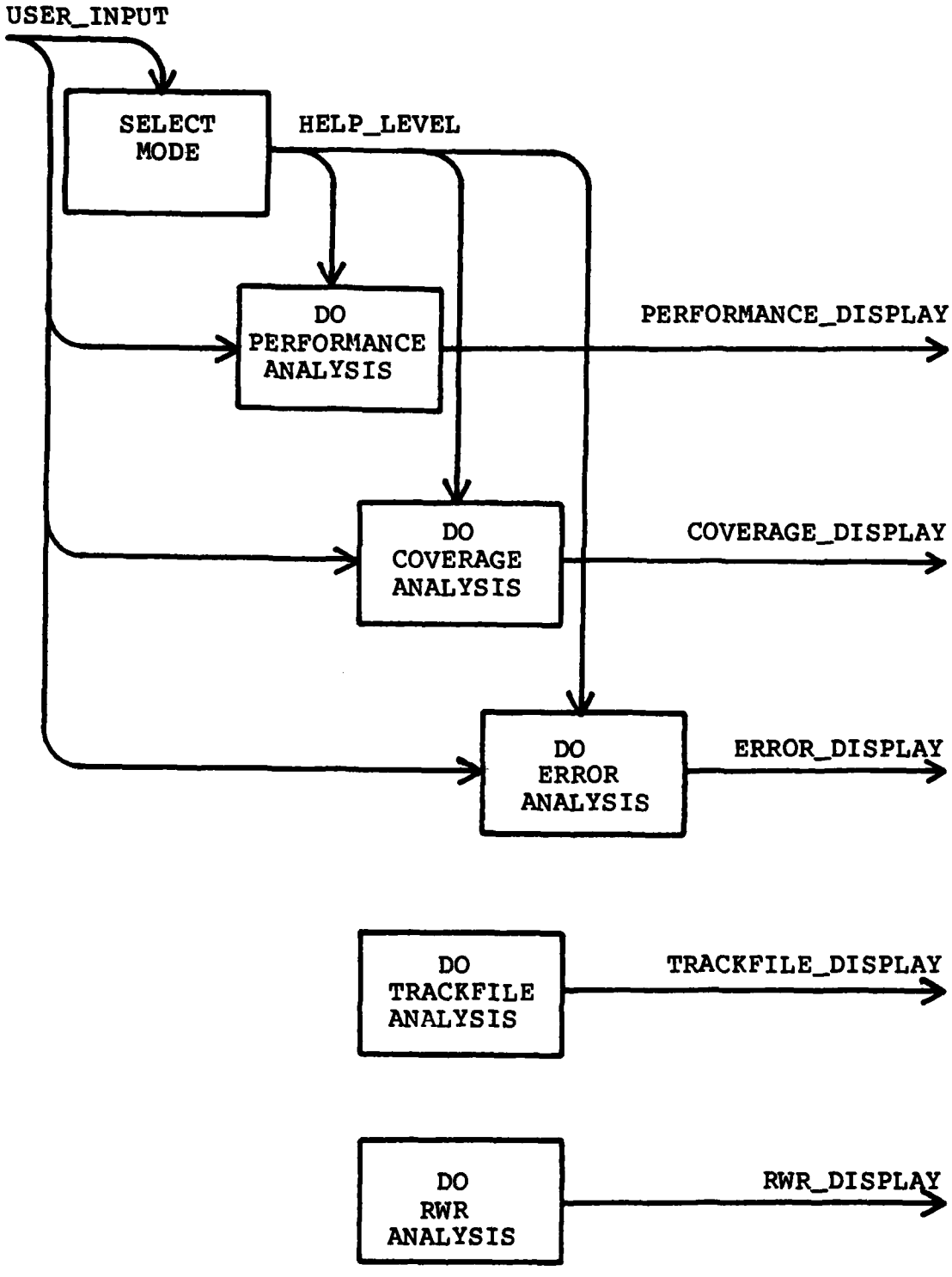
APPENDIX C

SADT DIAGRAMS

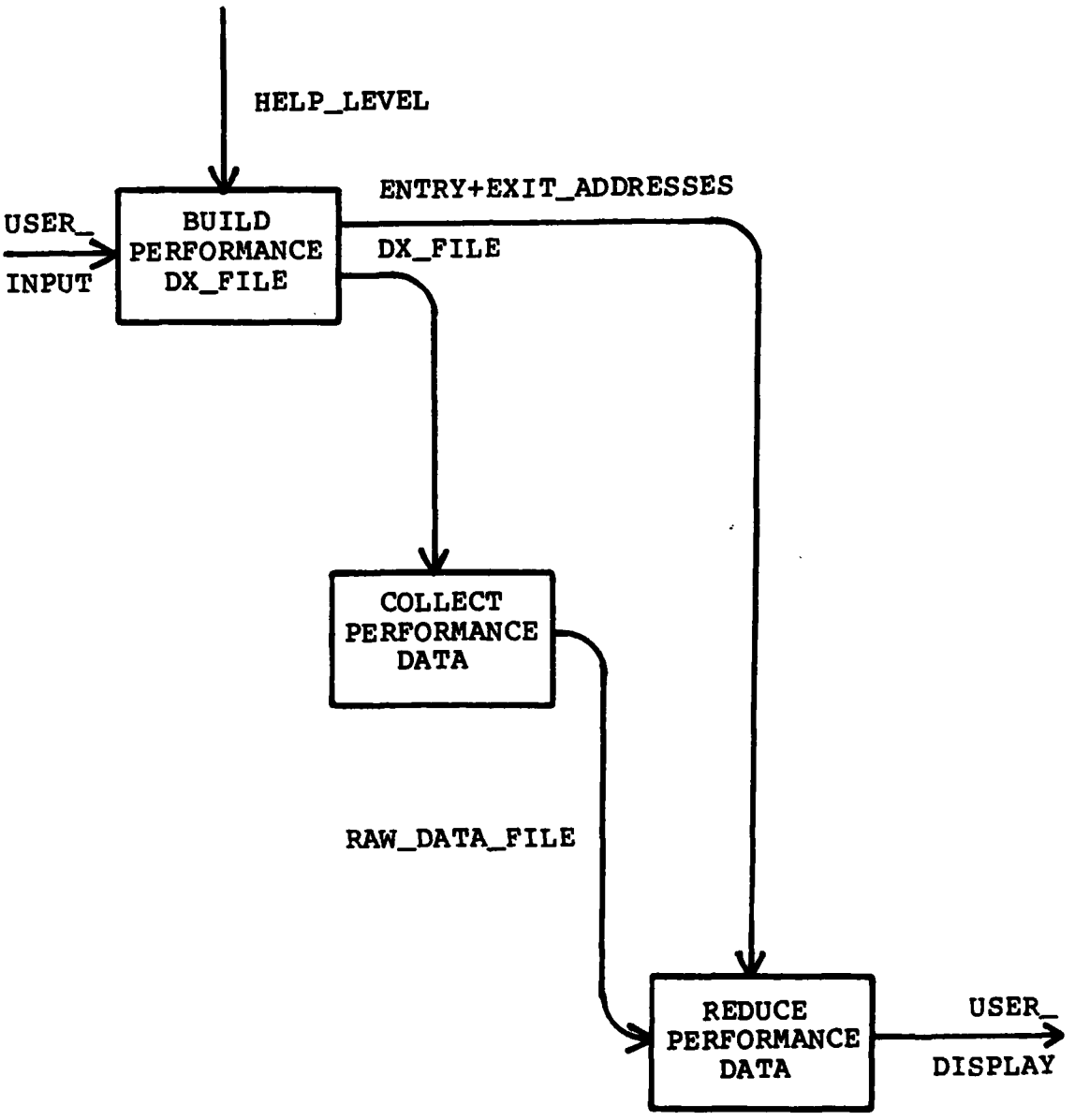
This appendix contains the Structured Analysis and Design Technique (SADT) diagrams used in developing the software requirements and data flow for the Data Extraction and Analysis System.

CONTENTS

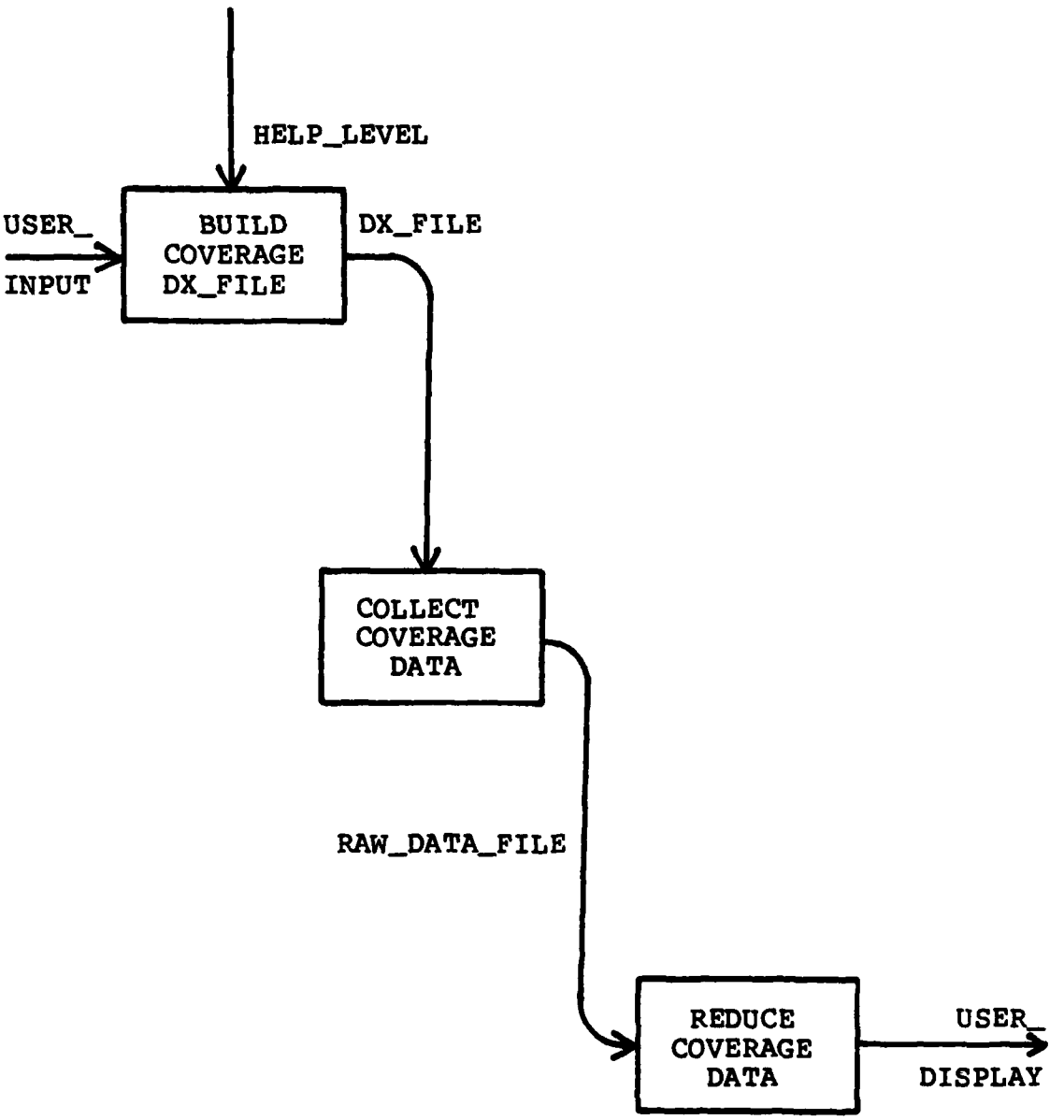
C.1 DATA_ANALYSIS



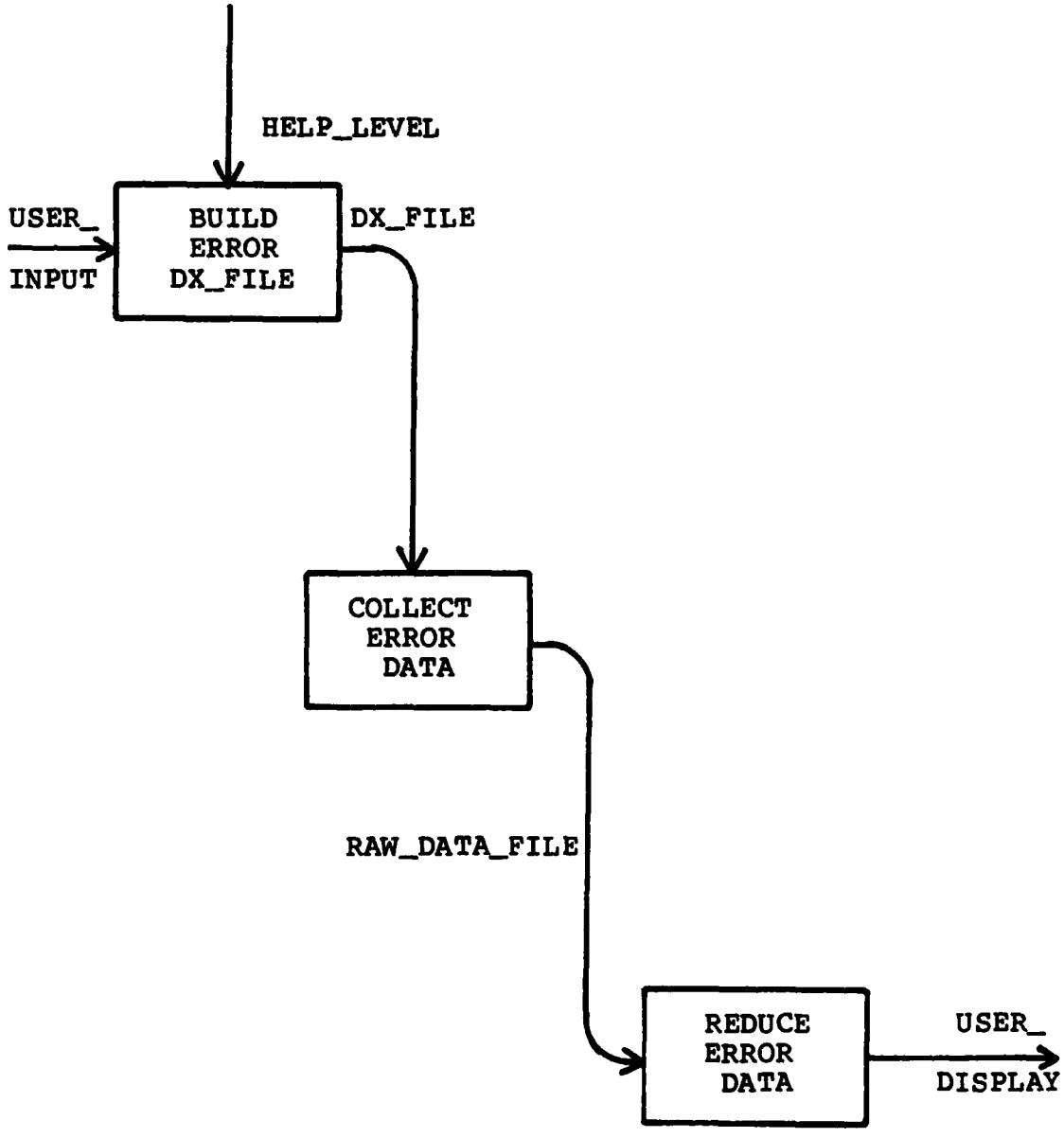
C.2 PERFORMANCE_ANALYSIS



C.3 COVERAGE_ANALYSIS



C.4 ERROR_ANALYSIS



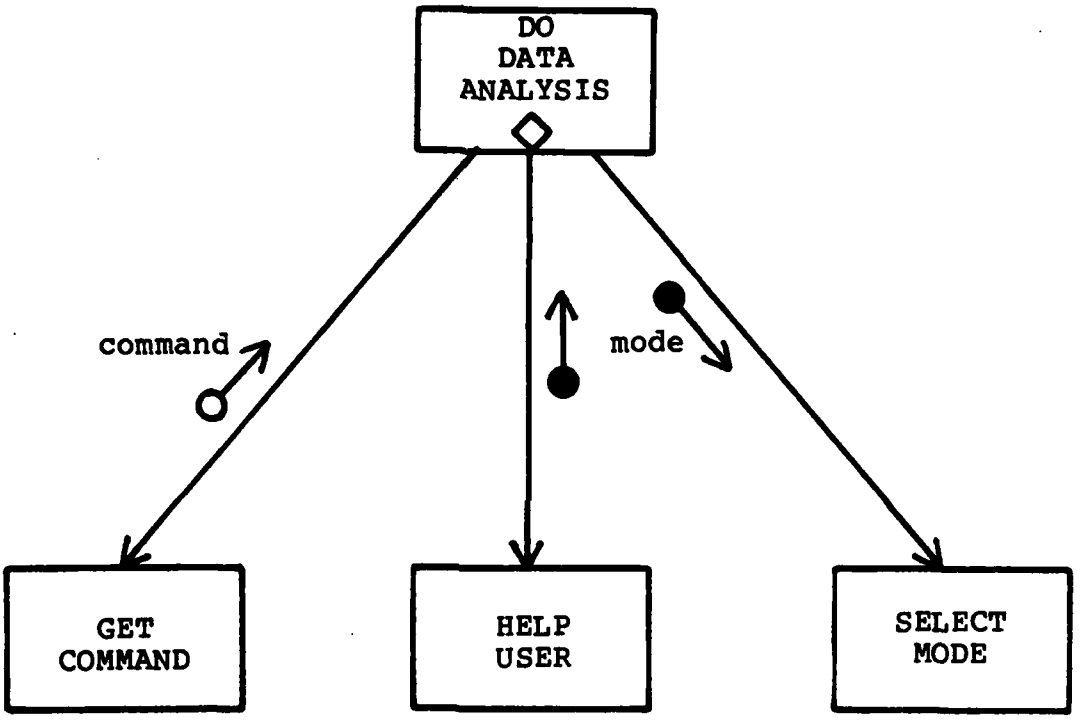
APPENDIX D

STRUCTURE CHARTS

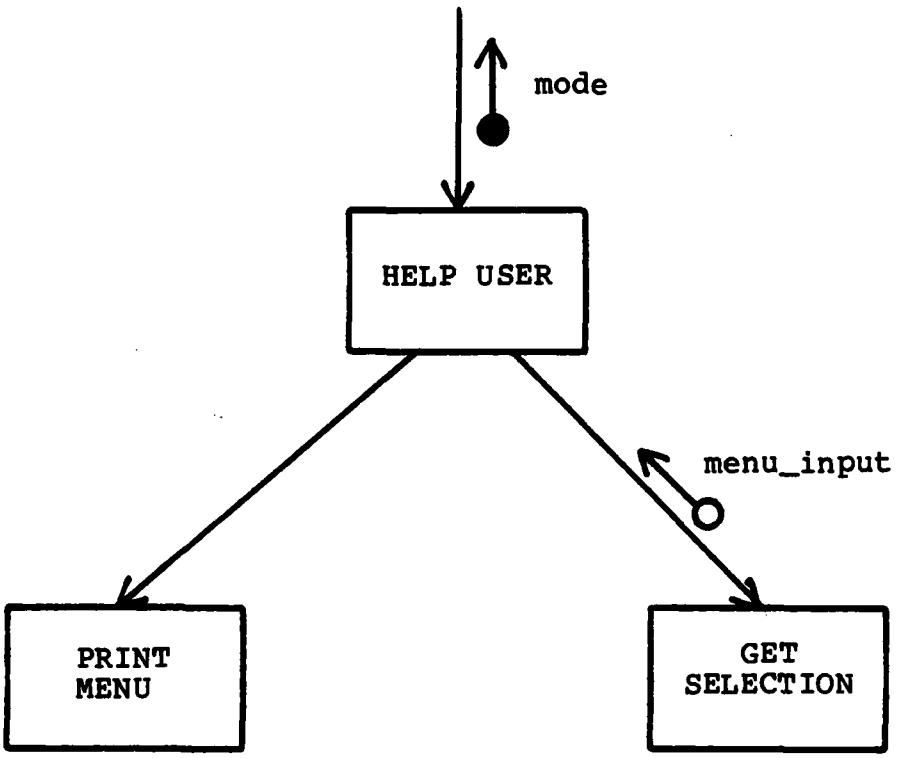
This appendix contains the structure charts used in the detailed design of the Data Extraction and Analysis System.

CONTENTS

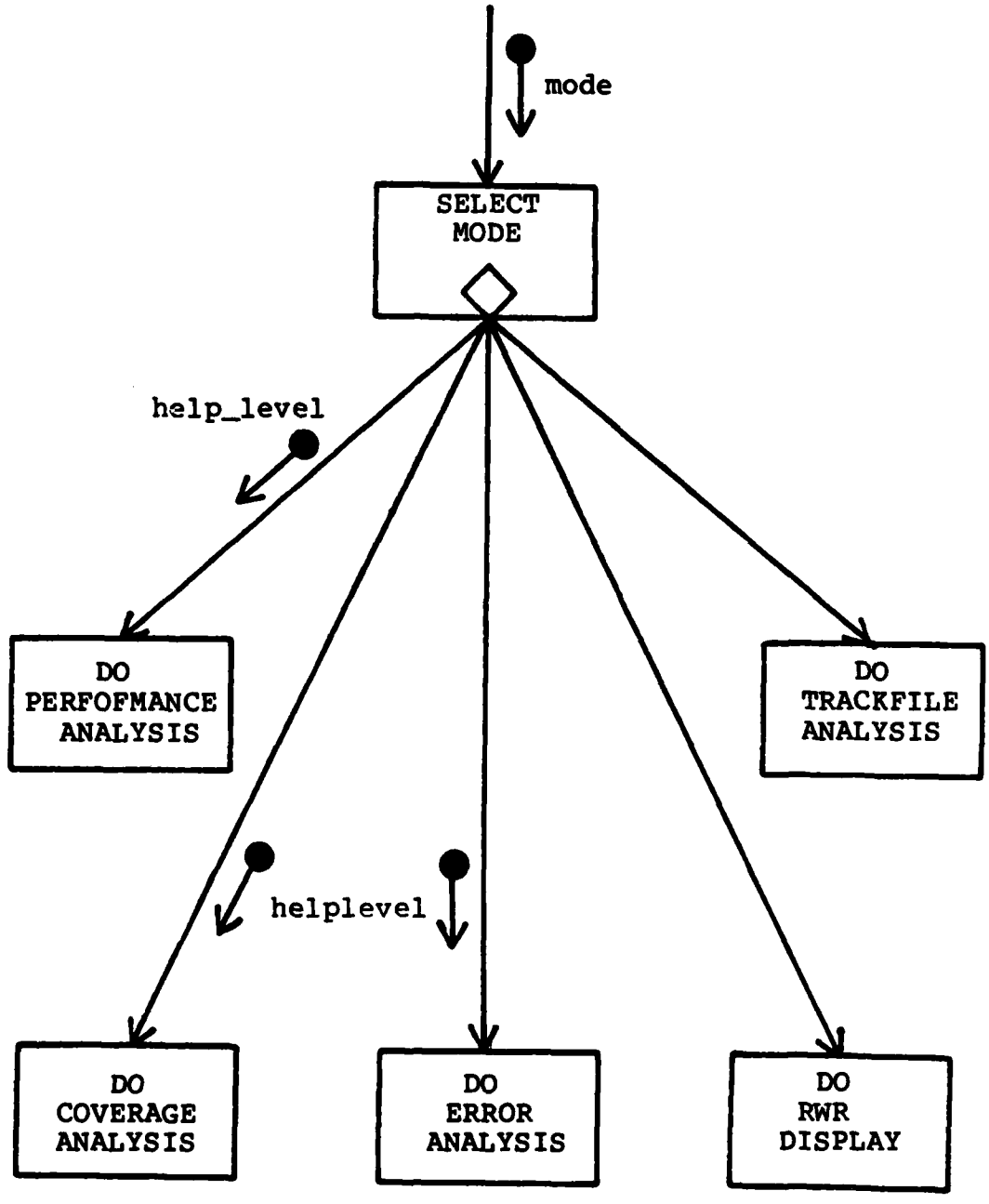
D.1 DATA ANALYSIS



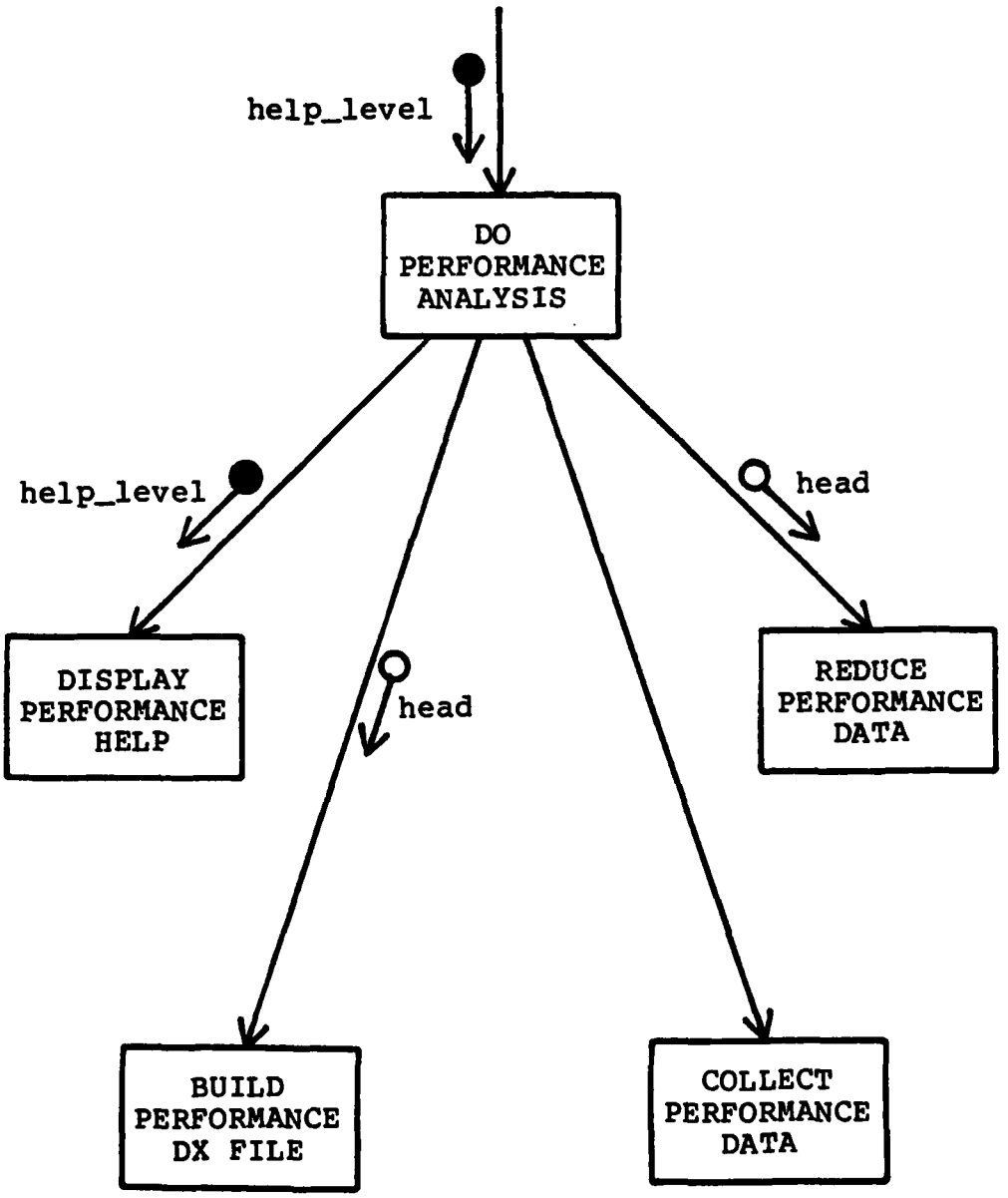
D.1.1 Help User -



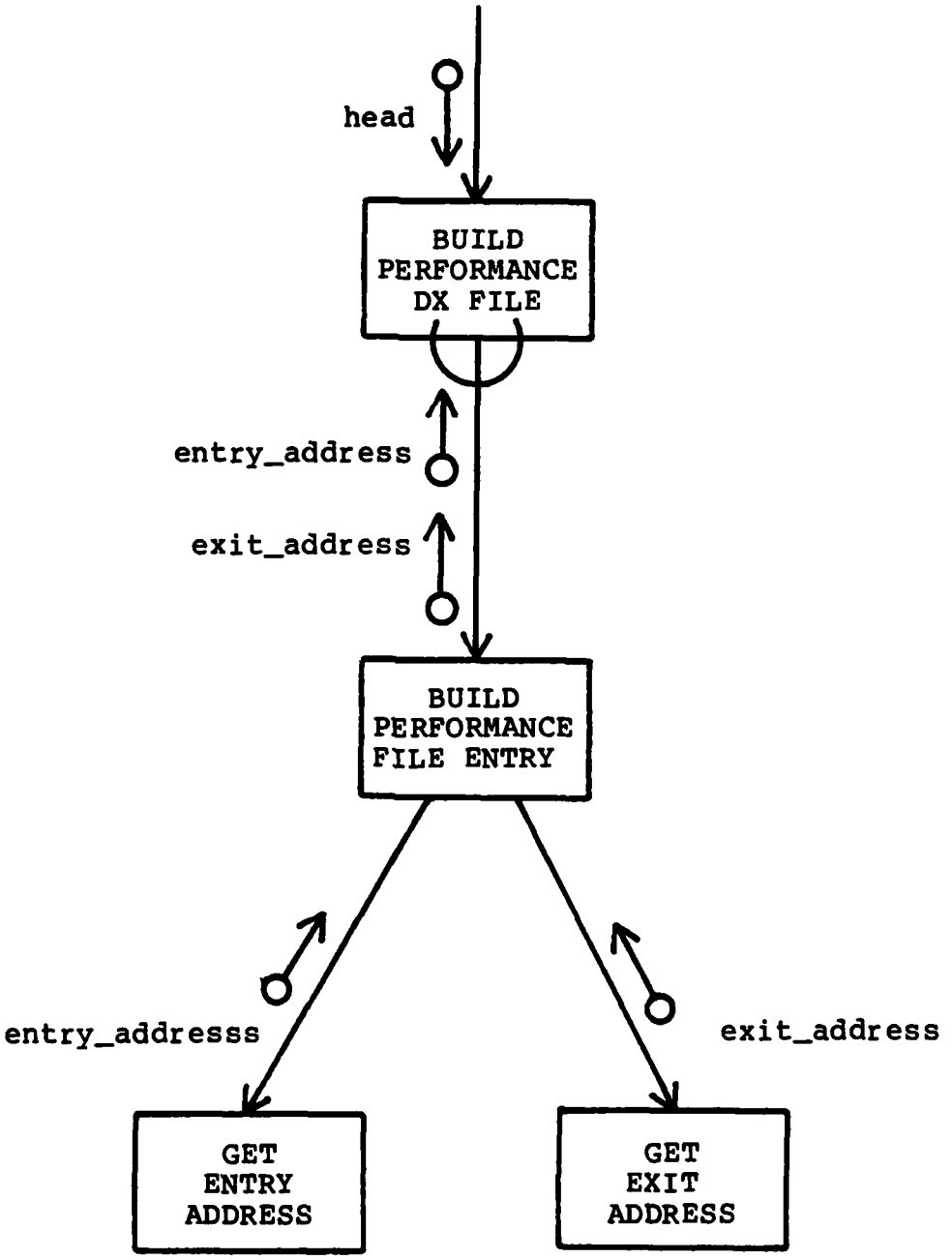
D.1.2 Select Mode -



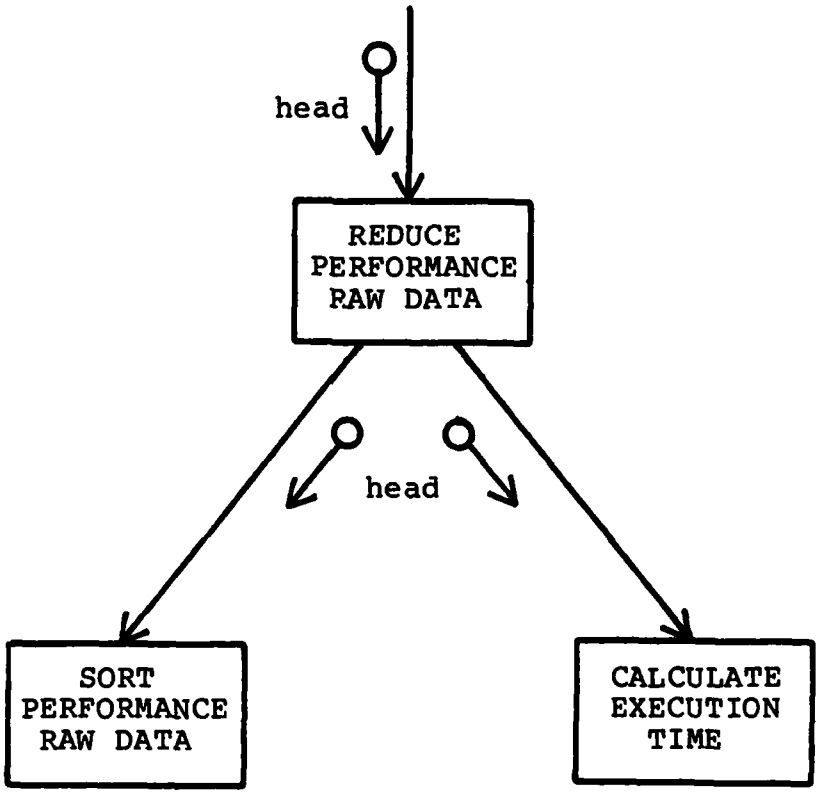
D.1.3 Do Performance Analysis -



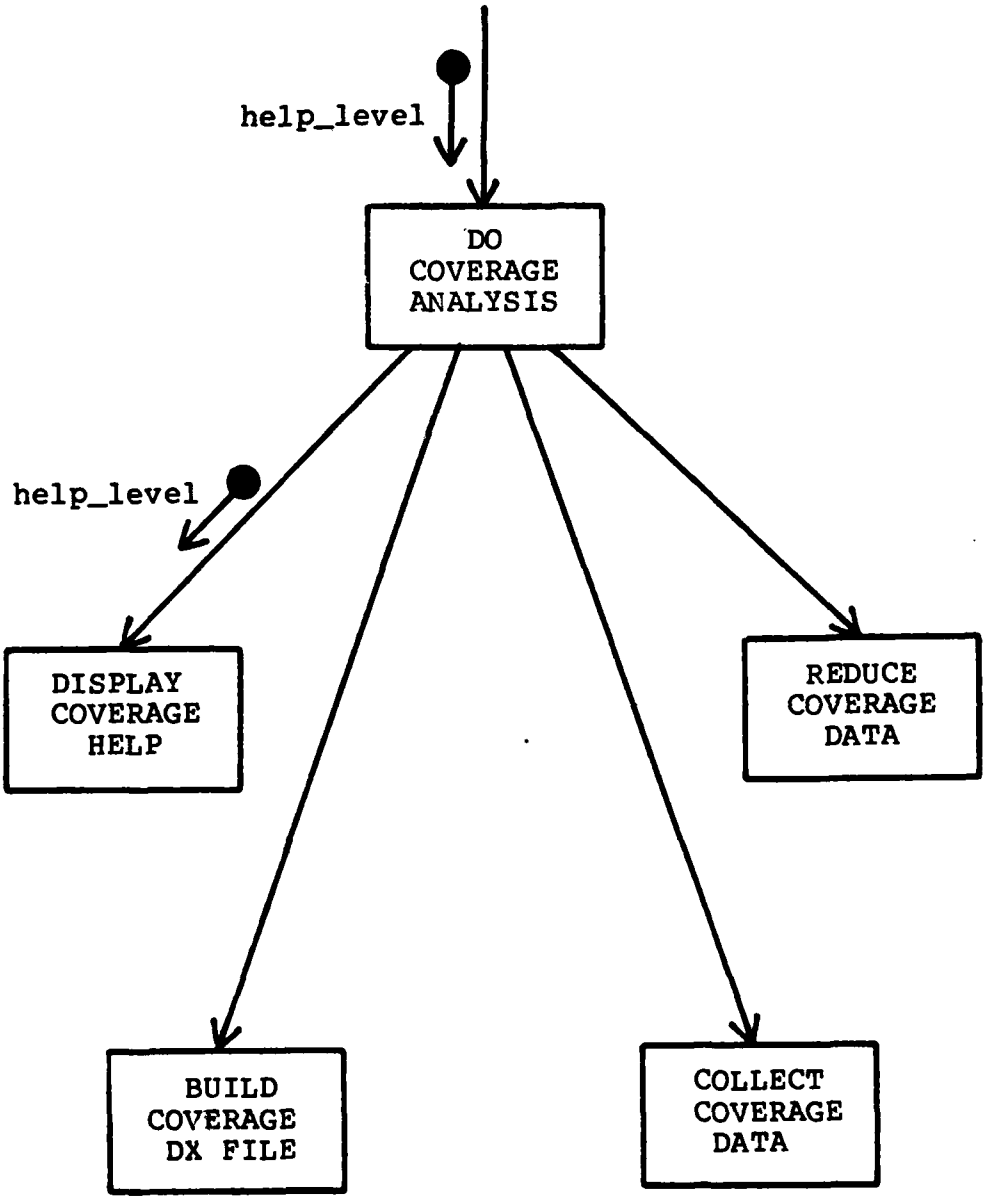
D.1.3.1 Build Performance Dx file -



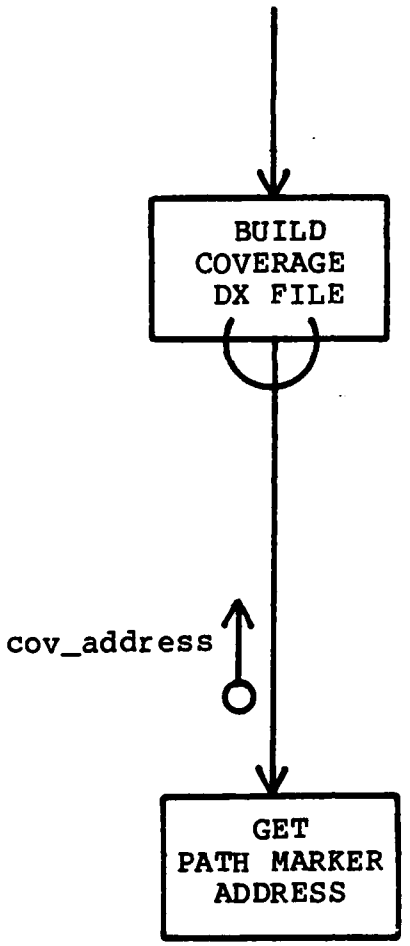
D.1.3.2 Reduce Performance Raw Data -



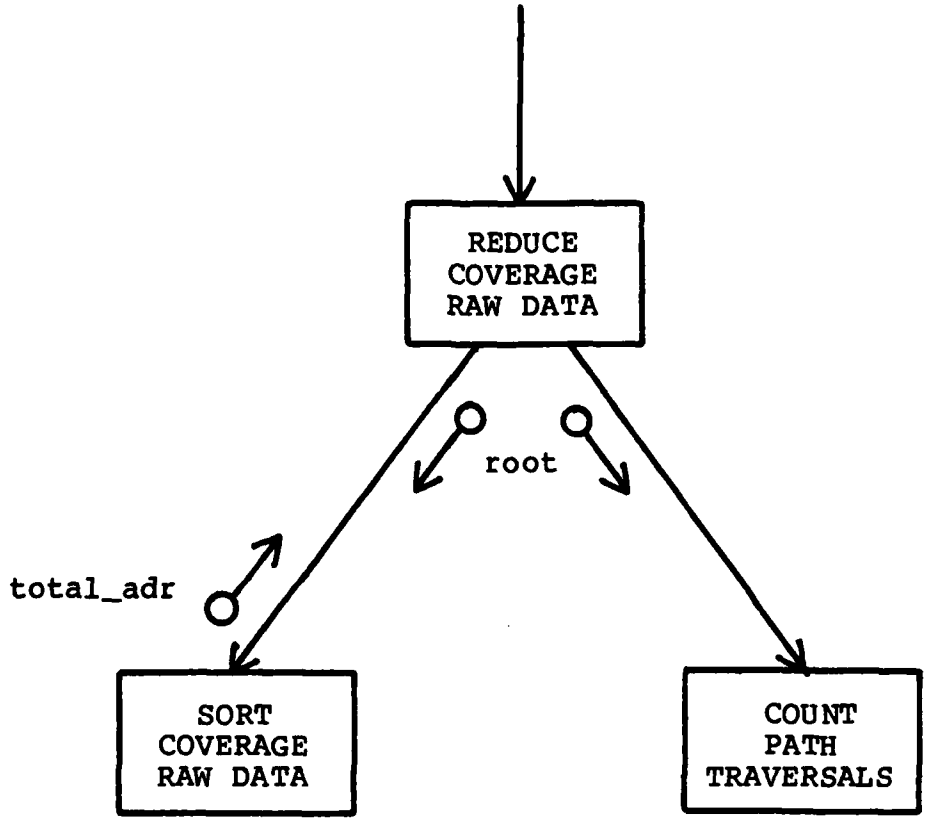
D.1.4 Do Coverage Analysis -



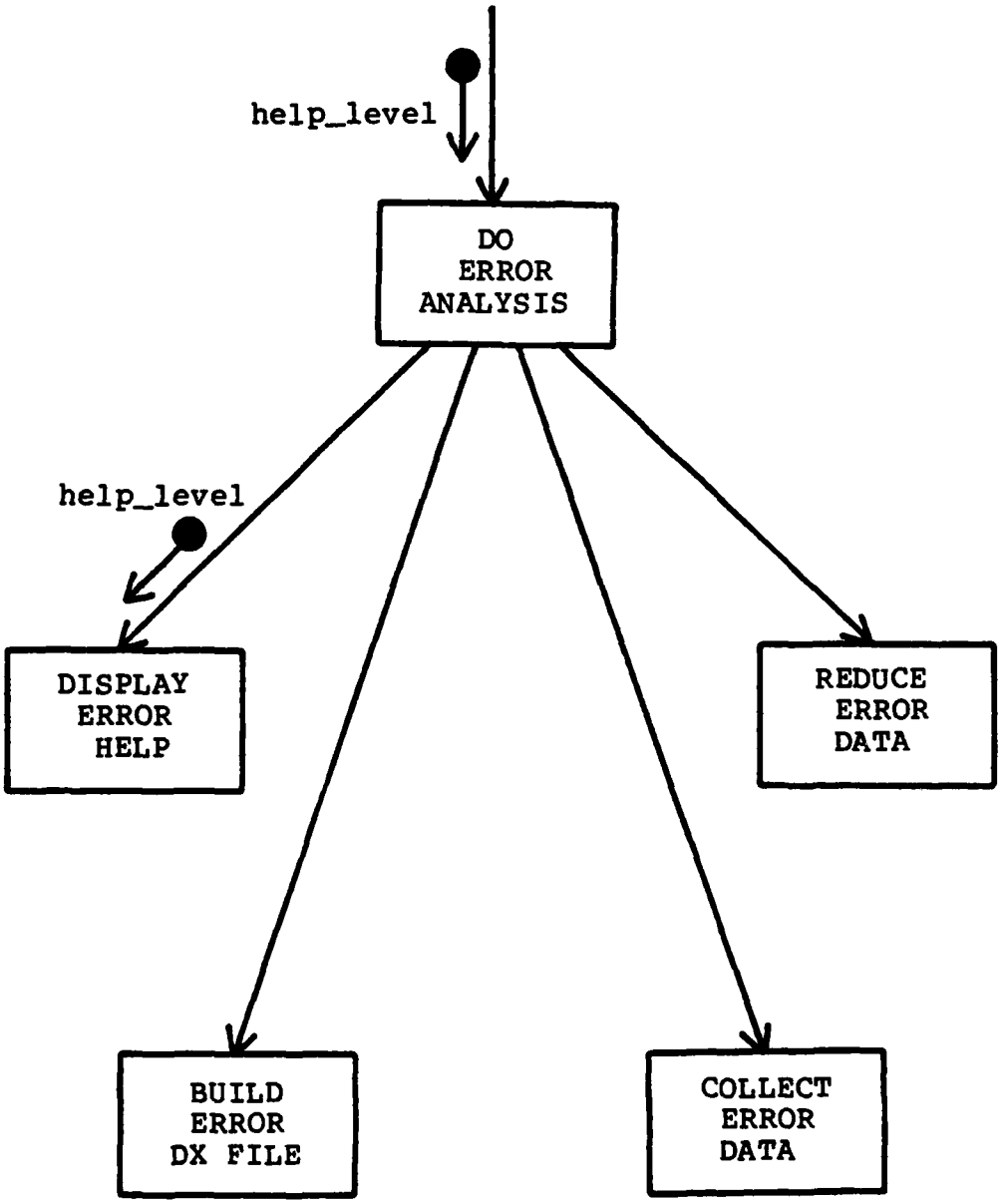
D.1.4.1 Build Coverage Dx file -



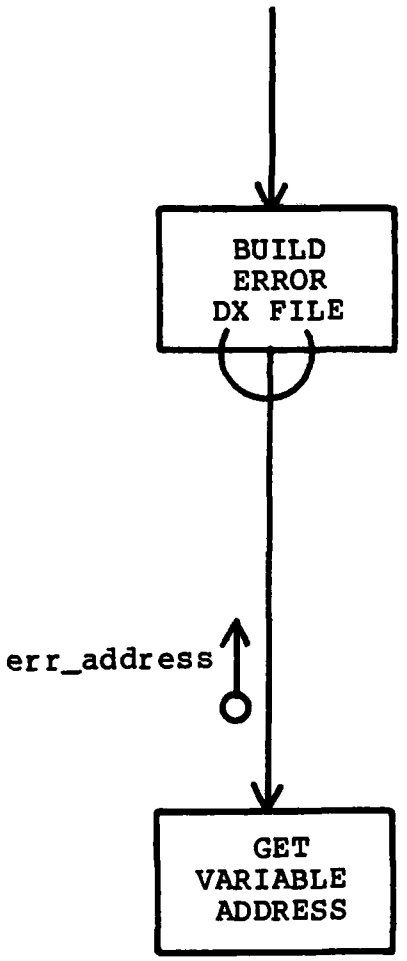
D.1.4.2 Reduce Coverage Raw Data -



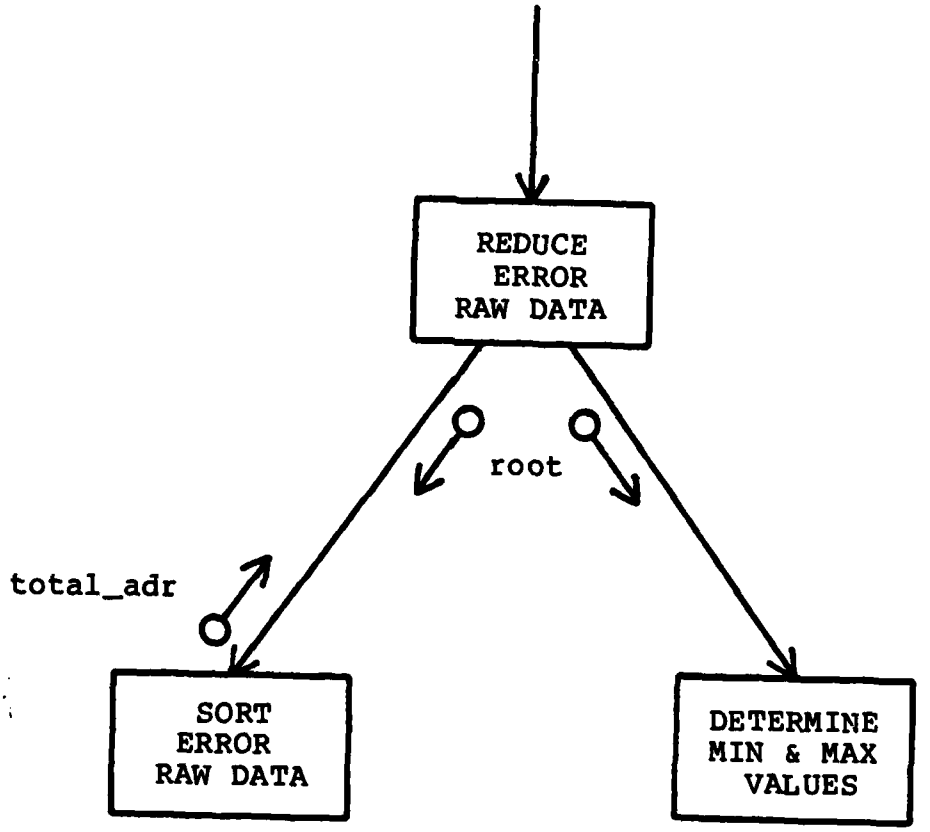
D.1.5 Do Error Analysis -



D.1.5.1 Build Error Dx file -



D.1.5.2 Reduce Error Raw Data -



APPENDIX E

DATA DICTIONARY

This appendix contains the description of all processes, constants, and variables used in the Data Extraction and Analysis System. The symbols used in describing the data composition will be defined followed by the data dictionary.

CONTENTS

E.1 SYMBOLS AND MEANINGS

1.	=	is composed of
2.	+	and
3.	|	or
4.	[ ]	choose one of (exclusive or)
5.	< >	at least one of (inclusive or)
6.	x { } Y	Iterations of the values from x to y times
7.	( )	Optional value
8.	* *	Comment
9.	<=	Less than or equal to
10.	>=	Greater than or equal to
11.	<>	Not equal to
12.	>	Greater than
13.	<	Less Than

E.2 DATA ELEMENTS

1. DATA ELEMENT NAME: bel

   DESCRIPTION: ASCII bell

   COMPOSITION: Character

   ALIASES: None

2. DATA ELEMENT NAME: clock_high

   DESCRIPTION: High 16 MSBs of elapsed time clock

   COMPOSITION: Integer

   ALIASES: None

3. DATA ELEMENT NAME: clock_low

   DESCRIPTION: Low 16 LSBs of elapsed time clock

   COMPOSITION: Integer

   ALIASES: None

4. DATA ELEMENT NAME: column

   DESCRIPTION: Column counter

   COMPOSITION: Integer

   ALIASES: None

5. DATA ELEMENT NAME: convert

   DESCRIPTION: Conversion value for lower to upper case

   COMPOSITION: Integer

   ALIASES: None

6. DATA ELEMENT NAME: entry_text

   DESCRIPTION: Entry address

   COMPOSITION: String

   ALIASES: None

7. DATA ELEMENT NAME: error_address

   DESCRIPTION: Error analysis "location" address

   COMPOSITION: Integer

   ALIASES: None

8. DATA ELEMENT NAME: error_text

   DESCRIPTION: Text form of error_address

   COMPOSITION: String

   ALIASES: None

9. DATA ELEMENT NAME: esc

   DESCRIPTION: ASCII escape

   COMPOSITION: Character

   ALIASES: None

10. DATA ELEMENT NAME: event_address

    DESCRIPTION: Address of an "event"

    COMPOSITION: Integer

    ALIASES: None

11. DATA ELEMENT NAME: event_time

DESCRIPTION: Time an "event" occured

COMPOSITION: Integrer

ALIASES: None

12. DATA ELEMENT NAME: execute_time

DESCRIPTION: Execution time of a routine

COMPOSITION: Integer

ALIASES: None

13. DATA ELEMENT NAME: Exit_text

DESCRIPTION: Exit address

COMPOSITION: String

ALIASES: None

14. DATA ELEMENT NAME: good_address

DESCRIPTION: Flag

COMPOSITION: [true, false]

ALIASES: None

15. DATA ELEMENT NAME: good_input

DESCRIPTION: Flag

COMPOSITION: [true, false]

ALIASES: None

16. DATA ELEMENT NAME: I

DESCRIPTION: Column counter

COMPOSITION: Integer

ALIASES: None

17. DATA ELEMENT NAME: input_text

DESCRIPTION: User input

COMPOSITION: String

ALIASES: None

18. DATA ELEMENT NAME: invalid_char

DESCRIPTION: Flag

COMPOSITION: [true,false]

ALIASES: None

19. DATA ELEMENT NAME: last

DESCRIPTION: Flag

COMPOSITION: [true,false]

ALIASES: None

20. DATA ELEMENT NAME: last_char

DESCRIPTION: Column count of last char

COMPOSITION: Integer

ALIASES: None

21. DATA ELEMENT NAME: marker

DESCRIPTION: -1 marker of an "event" record

COMPOSITION: Integer

ALIASES: None

22. DATA ELEMENT NAME: menu_input

DESCRIPTION: Menu selection

COMPOSITION: Character

ALIASES: None

23. DATA ELEMENT NAME: new_node

DESCRIPTION: Pointer to new node in binary tree

COMPOSITION: Pointer

ALIASES: None

24. DATA ELEMENT NAME: new_stack

DESCRIPTION: Pointer to new stack entry

COMPOSITION: Pointer

ALIASES: None

25. DATA ELEMENT NAME: next

DESCRIPTION: Pointer to next linked list record

COMPOSITION: Pointer

ALIASES: None

26. DATA ELEMENT NAME: next_time

DESCRIPTION: Pointer to time stack entry

COMPOSITION: Pointer

ALIASES: None

27. DATA ELEMENT NAME: octal_place

DESCRIPTION: octal_string column count

COMPOSITION: Integer

ALIASES: None

28. DATA ELEMENT NAME: octal_string

DESCRIPTION: Octal representation of decimal number

COMPOSITION: String

ALIASES: None

29. DATA ELEMENT NAME: quit_input

DESCRIPTION: User input for termination

COMPOSITION: Character

ALIASES: None

30. DATA ELEMENT NAME: routine

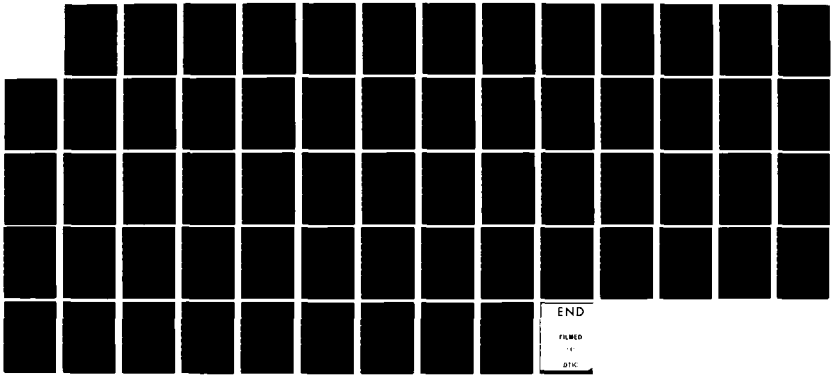
DESCRIPTION: Linked list element

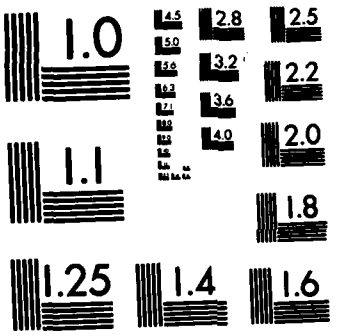
COMPOSITION: routinename + entryaddress + exitaddress + link

ALIASES: None

AD-A124 712 AUTOMATED TOOLS FOR TEST AND ANALYSIS OF RADAR WARNING RECEIVER SOFTWARE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. J R ROBERTSON 2/2

UNCLASSIFIED DEC 82 AFIT/GCS/EE/82D-28 F/G 17/9 NL





31. DATA ELEMENT NAME: stop_comand

DESCRIPTION: Flag

COMPOSITION: [true,false]

ALIASES: None

32. DATA ELEMENT NAME: tab

DESCRIPTION: ASCII tab character

COMPOSITION: Character

ALIASES: None

33. DATA ELEMENT NAME: tail

DESCRIPTION: Linked list pointer

COMPOSITION: Pointer

ALIASES: None

34. DATA ELEMENT NAME: terminate

DESCRIPTION: Flag

COMPOSITION: [true,false]

ALIASES: None

35. DATA ELEMENT NAME: transform

DESCRIPTION: Array

COMPOSITION: Array[performance..test] of string

ALIASES: None

36. DATA ELEMENT NAME: weight

DESCRIPTION: Weight of a column in an octal digit

COMPOSITION: Integer

ALIASES: None

E.3 DATA FLOWS

1. DATA FLOW NAME: address

DESCRIPTION: Memory address

COMPOSITION: Integer

ALIASES: exit_address, entry_address, cov_address, error_address

2. DATA FLOW NAME: address_text

DESCRIPTION: String representation of a memory address

COMPOSITION: string

ALIASES: entry_text, exit_text, cov_text, error_text

3. DATA FLOW NAME: adr_value

DESCRIPTION: Variable's value

COMPOSITION: Integer

ALIASES: None

4. DATA FLOW NAME: decimal

DESCRIPTION: Decimal number to be converted to octal

COMPOSITION: integer

ALIASES: cov_node_adr, error_node_adr, min_value, max_value

5. DATA FLOW NAME: error

   DESCRIPTION: Flag

   COMPOSITION: [true,false]

   ALIASES: last

6. DATA FLOW NAME: father

   DESCRIPTION: Pointer to node in binary tree

   COMPOSITION: Pointer

   ALIASES: None

7. DATA FLOW NAME: head

   DESCRIPTION: Pointer to first linked list record

   COMPOSITION: Pointer

   ALIASES: None

8. DATA FLOW NAME: help_level

   DESCRIPTION: Determine the level of help required

   COMPOSITION: [min, max]

   ALIASES: None

9. DATA FLOW NAME: input_string

   DESCRIPTION: User input

   COMPOSITION: String

   ALIASES: None

10. DATA FLOW NAME: mode

DESCRIPTION: data extraction mode

COMPOSITION: [performance, coverage, error track, rwr, quit]

ALIASES: none

11. DATA FLOW NAME: newaddress

DESCRIPTION: Pointer to new node to be inserted into binary tree

COMPOSITION: Pointer

ALIASES: None

12. DATA FLOW NAME: root

DESCRIPTION: Pointer to root node of binary tree

COMPOSITION: Pointer

ALIASES: None

13. DATA FLOW NAME: stack_top

DESCRIPTION: Pointer to top of stack

COMPOSITION: Pointer

ALIASES: entry_stack_top, time_stack_top

14. DATA FLOW NAME: stack_value

DESCRIPTION: Value of top of stack

COMPOSITION: Integer

ALIASES: entry_time, execute_time

15. DATA FLOW NAME: total_addr

DESCRIPTION: Total addresses read from rawdatafile

COMPOSITION: Integer

ALIASES: None

E.4 FILES

1. FILE NAME: dx_file

DESCRIPTION: Data extraction specification file

COMPOSITION: Text

ALIASES: None

2. FILE NAME: raw_data_file

DESCRIPTION: Data extraction raw data file

COMPOSITION: File of integer

ALIASES: None

APPENDIX F

PASCAL SOURCE LISTING

This appendix contains the source listing for the Data Analysis and Extraction System.

CONTENTS

F.1 PROGRAM DATA_ANALYSIS

```
*************************************************************

P R O G R A M   d a t a _ a n a l y s i s

*************************************************************

        VERSION      1.0 DEC 1982

AUTHOR:
        Joel R. Robertson

LANGUAGE:
        VAX-11 Pascal

PURPOSE:
        This program builds the data extraction file,
        collects data, and reduces the raw data.

REMARKS:
        This program was developed as part of the author's
        masters thesis at the Air Force Institute of
        Technology.

*************************************************************}

(*$s- ;suppress nonstandard warnings *)
(*$w- ;suppress variable length warnings *)

program data_analysis(input,output,dx_file,raw_data_file);

type
  token          = packed array [1..12] of char;
  valid_mode     = (performance,coverage,error,
                    track,rwr,quit,test);
  help_mode      = (min,max);
  memory         = integer;
  address_string = packed array [1..5] of char;
  routine_ptr    = ^routine;
  stack_ptr      = ^stack;
  cov_ptr        = ^cov_node;
  err_ptr        = ^err_node;
  dx_data        = integer;
  data_file      = file of dx_data;

  routine = record
    routine_name    : packed array [1..10] of char;
    entry_address   : memory;
    exit_address    : memory;
    entry_stack_top : stack_ptr;
    time_stack_top  : stack_ptr;
    link            : routine_ptr
  end; {routine}
```

```
    stack = record
      stack_time    : integer;
      stack_link    : stack_ptr
    end; {stack}

    cov_node = record
      cov_node_adr : memory;
      path_count    : integer;
      left,right    : cov_ptr
    end; {cov_node}

    err_node = record
      err_node_adr : memory;
      min_value     : integer;
      max_value     : integer;
      left,right    : err_ptr
    end; {err_node}

  var
    mode             : valid_mode;
    good_input       : boolean;
    transform        : array [performance..test] of token;
    input_string     : token;
    stop_command     : boolean;
    help_level       : help_mode;
    esc,tab,bel      : char;
    dx_file          : text;
    raw_data_file    : data_file;

  value
    stop_command    := false;

  {end declarations
```

F.1.1 Procedure get_input -

```
{*********************************************************

P R O C E D U R E   g e t _ i n p u t

*********************************************************

VERSION:      1.0 12-SEP-82

PURPOSE:
        Inputs a string from the console and converts all
        lower case characters to upper case.

INPUT:
        none

OUTPUT:
        input_string

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure get_input(var input_string : token);

var
     I       : integer; {column count}
     convert : integer; {upper to lower case convertsion}

begin
  convert := ord('a') - ord('A');
  read(input_string);
  for I := 1 to 12 do
    if input_string[I] in ['a'..'z'] then
      input_string[I] := chr(ord(input_string[I]) - convert);
end; {get_input
```

F.1.2 Procedure help_user -

```
{*********************************************************

P R O C E D U R E   h e l p _ u s e r

*********************************************************

VERSION:        1.0 13-SEP-82

PURPOSE:
        This procedure types a menu on the users console
        and accepts as input a menu selection. Also the
        help level is set to maximum.

INPUT:
        none

OUTPUT:
        mode
        help_level

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:
        none


*********************************************************}

procedure help_user(var mode : valid_mode;
                    var help_level : help_mode);

var
     menu_input       : char;

begin
  help_level := max;

  {set up menu}
  writeln(esc,'[2J');                    {clear screen}
  writeln;
  writeln;
  writeln(tab,'ALR-46 DATA EXTRACTION AND ANALYSIS SYSTEM');
  writeln;
  writeln(tab,tab,'**** M E N U ****');
  writeln;
  writeln;
  writeln(tab,'1.  <PERFORMANCE> ANALYSIS');
```

```
    writeln;
    writeln(tab,'2.  <COVERAGE> ANALYSIS');
    writeln;
    writeln(tab,'3.  <ERROR> CONDITION DETECTION');
    writeln;
    writeln(tab,'4.  <TRACK> FILE DISPLAY');
    writeln;
    writeln(tab,'5.  <RWR> GRAPHICS DISPLAY');
    writeln;
    writeln(tab,'6.  <QUIT> PROGRAM');
    writeln;
    write('ENTER CHOICE [1..6] => ');

    readln;
    read(menu_input);
    while not (menu_input in ['1'..'6']) do
      begin
        write(bel,'ERROR... RE-ENTER => ');
        readln;
        read(menu_input)
      end; {while}

    { determine mode }
    case menu_input of
      '1' : mode := performance;
      '2' : mode := coverage;
      '3' : mode := error;
      '4' : mode := track;
      '5' : mode := rwr;
      '6' : mode := quit
    end; {case}

  end; {help_user
```

F.1.3 Procedure get_address -

```
{*********************************************************

P R O C E D U R E   g e t _ a d d r e s s

*********************************************************

VERSION:        1.0 28-SEP-82

PURPOSE:
        This procedure returns the the address value
        plus a right justified string representation
        of the address.

INPUT:
        none

OUTPUT:
        address
        address_text

GLOBALS USED:
        none


REMARKS:

CALLED BY:
        perf_build_dx
        cov_build_dx
        error_build_dx

PROCEDURES AND FUNCTIONS CALLED:
       none

*********************************************************}

procedure get_address(var address : integer;
                      var address_text : address_string);

var
  last_char   : integer;
  column      : integer;
  weight      : integer;
  good_address,
  invalid_char        : boolean;
  input_text : packed array [1..5] of char;

begin {get address}
  good_address := false;
  repeat
    address_text := '     ';
    invalid_char := false;
    readln(input_text);
```

```
      column := 1;
      repeat
        last_char := column;
        column := column + 1
      until (input_text[column] = ' ') or (column > 5);
      for column := last_char downto 1 do
       address_text[5-(last_char-column)]:=input_text[column];
      for column := 5 downto (6-last_char) do
        if not (address_text[column] in ['0'..'7']) then
          invalid_char := true;
      if invalid_char then
        write('INVALID ENTRY => ')
      else
        good_address := true;
    until good_address;

    {convert octal string into decimal integer}
    address := 0;
    for column := 1 to 5 do
      begin
        if address_text[column] in ['1'..'7'] then
          weight := ord(address_text[column]) - ord('0')
        else
          weight := 0;
        case column of
          1 : address := address + 4096 * weight;
          2 : address := address + 0512 * weight;
          3 : address := address + 0064 * weight;
          4 : address := address + 0008 * weight;
          5 : address := address + weight
        end {case}
      end {for}
  end; {get address
```

F.1.4 Procedure octal -

```
{*********************************************************

P R O C E D U R E   o c t a l

*********************************************************

VERSION:        1.0 01-DEC-82

PURPOSE:
        This procedure types on the console the octal
        representation of the decimal number input
INPUT:
        dec_number

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:

CALLED BY:
       read_cov_tree
       read_err_tree

PROCEDURES AND FUNCTIONS CALLED:
       none

*********************************************************}

procedure octal(dec_number : integer);

VAR
  octal_string : packed array [1..6] of char;
  octal_place  : integer;


begin
  octal_string := '      ';
  octal_place := 0;

  {build octal string}
  repeat
   octal_place := octal_place + 1;
   octal_string[octal_place] := chr((dec_number mod 8)+48);
   dec_number := dec_number div 8
  until dec_number = 0;

  {type out octal string}
  for octal_place := 6 downto 1 do
    write(octal_string[octal_place])
end; {octal
```

F.1.5 Procedure perf_help -

```
{*********************************************************

P R O C E D U R E   p e r f _ h e l p

*********************************************************

VERSION:        1.0 13-SEP-82

PURPOSE:
        This procedure types instructions on the users
        console if help_level is set to 'max'.

INPUT:
        help_level

OUTPUT:
        none

GLOBALS USED:
        none


REMARKS:

CALLED BY:
        do_performance

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure perf_help(help_level : help_mode);

begin
  writeln(esc,'[2J');  {clear screen}
  writeln(tab,'*** PERFORMANCE ANALYSIS MODE ***');
  writeln;writeln;
  if help_level = max then
    begin
      writeln
      ('This mode determines the execution for program');
      writeln
      ('subroutines. An entry and an exit address must be');
      writeln('provided.')
    end;
end; {perf_help
```

F.1.6 Procedure perf_build_dx -

```
{*********************************************************

P R O C E D U R E   p e r f _ b u i l d _ d x

*********************************************************

VERSION:        1.0 13-SEP-82

PURPOSE:
        This procedure builds the performance analysis
        dx file.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        dx_file

REMARKS:

CALLED BY:
        do_performance

PROCEDURES AND FUNCTIONS CALLED:
        get_address

*********************************************************}

procedure perf_build_dx(head : routine_ptr);
var
  entry_text : address_string;
  exit_text  : address_string;
  quit_input : char;
  terminate  : boolean;
  tail,next  : routine_ptr;

begin
  terminate := false;
  tail := head;
  open(dx_file,'DXFILE',new);
  rewrite(dx_file);
  writeln;writeln;

  repeat
    writeln; writeln;
    writeln('*****************************************');
    writeln;
    writeln('ENTER NAME OF SUBROUTINE');
    write('[10 char max] => ');
    readln(tail^.routine_name);
```

```
        writeln; writeln;
        writeln('ENTER ENTRY ADDRESS');
        write('[1..77777 octal] => ');
        get_address(tail^.entry_address,entry_text);
        writeln; writeln;
        writeln('ENTER EXIT ADDRESS');
        write('[1..77777 octal] => ');
        get_address(tail^.exit_address,exit_text);
        writeln; writeln;

        {initialize stacks}
        tail^.entry_stack_top := nil;
        tail^.time_stack_top := nil;

        writeln(dx_file,entry_text);
        writeln(dx_file,exit_text);

        write('terminate input [Y/N] => ');
        readln(quit_input);
        writeln; writeln;
        if quit_input in ['Y','y'] then
          begin
            terminate := true;
            tail^.link := nil
          end
        else
          begin
            new(next);
            tail^.link := next;
            tail := next
          end;

      until terminate;
      writeln(dx_file,'    0');
      close(dx_file)
    end; {perf_build_dx
```

```
F.1.7  Procedure perf_collect_data -
*********************************************************

P R O C E D U R E   p e r f _ c o l l e c t _ d a t a

*********************************************************

VERSION:        1.0 13-SEP-82

PURPOSE:
        This procedure collects the performance analysis
        data.

INPUT:

OUTPUT:

GLOBALS USED:
        dx_file
        raw_data_file

REMARKS:
        dummy module - contractor supplied routine

CALLED BY:
        do_performance

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure perf_collect_data;

begin
  writeln;
  writeln('*******************************************');
  writeln('*** procedure perf_collect_data called ***');
  writeln('*******************************************');
  writeln
end; {perf_collect_data
```

F.1.8 Procedure push_stack -

```
{*********************************************************

P R O C E D U R E   p u s h _ s t a c k

*********************************************************

VERSION:        1.0 12-OCT-82

PURPOSE: pushes a value on the stack pointed to by
         stack_top.

INPUT:
        stack_top       -pointer to top of stack
        stack_value     -value to be pushed on stack

OUTPUT:
        stack_top       -new top of stack

GLOBALS USED:
        none


REMARKS:

CALLED BY:
        perf_reduce_data

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure push_stack(var stack_top : stack_ptr;
                     stack_value : integer);

var
  new_stack : stack_ptr;

begin
  new(new_stack);
  new_stack^.stack_time := stack_value;
  if stack_top = nil then
    new_stack^.stack_link := nil
  else
    new_stack^.stack_link := stack_top;
  stack_top := new_stack
end; {push_stack
```

F.1.9 Procedure popstack -

```
{*********************************************************

P R O C E D U R E   p o p _ s t a c k

*********************************************************

VERSION:        1.0 11-OCT-82

PURPOSE:
        pops value from top of stack


INPUT:
        stack_top       -pointer to top of stack

OUTPUT:
        stack_value     -value popped from stack
        stack_top       -new top of stack

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        perf_reduce_data

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure pop_stack(var stack_top   : stack_ptr;
                    var stack_value : integer;
                    var error       : boolean);

begin
  if stack_top = nil then
    error := true
  else
    begin
      stack_value := stack_top^.stack_time;
      stack_top := stack_top^.stack_link
    end;
end; {pop_stack
```

F.1.10 Procedure perf_reduce_data -

```
{*********************************************************

P R O C E D U R E   p e r f _ r e d u c e _ d a t a

*********************************************************

VERSION:        1.0 21-NOV-82

PURPOSE:
        This procedure reduces the performance analysis
        data.

INPUT:
        head                -front of linked linked list
        raw_data_file       -file produced by dx system

OUTPUT:
        none

GLOBALS USED:
        raw_data_file

REMARKS:

CALLED BY:
        do_performance

PROCEDURES AND FUNCTIONS CALLED:
        push_stack
        pop_stack

*********************************************************}

procedure perf_reduce_data(head : routine_ptr);

var
  tail          : routine_ptr;
  count         : integer;

  {'event' record}
  marker        : integer;
  event_address : integer;
  clock_high    : integer;
  clock_low     : integer;

  event_time,
  entry_time,
  execute_time  : integer;
  next          : routine_ptr;
  error,last    : boolean;
  next_time     : stack_ptr;

begin
```

```
open(raw_data_file,'DXDATA',old);
reset(raw_data_file);
writeln;
while (not eof(raw_data_file)) and (not error) do
  begin
    read(raw_data_file,marker,event_address,
         clock_high,clock_low);
    if marker = -1 then
     begin
      next := head;
      event_time := clock_low + 65536 * clock_high;
      repeat
       if next^.entry_address = event_address then
         push_stack(next^.entry_stack_top,event_time);
       if next^.exit_address = event_address then
        begin
         pop_stack(next^.entry_stack_top,entry_time,error);
         execute_time := event_time - entry_time;
         push_stack(next^.time_stack_top,execute_time)
         end;
       next := next^.link;
      until next = nil
     end
    else
      begin
        error := true;
        writeln('RAW DATA FILE FORMAT ERROR')
      end {else}
  end; {while}
close(raw_data_file);

{*********************************************}
{* Format and print out the execution times *}
{*********************************************}

if error = false then
  begin
    writeln(esc,'[2J');  {clear screen}
    next := head;
    repeat
      count := 0;       {reset count}
      writeln;
      write(tab,'EXECUTION TIMES FOR SUBROUTINE ');
      writeln(next^.routine_name);
      writeln
      (tab,'---------------------------------------');
      next_time := next^.time_stack_top;
      repeat
        last  := false;  {not end yet}
        pop_stack(next_time,execute_time,last);
        if not last then {stack empty ?}
          begin
            write(execute_time);
            count := count +1
```

```
            end;
          if count > 4 then
            begin
              count := 0;  {reset count}
              writeln      {new line}
            end
        until last;
        next := next^.link;
        writeln;
        writeln
        (tab,'---------------------------------------')
      until next = nil;
      writeln;
    end
end; {perf_reduce_data
```

F.1.11 Procedure do_performance -

```
*************************************************************

P R O C E D U R E   d o _ p e r f o r m a n c e

*************************************************************

VERSION:        1.0 13-SEP-82

PURPOSE:
        This procedure does the subroutine performance
        analysis

INPUT:
        help_level

OUTPUT:
        head

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:
        perf_help
        perf_build_dx
        perf_collect_data
        perf_reduce_data

*************************************************************}

procedure do_performance(help_level : help_mode);

var
  head : ^routine;

begin {do_performance}
  perf_help(help_level);

  {build the performance analysis dx file}
  new(head);
  perf_build_dx(head);

  {call the contractor supplied data extraction routine}
  perf_collect_data;

  {reduce the extracted data}
  perf_reduce_data(head);

end; {do_performance
```

F.1.12 Procedure cov_help -

```
{*********************************************************

P R O C E D U R E   c o v _ h e l p

*********************************************************

VERSION:        1.0 22-NOV-82

PURPOSE:
        This procedure types instructions on the users
        console if help_level is set to 'max'.

INPUT:
        help_level

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        do_coverage

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure cov_help(help_level : help_mode);

begin
  writeln(esc,'[2J');  {clear screen}
  writeln(tab,'*** COVERAGE ANALYSIS MODE ***');
  if help_level = max then
    begin
      writeln
      ('This mode counts the occurances of path markers');
      writeln('to determine the testing coverage.');
    end;
  writeln;
  writeln('Enter path marker address <0 to end>')
end; {cov_help
```

F.1.13 Procedure cov_build_dx -

```
{*********************************************************

P R O C E D U R E   c o v _ b u i l d _ d x

*********************************************************

VERSION:        1.0 22-NOV-82

PURPOSE:
        This procedure builds the coverage analysis
        dx file. dx_file will be an 'event' type file.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        dx_file

REMARKS:

CALLED BY:
        do_coverage

PROCEDURES AND FUNCTIONS CALLED:
        get_address

*********************************************************}

procedure cov_build_dx;

var
  cov_address    : memory;
  cov_text       : address_string;

begin
  open(dx_file,'DXFILE',new);
  rewrite(dx_file);
  repeat
    write('PATH MARKER [1..77777 octal] => ');
    get_address(cov_address,cov_text);
    writeln(dx_file,cov_text)
  until cov_address = 0;
  close(dx_file)
end; {cov_build_dx
```

F.1.14 Procedure cov_collect_data -

```
{*********************************************************

P R O C E D U R E   c o v _ c o l l e c t _ d a t a

*********************************************************

VERSION:        1.0 27-SEP-82

PURPOSE:
        This procedure collects the coverage analysis
        data.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        dx_file
        raw_data_file

REMARKS:
        dummy module - contractor supplied routine

CALLED BY:
        do_coverage

PROCEDURES AND FUNCTIONS CALLED:

*********************************************************}

procedure cov_collect_data;

begin
  writeln;
  writeln('*****************************************');
  writeln('*** procedure cov_collect_data called ***');
  writeln('*****************************************');
  writeln
end; {cov_collect_data
```

F.1.15 Procedure cov_tree_search -

```
{*********************************************************

P R O C E D U R E   c o v _ t r e e _ s e a r c h

*********************************************************

VERSION:        1.0 21-NOV-82

PURPOSE:
        This procedure searches the binary tree for an
        address. If the address is found the path count
        is incremented, otherwise the address will be
        inserted into the binary tree.

INPUT:
        father
        new_address

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:
        this is a recursive procedure

CALLED BY:
        build_cov_tree

PROCEDURES AND FUNCTIONS CALLED:
        cov_tree_search

*********************************************************}

procedure cov_tree_search(var father : cov_ptr;
                              new_address : memory);

var
  new_node : cov_ptr;

begin
  if new_address = father^.cov_node_adr then
    {we found it update count}
    father^.path_count := father^.path_count + 1
  else
    if new_address < father^.cov_node_adr then
      begin
        if father^.left = nil then
          begin
            new(new_node);
            father^.left := new_node;
            with new_node^ do
```

```
                    begin
                      cov_node_adr := new_address;
                      path_count := 1;
                      left := nil;
                      right := nil
                    end {with}
                end
              else
                begin
                  father := father^.left;
                  cov_tree_search(father,new_address)
                end
            end
          else
            if father^.right = nil then
              begin
                new(new_node);
                father^.right := new_node;
                with new_node^ do
                  begin
                    cov_node_adr := new_address;
                    path_count := 1;
                    left := nil;
                    right := nil
                  end {with}
              end
            else
              begin
                father := father^.right;
                cov_tree_search(father,new_address)
              end {else}
        end; {cov_tree_search
```

F.1.16 Procedure build_cov_tree -

```
{*********************************************************

P R O C E D U R E   b u i l d _ c o v _ t r e e

*********************************************************

VERSION:        1.0 21-NOV-82

PURPOSE:
        This procedure builds the coverage analysis
        binary tree.

INPUT:
        root
        total_adr

OUTPUT:
        none

GLOBALS USED:
        raw_data_file

REMARKS:

CALLED BY:
        cov_reduce_data

PROCEDURES AND FUNCTIONS CALLED:
        cov_tree_search

*********************************************************}

procedure build_cov_tree(var root : cov_ptr;
                         var total_adr : integer);

var
  new_address : memory;     {address to be inserted in tree}
  new_node,                 {pointers to nodes}
  father      : cov_ptr;

begin {build_cov_tree}
  open(raw_data_file,'DXDATA',old);
  {insert the first word in the root node}
  reset(raw_data_file);  {discard marker}
  if not eof(raw_data_file) then
    begin
      get(raw_data_file); {get address}
      new_address := raw_data_file^;
      total_adr := 1;
      with root^ do
        begin
          cov_node_adr := new_address;
          path_count := 1;
```

```
            left := nil;
            right := nil
          end; {with}
        get(raw_data_file); {discard clock high}
        get(raw_data_file); {discard clock low}
      end
    else
      root := nil;

    {attach each address to its father}
    get(raw_data_file); {discard marker}
    while not eof(raw_data_file) do
      begin
        get(raw_data_file); {get address}
        new_address := raw_data_file^;
        total_adr := total_adr + 1;
        father := root; {first father is root}
        {insert each path address into tree}
        cov_tree_search(father,new_address);
        get(raw_data_file); {discard clock high}
        get(raw_data_file);  {discard clock low}
        get(raw_data_file); {discard marker}
      end; {while}
    close(raw_data_file)
  end; {build_cov tree
```

F.1.17 Procedure read_cov_tree -

```
{*************************************************************

P R O C E D U R E   r e a d _ c o v _ t r e e

*************************************************************

VERSION:        1.0 01-DEC-82

PURPOSE:
        This procedure does an in order traversal of the
        binary tree.

INPUT:
        root

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:
        recursive procedure

CALLED BY:
        cov_reduce_data

PROCEDURES AND FUNCTIONS CALLED:
        read_cov_tree
        octal

*************************************************************}

procedure read_cov_tree(root : cov_ptr);

begin
  if root <> nil then
    begin
      {stoping state not reached - perform recursion}
      read_cov_tree(root^.left);
      write(tab);
      octal(root^.cov_node_adr);
      writeln('   => ',root^.path_count);
      read_cov_tree(root^.right);
    end {if}
end; {read_cov_tree
```

F.1.18 Procedure cov_reduce_data -

```
{*********************************************************

P R O C E D U R E   c o v _ r e d u c e _ d a t a

*********************************************************

VERSION:        1.0 21-NOV-82

PURPOSE:
        This procedure reduces the coverage analysis
        data. First it builds a binary tree and counts
        the number of traversals. second, it does an
        inorder traversal of the tree and prints out
        the number of traversals.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        raw_data_file

REMARKS:

CALLED BY:
        do_coverage

PROCEDURES AND FUNCTIONS CALLED:
        build_cov_tree
        read_cov_tree

*********************************************************}

procedure cov_reduce_data;

var
  total_addr : integer;   {number of addresses read}
  root       : cov_ptr;   {root of tree}

begin
  writeln;
  new(root); {create root node}
  build_cov_tree(root,total_addr);
  if total_addr <> 0 then
    begin
      writeln(esc,'[2J');       {clear screen}
      writeln;
      writeln('       ADDRESS       NUMBER OF TRAVERSALS');
      writeln('       -------       --------------------');
      read_cov_tree(root);
      writeln;
```

```
        write('TOTAL NUMBER OF PATHS COVERED = ');
        writeln(total_addr: 2)
      end
    else
      writeln('FILE EMPTY')
  end; {cov_reduce_data
```

F.1.19 Procedure do_coverage -

```
{*********************************************************

P R O C E D U R E   d o _ c o v e r a g e

*********************************************************

VERSION:        1.0 27-SEP-82

PURPOSE:
        This procedure does the coverage analysis.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:
        cov_help
        cov_build_dx
        cov_collect_data
        cov_reduce_data

*********************************************************}

procedure do_coverage(help_level : help_mode);

begin
  cov_help(help_level);

  {build the coverage analysis dx file}
  cov_build_dx;

  {call the contractor supplied data extraction routine}
  cov_collect_data;

  {reduce the extracted data}
  cov_reduce_data;

end; {do_coverage
```

F.1.20 Procedure error_help -

```
{*********************************************************

P R O C E D U R E   e r r o r _ h e l p

*********************************************************

VERSION:        1.0 22-NOV-82

PURPOSE:
        This procedure types instructions on the users
        console if help_level is set to 'max'.

INPUT:
        help_level

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:

CALLED BY:
        do_error

PROCEDURES AND FUNCTIONS CALLED:
        none

*********************************************************}

procedure error_help(help_level : help_mode);

begin
  writeln(esc,'[2J');  {clear screen}
  writeln(tab,'*** ERROR ANALYSIS MODE ***');
  if help_level = max then
    begin
      writeln
      ('This mode determines the minimun and maximum');
      writeln
      ('values obtained by a program variable during');
      writeln('a particular test run.')
    end;
  writeln;
  writeln
  ('Enter variable addresses to be extracted in octal');
  writeln('end entry with 0.')
end; {error_help
```

F.1.21 Procedure error_build_dx -

```
{*********************************************************

P R O C E D U R E   e r r o r _ b u i l d _ d x

*********************************************************

VERSION:        1.0 22-NOV-82

PURPOSE:
        This procedure builds the error analysis dx file.
        The dx_file is a 'location' format file.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        dx_file

REMARKS:

CALLED BY:
        do_error

PROCEDURES AND FUNCTIONS CALLED:
        get_address

*********************************************************}

procedure error_build_dx;

var
  error_address     : memory;
  error_text        : address_string;

begin
  open(dx_file,'DXFILE',new);
  rewrite(dx_file);
  writeln(dx_file,'    0');
  repeat
    write('ADDRESS [1..77777 octal] => ');
    get_address(error_address,error_text);
      if error_address <> 0 then
        begin
          writeln(dx_file,error_text,',',error_text)
        end
  until error_address = 0;
  close(dx_file)
end; {error_build_dx
```

F.1.22 Procedure error_collect_data -

```
{*********************************************************

P R O C E D U R E   e r r o r _ c o l l e c t _ d a t a

*********************************************************

VERSION:        1.0 27-SEP-82

PURPOSE:
        This procedure collects the error analysis data.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        dx_file
        raw_data_file

REMARKS:
        dummy module - this routine is contractor supplied

CALLED BY:
        do_error

PROCEDURES AND FUNCTIONS CALLED:

*********************************************************}

procedure error_collect_data;

begin
  writeln;
  writeln('*********************************************');
  writeln('*** procedure error_collect_data called ***');
  writeln('*********************************************');
  writeln
end; {error_collect_data
```

F.1.23 Procedure err_tree_search -

```
{*********************************************************

P R O C E D U R E   e r r _ t r e e _ s e a r c h

*********************************************************

VERSION:        1.0 04-NOV-82

PURPOSE:
        This procedure searches the binary tree for an
        address. If the address is found new min and
        max values will be calculated, otherwise the
        address will be inserted into the binary tree.

INPUT:
        father
        new_address
        adr_value

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:
        this is a recursive procedure

CALLED BY:
        build_err_tree

PROCEDURES AND FUNCTIONS CALLED:
        err_tree_search

*********************************************************}

procedure err_tree_search(var father : err_ptr;
                              new_address : memory;
                              adr_value : integer);

var
  new_node : err_ptr;

begin
  if new_address = father^.err_node_adr then
    begin
      {we found it, calculate new min and max }
      if adr_value > father^.max_value then
        father^.max_value := adr_value;
      if adr_value < father^.min_value then
        father^.min_value := adr_value
    end
  else
```

```
        if new_address < father^.err_node_adr then
          begin
            if father^.left = nil then
              begin
                new(new_node);
                father^.left := new_node;
                with new_node^ do
                  begin
                    err_node_adr := new_address;
                    min_value := adr_value;
                    max_value := adr_value;
                    left := nil;
                    right := nil
                  end {with}
              end
            else
              begin
                father := father^.left;
                err_tree_search(father,new_address,adr_value)
              end
          end
        else
          if father^.right = nil then
            begin
              new(new_node);
              father^.right := new_node;
              with new_node^ do
                begin
                  err_node_adr := new_address;
                  min_value := adr_value;
                  max_value := adr_value;
                  left := nil;
                  right := nil
                end {with}
            end
          else
            begin
              father := father^.right;
              err_tree_search(father,new_address,adr_value)
            end {else}
      end; {err_tree_search
```

F.1.24 Procedure build_err_tree -

```
{*********************************************************

P R O C E D U R E   b u i l d _ e r r _ t r e e

*********************************************************

VERSION:        1.0 04-NOV-82

PURPOSE:
        This procedure builds the error analysis
        binary tree.

INPUT:
        root
        total_adr

OUTPUT:
        none

GLOBALS USED:
        raw_data_file

REMARKS:

CALLED BY:
        err_reduce_data

PROCEDURES AND FUNCTIONS CALLED:
        err_tree_search

*********************************************************}

procedure build_err_tree(var root : err_ptr;
                         var total_adr : integer);

var
  new_address : memory;  {address to be inserted in tree}
  adr_value   : dx_data; {value written to a location}

  new_node,               {pointers to nodes}
  father      : err_ptr;

begin {build_err_tree}
  open(raw_data_file,'DXDATA',old);
  {insert the first word in the root node}
  reset(raw_data_file);
  if not eof(raw_data_file) then
    begin
      read(raw_data_file,new_address,adr_value);
      total_adr := 1;
      with root^ do
        begin
          err_node_adr := new_address;
```

```
            min_value := adr_value;
            max_value := adr_value;
            left := nil;
            right := nil
          end; {with}
      end
    else
      root := nil;

    {attach each address to its father}
    while not eof(raw_data_file) do
      begin
        read(raw_data_file,new_address,adr_value);
        total_adr := total_adr + 1;
        father := root; {first father is root}
        {insert each path address into tree}
        err_tree_search(father,new_address,adr_value);
      end; {while}
    close(raw_data_file)
  end; {build_err tree
```

F.1.25 Procedure read_err_tree -

```
{*************************************************************

P R O C E D U R E   r e a d _ e r r _ t r e e

*************************************************************

VERSION:        1.0 01-DEC-82

PURPOSE:
        This procedure does an in order traversal of the
        binary tree.

INPUT:
        root

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:
        recursive procedure

CALLED BY:
        err_reduce_data

PROCEDURES AND FUNCTIONS CALLED:
        read_err_tree
        octal

*************************************************************}

procedure read_err_tree(root : err_ptr);

begin
  if root <> nil then
    begin
      {stoping state not reached - perform recursion}
      read_err_tree(root^.left);
      write(tab);
      octal(root^.err_node_adr);
      write('   =>   ');
      octal(root^.min_value);
      write('       ');
      octal(root^.max_value);
      writeln;
      read_err_tree(root^.right);
    end {if}
end; {read_err_tree
```

F.1.26 Procedure error_reduce_data -

```
{*************************************************************

P R O C E D U R E   e r r o r _ r e d u c e _ d a t a

*************************************************************

VERSION:        1.0 27-SEP-82

PURPOSE:
        This procedure reduces the error analysis data.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        raw_data_file

REMARKS:

CALLED BY:
        do_error

PROCEDURES AND FUNCTIONS CALLED:

*************************************************************}

procedure error_reduce_data;

var
  total_addr : integer;   {number of addresses read}
  root       : err_ptr;   {root of tree}

begin
  writeln;
  new(root); {create root node}
  build_err_tree(root,total_addr);
  if total_addr <> 0 then
    begin
      writeln(esc,'[2J');       {clear screen}
      writeln;
      writeln
      (tab,'ADDRESS         MIN VALUE  MAX VALUE');
      writeln
      (tab,'-------         ---------  ---------');
      read_err_tree(root);
      writeln;
```

```
        write('TOTAL NUMBER OF ADDRESSES READ =');
        writeln(total_addr :2)
      end
    else
      writeln('FILE EMPTY')
  end; {error_reduce_data
```

F.1.27 Procedure do_error -

```
{*************************************************************

P R O C E D U R E   d o _ e r r o r

*************************************************************

VERSION:        1.0 27-SEP-82

PURPOSE:
        This procedure does the error analysis.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        error_dx_file

REMARKS:
        dummy module

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:
        error_help
        error_build_dx
        error_collect_data
        error_reduce_data

*************************************************************}

procedure do_error(var help_level : help_mode);

begin
  error_help(help_level);

  {build the error analysis dx file}
  error_build_dx;

  {call the contractor supplied data extraction routine}
  error_collect_data;

  {reduce the extracted data}
  error_reduce_data
end; {do_error
```

F.1.28 Procedure do_trackfile -

```
{*********************************************************

P R O C E D U R E   d o _ t r a c k f i l e

*********************************************************

VERSION:        1.0 12-SEP-82

PURPOSE:
        This procedure displays the contents of the emitter
        track file.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:
        Dummy module -
        Code for this module from previous thesis

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:

*********************************************************}

procedure do_trackfile;

begin
  writeln;
  writeln('*******************************************');
  writeln('*** not inplemented in thesis version ***');
  writeln('*******************************************');
  writeln
end; {do_trackfile
```

F.1.29 Procedure do_rwr -

```
{*********************************************************

P R O C E D U R E   d o _ r w r

*********************************************************

VERSION:        1.0 12-SEP-82

PURPOSE:
        This procedure displays a simulation of the RWR
        CRT on the color graphics terminal.

INPUT:
        none

OUTPUT:
        none

GLOBALS USED:
        none

REMARKS:
        Dummy module -
        Code for this module from previous thesis

CALLED BY:
        data_analysis

PROCEDURES AND FUNCTIONS CALLED:

*********************************************************}

procedure do_rwr;

begin
  writeln;
  writeln('*************************************');
  writeln('*** not inplemented in thesis version ***');
  writeln('*************************************');
  writeln
end; {do_rwr
```

F.1.30 Main Executive -

```
{*************************************************************

THIS IS THE MAIN EXECUTIVE ROUTINE

*************************************************************}

begin {data_analysis}
  esc := chr(27);                {ASCII ESCAPE}
  bel := chr(7);                 {BEEP}
  tab := chr(9);                 {ASCII TAB}

  writeln(esc,'[2J');                    {clear_screen}
  writeln
  (tab,'*** ALR-46 DATA EXTRACTION AND ANALYSIS SYSTEM ***');
  writeln;
  writeln('Type <help> for menu');
  writeln;

  {initialize the transformation array}
  transform[performance] := 'PERFORMANCE ';
  transform[coverage] := 'COVERAGE    ';
  transform[error] := 'ERROR       ';
  transform[track] := 'TRACK       ';
  transform[rwr] := 'RWR         ';
  transform[quit] := 'QUIT        ';

  repeat
    help_level := min;
    good_input := false;
    {loop until valid command}
    while not good_input do
      begin
        write ('ENTER COMMAND => ');
        get_input(transform[test]);

        { select mode from menu or command }
        if transform[test] = 'HELP        ' then
          begin
            help_user(mode,help_level);
            good_input := true
          end
        else
          begin
            mode := performance;
            {search the transformation array for a match}
            while transform[mode] <> transform[test] do
              mode := succ(mode);
            {check for invalid input}
            if mode = test then
              writeln (bel,'INVALID COMMAND... try again')
            else
              good_input := true
          end {else}
```

```
        end; {while}

      {command has been converted into an enumeration type}
      case mode of
        performance : do_performance(help_level);
        coverage       : do_coverage(help_level);
        error   : do_error(help_level);
        track   : do_trackfile;
        rwr     : do_rwr;
        quit    : stop_command := true
      end; {case}
    until stop_command
  end. {analysis

  }
```

APPENDIX G

TEST DOCUMENTATION

This appendix contains the test plans used in validating the proper operation of the Data Extraction and Analysis System. Test cases were derived from the software requirements using equivalence partitioning and boundary value analysis.

CONTENTS

G.1 TEST PLANS

TEST CASE	INPUT	EXPECTED OUTPUT
01	Command PERFORMANCE	Enter performance analysis mode
02	Command COVERAGE	Enter coverage analysis mode
03	Command ERROR	Enter error analysis mode
04	Command TRACK	Enter trackfile analysis mode
05	Command RWR	Enter RWR analysis mode
06	Command HELP	Print menu
07	Command QUIT	Exit to operating system
08	Command INVALID	Print error message
09	Menu selection #1	Enter performance analysis mode
10	Menu selection #2	Enter coverage analysis mode
11	Menu selection #3	Enter error analysis mode
12	Menu selection #4	Enter trackfile analysis mode

TEST CASE	INPUT	EXPECTED OUTPUT
13	Menu selection #5	Enter RWR analysis mode
14	Menu selection #6	Exit to operating system
15	Menu selection #7	Print error message
16	Performance mode: Entry address = 1..32767 Exit address = 1..32767	'DXFILE.DAT' contains right justified 5 char entry and exit addresses
17	Performance mode: Entry address > 32767	Print error message
18	Performance mode: Exit address > 32767	Print error message
19	Performance mode: terminate input	Execute data collection Print module name and execution times
20	Performance mode: Empty data file	Print module name with no execution times
21	Coverage mode: Path address 1..32767	'DXFILE.DAT' contains right justified 5 char path address
22	Coverage mode: Path address > 32767	Print error message
23	Coverage mode: Path address = 0	Execute data collection Print path address and # times executed
24	Coverage mode: Empty data file	No data printed

TEST CASE	INPUT	EXPECTED OUTPUT
25	Error mode: Variable address = 1..32767	'DXFILE.DAT' contains right justified 5 digit text of addresses
26	Error mode: Variable address > 32767	Print error message
27	Error mode: variable address = 0	Execute data collection Print variable addresses with min and max values
28	Error mode: Empty data file	No data printed

G.2 TEST DATA AND RESULTS

G.2.1 Performance Analysis - This section contains the test data used to validate the "performance analysis" mode of the Data Extraction and Analysis System.

G.2.1.1 Input Data -

Subroutine Name - TEST1
Entry Address - 1
Exit Address - 77777

Subroutine Name - TEST2
Entry Address - 400
Exit Address - 1000

Subroutine Name - TEST3
Entry Address - 2000
Exit Address - 4000

G.2.1.2 Data Reduction Simulation Data - The following data is data used to simulate the operation of the data extraction subsystem. This data is in the internal decimal representation used by the DEAS. The format of the data is described in Appendix B.

```
   -1
    1
    0
    0
   -1
32767
    0
```

```
10
-1
256
0
20
-1
512
0
30
-1
1024
0
40
-1
2048
0
50
-1
1
0
60
-1
256
0
70
-1
1024
0
80
-1
2048
0
90
-1
512
0
100
-1
32767
0
110
-1
1024
0
120
-1
2048
0
130
-1
256
0
140
-1
```

```
  512
    0
  150
   -1
    1
    0
  160
   -1
32767
    0
  170
```

G.2.1.3 Dx file Produced - The following is the data extraction specification file to be used by the data extraction subsystem. The file is a text file and is verified by visual comparison with the input data.

```
    1
77777
  400
 1000
 2000
 4000
    0
```

G.2.1.4 Console Output - The following is the display generated by the DEAS when supplied with the previously described data.

```
EXECUTION TIMES FOR SUBROUTINE TEST1
----------------------------------------
10          50          10
----------------------------------------

EXECUTION TIMES FOR SUBROUTINE TEST2
----------------------------------------
10          30          10
----------------------------------------

EXECUTION TIMES FOR SUBROUTINE TEST3
----------------------------------------
10          10          10
----------------------------------------
```

G.2.2 Coverage Analysis - This section contains the test data used to validate the "coverage analysis" mode of the Data Extraction and Analysis System.

G.2.2.1 Input Data -

PATH MARKER #1	1
PATH MARKER #2	400
PATH MARKER #3	1000
PATH MARKER #4	2000
PATH MARKER #5	4000
PATH MARKER #6	77777

G.2.2.2 Data Reduction Simulation Data - The same simulation file used for the "performance analysis" mode was used for this mode.

G.2.2.3 Dxfile Produced - The following is the data extraction specification file to be used by the data extraction subsystem. The file is a text file and is verified by visual comparison with the input data.

```
    1
  400
 1000
 2000
 4000
77777
    0
```

G.2.2.4 Console Output - The following is the display generated by the DEAS when supplied with the previously described data.

ADDRESS		NUMBER OF TRAVERSALS
1	=>	3
400	=>	3
1000	=>	3
2000	=>	3
4000	=>	3
77777	=>	3

TOTAL NUMBER OF PATHS COVERED = 18

G.2.3 Error Analysis - This section contains the test data used to validate the "error analysis" mode of the Data Extraction and Analysis System.

G.2.3.1 Input Data -

VARIABLE ADDRESS #1 66

VARIABLE ADDRESS #2 504

VARIABLE ADDRESS #3 1733

VARIABLE ADDRESS #4 3720

VARIABLE ADDRESS #5 1

VARIABLE ADDRESS #6 77777

VARIABLE ADDRESS #7 13055

G.2.3.2 Data Reduction Simulation Data - The following data is data used to simulate the operation of the data extraction subsystem. This data is in the internal decimal representation used by the DEAS. The format of the data is described in Appendix B.

```
    1
    0
    1
  456
    1
32767
32767
32767
 5677
 3455
  987
 6546
   54
  786
  324
 4000
 2000
  400
  987
   20
 5677
    0
   54
32767
```

G.2.3.3 Dxfile Produced - The following is the data extraction specification file to be used by the data extraction subsystem. The file is a text file and is verified by visual comparison with the input data.

```
    0
   66,   66
  504,  504
 1733, 1733
 3720, 3720
    1,    1
77777,77777
```

13055,13055

G.2.3.4 Console Output - The following is the display generated by the DEAS when supplied with the previously described data.

ADDRESS		MIN VALUE	MAX VALUE
1	=>	0	77777
66	=>	1422	77777
504	=>	7640	7640
1733	=>	24	14622
3720	=>	620	620
13055	=>	0	6577
77777	=>	77777	77777

TOTAL NUMBER OF ADDRESSES READ = 12

VITA

Mr. Joel R. Robertson was born on January 24, 1952, in Huntsville, Alabama. In 1969, he graduated from John Marshall High School in Glendale, West Virginia. He attended West Liberty State College from which he received an Associate of Science degree in Electronics Technology in 1971. He then attended West Virginia University from which he received a Bachelor of Science degree in Electrical Engineering in 1975. Following graduation, he accepted a position with the Federal Aviation Administration in Atlanta, Georgia. He was employed with the FAA as an electrical engineer installing radar systems until 1979 when he accepted employment with the Electronic Warfare Division of the United States Air Force at Robins AFB, Warner Robins, Georgia, as a civilian electronic engineer working in the Radar Warning Receiver Section. He entered the Air Force Institute of Technology at Wright Patterson AFB, Ohio in June 1981.

Permanent Address: 32 Seminole Road
Brunswick, Georgia 31520

END

FILMED

3-83

DTIC