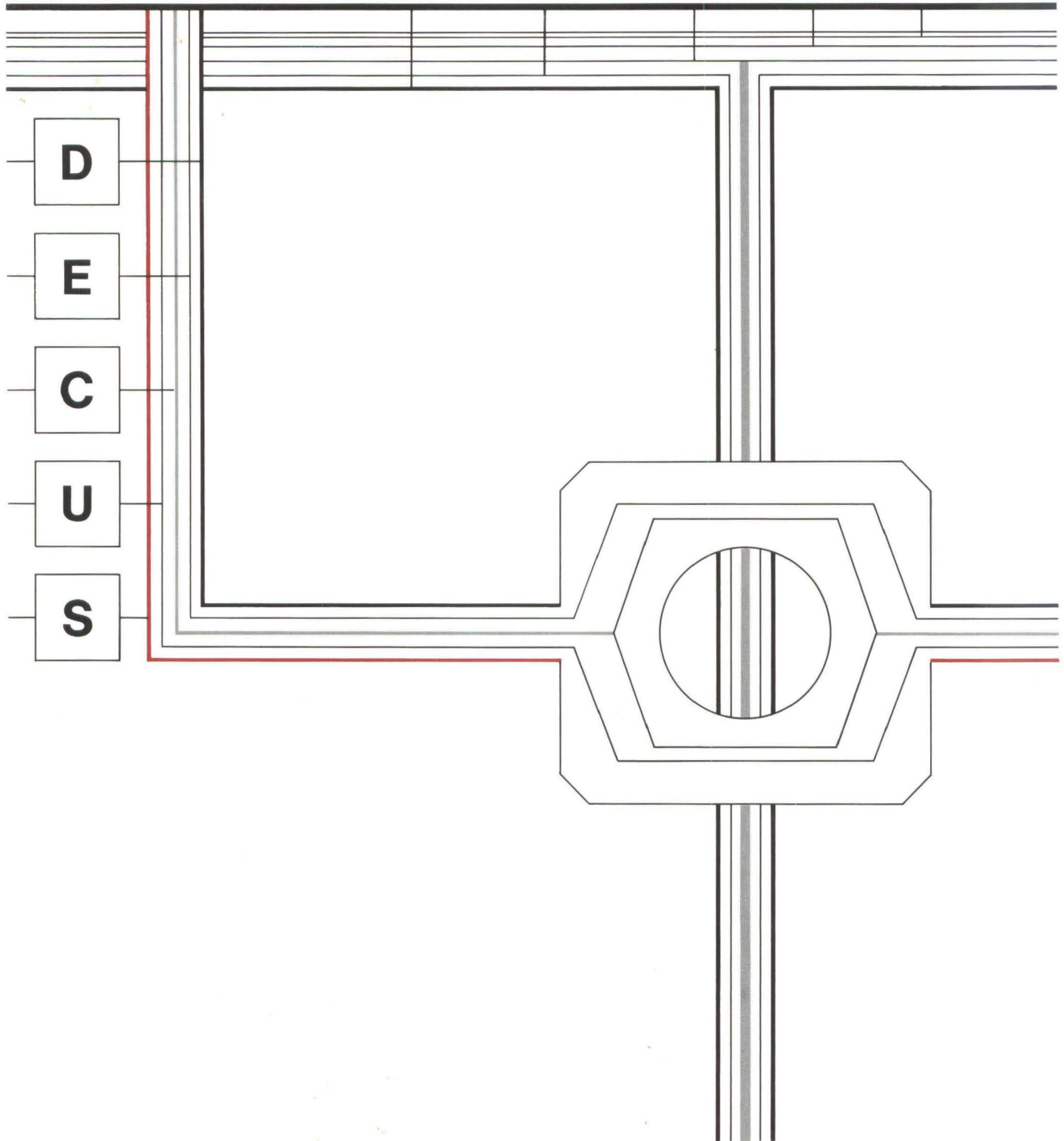


USA

1985 FALL

PROCEEDINGS OF THE DIGITAL EQUIPMENT COMPUTER USERS SOCIETY





**PROCEEDINGS
OF THE
DIGITAL EQUIPMENT
COMPUTER USERS
SOCIETY**

**Presentation and Reports
USA Fall 1985**

**Anaheim, California
December 9-13, 1985**

Printed in the U.S.A.

"The Following are trademarks of Digital Equipment Corporation"

ALL-IN-1	Digital logo	Rainbow
DEC	EduSystem	RSTS
DECnet	Eve	RSX
DECmate	IAS	RT
DECsystem-10	MASSBUS	UNIBUS
DECSYSTEM-20	PDP	VAX
DECUS	PDT	VMS
DECwriter	P/OS	VT
DIBOL	Professional	Work Processor

Copyright ©DECUS and Digital Equipment Corporation 1986
All Rights Reserved

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation or DECUS. Digital Equipment Corporation and DECUS assume no responsibility for any errors that may appear in this document.

POLICY NOTICE TO ALL ATTENDEES OR CONTRIBUTORS "DECUS PRESENTATIONS, PUBLICATIONS, PROGRAMS, OR ANY OTHER PRODUCT WILL NOT CONTAIN TECHNICAL DATA/INFORMATION THAT IS PROPRIETARY, CLASSIFIED UNDER U.S. GOVERNED BY THE U.S. DEPARTMENT OF STATE'S INTERNATIONAL TRAFFIC IN ARMS REGULATIONS (ITAR)."

DECUS and Digital Equipment Corporation make no representation that in the interconnection of products in the manner described herein will not infringe on any existing or future patent rights nor do the descriptions contained herein imply the granting of licenses to utilize any software so described or to make, use or sell equipment constructed in accordance with these descriptions.

Ada is a trademark of the U.S. Government, XEROX is a trademark of Xerox Corporation, IBM, PROFS are trademarks of International Business Machines Corporation, UNIX is a trademark of AT&T Bell Laboratories, CP/M, PL/I are trademarks of Digital Research, Inc., MS-DOS is a trademark of Microsoft Corporation, TSX-PLUS is a trademark of S&H Computer Systems Inc, R:BASE.4000 is a trademark of Microrim, Intel 8088 is a trademark of Intel Corporation, LOTUS 1-2-3 is a trademark of Lotus Development Corporation, MULTIPLAN is a trademark of Microsoft Corporation, Mylar is a trademark of E. I. DuPont deNemours & Co., PLOTLN is a trademark of Image Research and Compugraphic Corporation, MUMPS is a trademark of Massachusetts General Hospital, Macintosh is a trademark of licensed to Apple Computer, Inc., Multibus is a registered mark of Intel Corporation, 8086 is a trademark Intel Corporation, VENIX is a trademark of Ventur Com., Inc, Appletalk is a trademark of Apple Computers, Inc., INGRES is a trademark of Relational Technology, Inc..

The articles are the responsibility of the authors and therefore, DECUS and Digital Equipment Corporations, assume no responsibility or liability for articles or information appearing in the document.
The views herein expressed are those of the authors and do not necessarily express the views of DECUS or Digital Equipment Corporation.

FOREWARD

This Proceedings is published by DECUS (Digital Equipment Computer Users Society), a world-wide society of users of computers, computer peripheral equipment and software manufactured by Digital Equipment Corporation. The U.S. Chapter of DECUS has approximately 41,000 active members.

DECUS maintains a library of programs for exchange among members and organizes meetings on local, national and international levels to fulfill its primary functions of advancing the art of computation and providing a means of interchange of information and ideas among members. Two major technical symposia are held annually in the United States.

For information on the availability of back issues of Proceedings as well as forthcoming DECUS symposia, contact the following:

DECUS U.S. Chapter
Digital Equipment Corporation
219 Boston Post Road, BP02
Marlboro, MA 01752-1850

All issues of past Proceedings are available on microfilm from:

University Microfilms International
300 North Zeeb Road
Ann Arbor, MI 48106

PREFACE

This volume of the Proceedings contains papers which were presented at the Fall 1985 Symposium of the Digital Equipment Computer Users Society.

The Fall 1985 Symposium was held at the Disneyland Hotel and Convention Center, in Anaheim, California, from December 9 through December 14, 1985.

Five thousand seven hundred and sixteen DECUS members converged on the Disneyland Hotel, and Disneyland itself that week. They attended birds-of-a-feather sessions, 70 pre-symposium seminars, and approximately 1000 presentations.

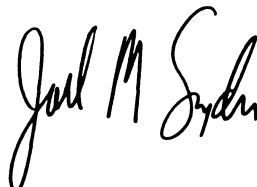
In Anaheim, Digital and DECUS emphasized networks. Increasingly, networks are moving out of the computer room and into the office, out of the office and into the corporate backbone. Most SIGs had sessions which highlighted the benefits and problems of using their products in networks. Networks, whether composed of people or computers, are becoming more important in the global scheme of things.

The National Science Foundation is sponsoring a super-computer project by linking United States universities to computer centers with supercomputers. The interconnection of multiple campus networks that this project requires raises many questions—who is responsible for naming and addressing? How will routing be done? What implications are there for networks that the universities may be connected to? None of these are answered easily. This is the kind of problem that is becoming common, the interconnection of large networks. Digital has Easynet, 40,000 computers in one DECnet network. DECUS has "Usernets," thousands of smaller networks. By working together, Digital and DECUS can provide tools and technology to ease the interconnection of networks.

Each day, more and more Digital computers are linked into larger and larger networks, and networks begin to expand until they touch one another. The global internet, in which any system can communicate with any other system, is quickly moving upon us. Communications across the city, the state, the country, and the world are more and more commonplace; soon an electronic mail message from Sydney will be as common as one from New York. Computer networks will expand the number of people we talk to, and change the way we do the talking.

In this time of high global tensions, the increased availability of a direct channel between one human being and another will help our ability to speak to one another, to reach out to one another, and to understand one another. It is not easy to fight with someone that you talk to every day; it is impossible to have conflict with someone you truly understand. By linking ourselves together electronically, we will strengthen the bonds that hold us together as one people, sharing one planet, in peace.

My thanks on behalf to the entire membership go out to Mr. Jack Cundiff and Dr. Jeff Jalbert, the DECUS volunteers who led the Anaheim symposium effort. Their intensive weeks of work with Ms. Nancy Wilga, Ms. Joanie Mann, and Ms. Gloria Caputo of the DECUS staff made our national meeting truly impressive. Their experience and leadership are sincerely appreciated. For her special work on the Proceedings, I would also like to thank my colleague, DECUS staff member Cheryl Smith.



Proceedings Editor
DECUS U. S. Chapter Publications Committee



TABLE OF CONTENTS

ARTICLE	PAGE	ARTICLE	PAGE
ARTIFICIAL INTELLIGENCE SIG			
A VAX LISP Expert System for Analyzing Security Alarm Data Sarah Townsend	3		
The Implementation of the Fault Localization, Assessment, and Repair Expert System (FLARES)—Tools, Languages, and Issues Susan E. Bill-Wray, John T. Williamson	7		
BUSINESS APPLICATIONS SIG			
Design Principles for Software Manufacturing Tools Paul G. Bassett	15		
Decision Support Systems and DEC Micros Kuriakose Athappilly	25		
DATA ACQUISITION, ANALYSIS, RESEARCH, AND CONTROL SIG			
LISREL: An Application, An Explanation Leanne Whiteside	33		
System Chargeback and Resource Tracking using RS/1 Robert B. Goldstein, Gertrude Stabiner	55		
Development of a Digital Interactive Controlled Evaluation System Scott B. Eckert, Robert L. Ewing, Gary B. Lamont	61		
Customizing RS/1 a GA Technologies Aram K. Kevorkian	65		
PRO: A Multiple Priority, Multitasking Process Control System and Language as Implemented in an Inhalation Exposure Facility Edwin R. Lappi, Leon C. Walsh	69		
Expert System Usage in the Laboratory Thomas A. Turano	77		
DATA MANAGEMENT SIG			
Encryption for Beginners Bart Z. Lederman	93		
Introduction to VAX Information Architecture Databases Eric A. Newcomer	102		
The Implementation of an Academic Faculty and Student Database Management System David A. Gaitros, Robert L. Weing, Gary B. Lamont	111		
Data Management System for Academic Personnel Admin- stration Lisa M. Rotunni	115		
SQL/DSRI and QUEL/DSRI Implementation John D. Markel	121		
DBMS-20 Sorted Set Structures Jeffrey S. Finton, David W. Chilson	129		
A Programmer's Database System for Software Development and Maintenance Rachel Schwab	159		
MATRIX: A File Organization for Image Processing Philippe E. Collard	163		
DATATRIEVE SIG			
DATATRIEVE-11 to VAX DATATRIEVE Conversion Panel Joe H. Gallagher, Bart Z. Lederman	173		
EDUSIG			
Computerized Decision Support for College Administrators Walter H. Frey, Vernon M. Cline	181		
Evaluation, Selecting, and Implementing an On-Line Library Card Catalog Rob Robinson	185		
DAL Magic—Some Surprising Features of DAL Pete Boysen	193		
8088 Macro Assembler on the Rainbow Micro Computer Robert S. Workman	211		
GRAPHICS APPLICATIONS SIG			
TCHART: Development of a Device Independent Chart Draw- ing Program Judith Bardell	219		
A Software Display System for Medical Image Processing Luc Bidaut	225		
LARGE SYSTEMS SIG			
Using Personal Computers with System 1022 Randolph M. Pacetti	237		
VMS for TOPS Users: End User Interface Kathy Rosenbluh	247		
VMS for TOPS Users: Program Development Kathy Rosenbluh	253		
TOPS to VMS Business Application: TOPS-10/VAX Perfor- mance Comparison Frank Francois, Ralph Bender	257		
LISP on 36-Bit Systems Randolph M. Pacetti	277		
TOPS-20 Directions Donald A. Kassebaum	279		
TOPS-20 V6.1 for Users Carla J. Rissmeyer	281		
TOPS-20 V6.1 for System Administrators	283		
TOPS-20 V6.1 for Systems Programmers Douglas Bigelow	285		
Ethernet Planning and Installation Considerations Donald A. Kassebaum	287		
Hardware Planning for Integration Customers Gary Bremer	317		
TOPS/VMS Performance Comparison Gary Bremer	319		
TOPS-10/20 and VMS Layered Product Comparisons Gary Bremer	325		



Sarah J. Townsend
Institute for Defense Analyses
Alexandria, Virginia

Version 4 of VAX VMS provides auditing capabilities designed to notify managers of security attacks and breaks. This paper outlines a prototype of an expert system designed to analyze these security alarm messages in order to recognize events causing a break as well as persons attempting an attack.

This paper describes the workings of this LISP program, in more detail than general artificial intelligence theory, but including no LISP code. Plans for future enhancements of the system are given. Suggestions to DEC for security auditing improvements are also included.

The research for this paper was done at the University of Maryland at Baltimore County on a VAX 8600 running VMS Version 4.2. I would like to thank them for their support of this research, especially Jack Seuss of the Computer Center at UMBC who made computer resources available for this research.

Much of the security theory and some of the A.I. theory used in the research described in this paper is from work done by Dr. David J. Slater (UMBC Instructor) and I would like to thank him for his many valuable contributions and suggestions.

Before the main description of the system begins, a few definitions are needed.

Definitions

Prototype

A prototype has the following properties:

- A. It demonstrates the basic workings of the system.
- B. It is extensible. That is, there is nothing in the prototype system that only works because what is being handled is a subcase of the total problem.
- C. It gives a clear picture of how a more fully developed system will operate.

Event

An event is meant to mean anything recordable that might relate to the security of the system. Currently this includes only the security alarm messages generated by the audit command (new in VMS Version 4). Later it will include other data sources, such as monitor and accounting data.

Thread

Threads are chains of events which are linked in one of two manners. Either they are chains of events all of which are similar in some manner, as

in A below, or they are chains of events, such that each event is a precondition for some subsequent events, as shown in B below.

Frame

A frame is a description of a possible type of security problem, with descriptions of what types of threads are necessary for, and what types of threads are relevant to this type of problem. This is the standard usage of the term in artificial intelligence.

Picture

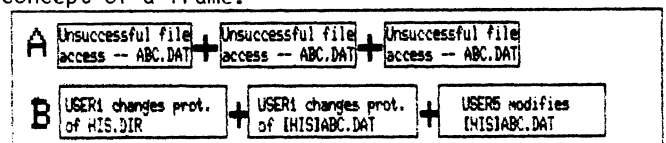
A picture is an instantiation of a frame. That is, it is a collection of threads that match some notion of a security problem.

Why LISP?

This system was written in VAX COMMON LISP, because LISP is the language most adapted to our application. Two features of LISP make it particularly suited to our needs:

Flexibility

Traditional expert systems involve well defined problems which are understood by experts and solely use knowledge from experts. This program deals with heuristics imitating common sense in a poorly defined problem which is not well understood, and in which many of the experts disagree. Thus, it was clear from the start that the evolution of new expert system techniques was going to be part of this research. Specifically, the ability to redefine a connection and change the type of logic used in making connections was essential. Only in LISP are relationships defined in such a way as to allow logic containing relationships of relationships. It was also necessary to extend the traditional concept of a frame.



Tools

LISP provided tools which easily facilitate the maintenance of events and the manipulation of frames even though both items are complicated structures.

Overview

The system first creates a list of events, Figure 1.

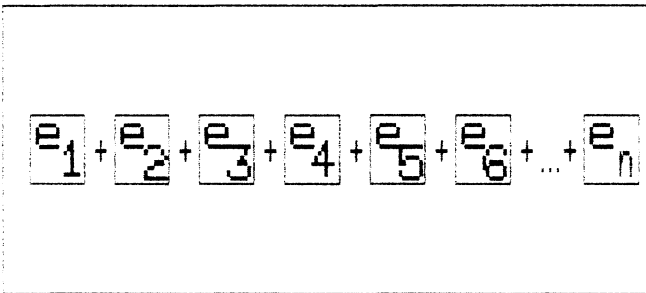


Figure 1

These events are then checked for connections to form threads, Figure 2. Any number of events may be in a single thread. An event may be in any number of threads. A single event may be a thread. An example is modification of the operator log file. If an unauthorized person modifies that file, no other evidence of the attack may exist.

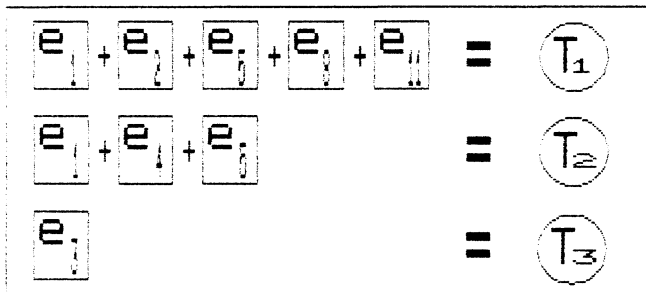


Figure 2

A good way to picture threads is as different shaped objects, like those depicted in Figure 3. Each shape represents a different type of thread. For example, a series of illegal login attempts may be thought of as an oval.

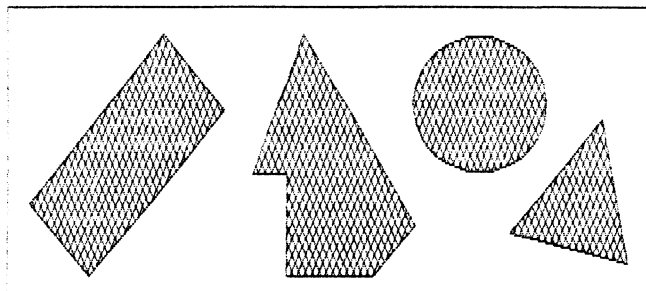


Figure 3

A frame can then be pictured as a block with different shaped holes in it (Figure 4). Only a particular shape of thread fits into each of the frame's holes.

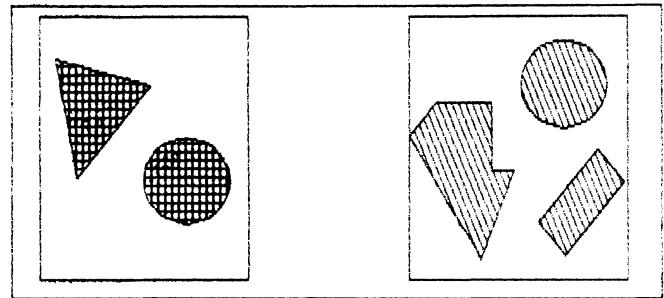


Figure 4

For each thread it is then determined which frames these threads may be used in, Figure 5.

Each time such a frame is found, an attempt is made to find the remaining threads necessary to form a picture from this frame, Figure 6. Finally, a search is made for threads which help to clarify the picture created.

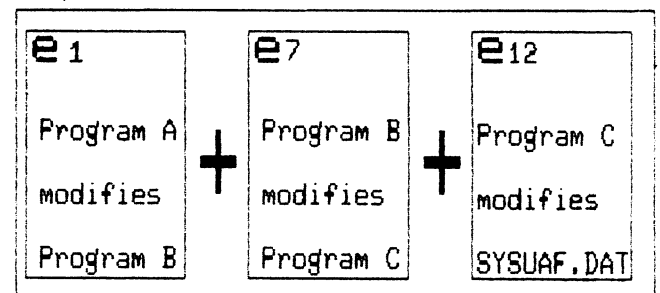
The final part of the system is a description facility, which prints out a report on each of the pictures. The user decides the level of detail printed.

Thread Creation

Each event is linked to objects in the system based on any piece of information connected with the event. These include:

1. User name
2. Device
3. File name
4. Time

Two routines then look at this structure of events, attempting to create threads. The first such routine simply looks for a multiplicity of events relating to the same object. The second routine looks for chained events where the attacked object of the first event is the attacker in a later event, Example 1.



Example 1

The actual method of establishing connections is a series of complex heuristics. Some connections are simple pattern matching. Others recognize more complex relationships, such as the fact that a particular program behaves differently when activated by a privileged user. Future versions of the system will allow thread recognition to be user tunable depending on the environment. A site where all software used is written in-house may decide to ignore functions that test for Trojan Horse programs, for example.

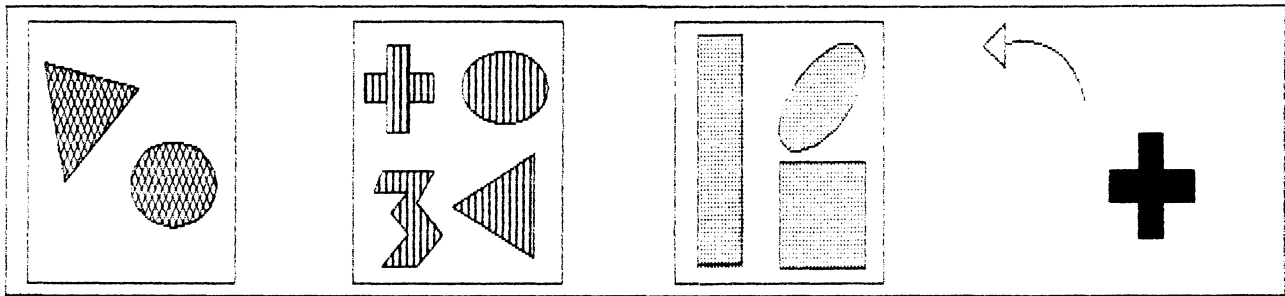


Figure 5

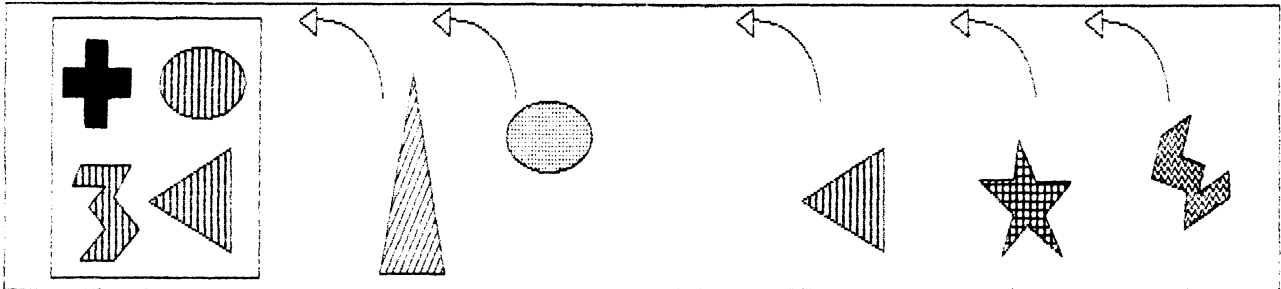
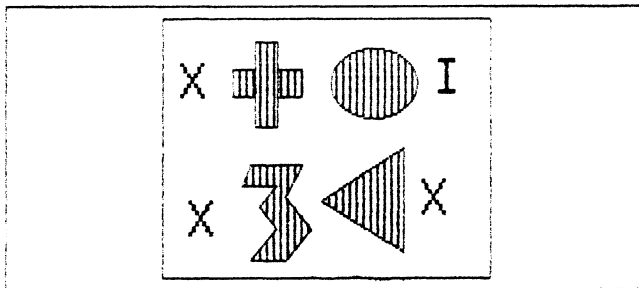


Figure 6

Creation of Pictures

In our program, the holes of a frame are actually heuristic routines. They compare the thread passed to the routine with the shape they want, and return a flag designating whether the thread matched.

Not all the holes of a frame need to be filled for a picture to exist. Some of the holes are flagged as necessary (those labeled I in Example 2), while others are flagged such that only one of a group of holes must be filled for the frame to be full (those labeled X in Example 2). For example, the oval in Example 2 may be modification of the system startup file. The other holes then represent different methods of accomplishing the attack: the plus sign changing the file's protection; the triangle changing the UIC of the user; and the squiggly shape changing mode to kernel.



Example 2

Each hole labeled X in Example 2 is significant unto itself. Even though they are optional in this frame, each hole is part of another frame where it is necessary (labeled I) and where other holes explain how the attack was accomplished.

Holes also have a number associated with them designating the number of occurrences of the thread to be counted as part of a single picture. Infinitely many illegal login attempts are part of the same

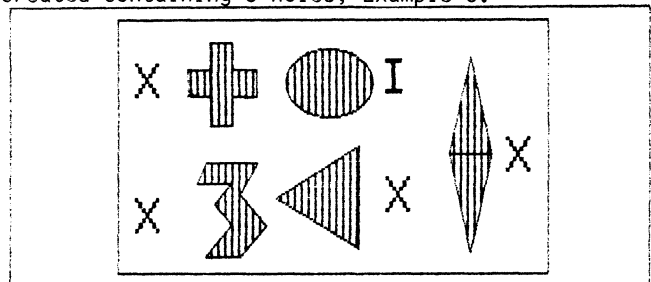
picture, for example, while each occurrence of SYSUAF.DAT modification is a separate picture.

Future versions of our program will include the concept of dynamic frames. This will allow a frame to create a new frame if data shows the old one to be inadequate.

The basic technique used to create adjustable frames is that a three-valued recognition function is associated with each frame. This function can return the traditional values of match and failure to match. However, it can also return a value which indicates that the matching criteria may need to be extended.

This returned value invokes a series of procedures which try to determine what extensions to the frame or matching criteria would enable a match. Then a second series of procedures is invoked which attempt to determine which, if any, of these extensions are reasonable. An extension may be the addition of a hole, the removal of a hole, or the initiation of monitoring to collect relevant data.

To extend Example 2, suppose the system startup file was modified but none of the holes labeled X are filled. The program will then search through events for anything which is connected to the system startup file or the attacker. If an entry is found, such as gaining SYSPRIV privilege, then a new frame is created containing 5 holes, Example 3.



Example 3

Reporting

One of the features of this system is that it gives a summary report classifying pictures as to their nature. Some of the possible natures of pictures are:

1. Definite successful breach of system.
2. Possible breach of system.
3. Dangerous attack.
4. Attack of lesser danger.
5. Suspicious unexplained happenings.
6. User who probably needs greater system education.

This summary report is short, approximately one line per entry. The security manager may then request reports at higher levels of detail on any of the pictures.

Areas of Planned Improvement

* At present the only events the system looks at are those signaled by the VMS security auditing mechanism. Additional monitoring planned includes:

- A. Accounting records.
- B. Monitor records.
- C. Detached processes created specifically by this system.
- D. A user-history database of each user's typical activities.

* At present the system does not have the capability of increasing the monitoring of areas where it looks like there is the possibility of suspicious activity.

* At present the system runs extremely slowly.

* The heuristics for establishing connections, and the types and capabilities of the frames, will be continually enhanced.

* The concept of dynamic frames as described earlier will be added.

* Thread-making tunability will be included to allow the system to closely fit individual sites.

Suggestions for DEC Security Auditing Improvement

During the development of this system, we noticed several areas where DEC's security auditing might be improved:

* There should be several classes of security audit so that one could reply/disable a terminal for some but not all security events. This is important because most events are meaningful only when viewed in a larger picture, while there are a few events of which one might want immediate notification.

* Allowing security events to be recorded in a file other than OPERATOR.LOG.

*Having security events recorded in a compressed form.

*Having some form of hardware that makes it impossible for a user no matter how privileged to alter the security log (such as a tape drive that cannot be rewound under software control).

THE IMPLEMENTATION OF THE FAULT LOCALIZATION, ASSESSMENT AND REPAIR
EXPERT SYSTEM (FLARES) -- TOOLS, LANGUAGES, AND ISSUES

Susan E. Bill-Wray
John T. Williamson
Combat Control Systems Department
Naval Underwater Systems Center
Newport, R.I.

AVAILABLE FOR PUBLIC RELEASE

ABSTRACT

Digital Equipment Corporation's BLISS-based OPS5, Version 1, is currently being used in the development of an expert system named FLARES. The authors' experience with OPS5 has led to the discovery of five maxims on the use of OPS5. The experience has also provided the basis upon which to form an evaluation of OPS5 as an expert system development language. This paper presents the FLARES activity, the five maxims on OPS5, and the evaluation.

1. INTRODUCTION

FLARES (Fault Localization, Assessment and Repair Expert System) is a knowledge-based system currently under development at the Naval Underwater Systems Center. FLARES has the three functions of diagnostic reasoning, system assessment, and equipment repair assistance. The major development language is Digital Equipment Corporation's BLISS-based OPS5, Version 1. The FLARES project has provided the authors with experience in the use of OPS5 which is being shared through this paper. First, in sections 2 and 3, an overview of the FLARES domain and the system design are presented. Section 4 describes the OPS5 language in preparation for more detailed discussions on the lessons that have been learned about using OPS5. These learned lessons are presented in section 5. An evaluation of OPS5 as an expert system development language is given in section 6, and a summary is given in section 7.

2. THE FLARES DOMAIN

FLARES is a knowledge-based expert system designed to aid submarine personnel in the troubleshooting of faulted equipment, to perform an assessment of the degraded equipment, and to provide repair guidance. One of the duties of submarine personnel is the operation and maintenance of complex electronic equipment. During times of equipment failure, this duty requires the ability to locate hardware faults, assess the operational status of equipment that is in a degraded mode, and successfully repair the equipment. Knowledge is needed about electronic equipment, the interpretation of fault indicators, troubleshooting procedures, output signal requirements for various uses of the equipment, current and possible equipment configurations, and maintenance and repair

procedures.

The particular piece of equipment for which FLARES is targeted contains self-diagnostic tests which are used in the troubleshooting procedures. These tests, however, are not exhaustive, and they do not, by themselves, always indicate a single source of the fault. The tests provide fault codes when electrical malfunctions are detected in the circuitry being tested. Associated with each fault code is a list of the circuit cards that could have caused the test to fail. Tables of the available diagnostic tests, their purposes, their possible fault codes, and the circuit cards associated with each fault code exist for the operator's use. Troubleshooting this equipment involves an iterative process of the interpretation of fault codes and the execution of appropriate diagnostic tests in an attempt to minimize the number of suspected cards.

This equipment also has the capability of graceful degradation. It is a subpart of a larger, multi-equipment system. The system is used to perform a number of functions, and each function may require only a subset of the equipment's generated signals. If the required signals are not affected by the equipment's fault, then the equipment, and therefore the overall system, will still be operational. After the successful diagnosis of the cause of failure, an assessment can be made of the severity of the equipment's faults and the impact on the system functions. This is accomplished through knowledge of possible re-configurations and the system's signal requirements of the equipment. If the assessment is made that the equipment is in an operational state, then the issue of repair can be delayed.

Repair of this equipment is relatively

straightforward. The faulted card is located within the equipment, removed, and replaced with a spare. Of course, a spare unit in working condition is not always available, especially aboard a submarine; so an updated inventory of repair supplies should be made before attempting equipment repair. If system repair is not an option, then assessment of the equipment's operational status takes on greater importance. FLARES has been designed to closely follow the characteristics of this domain. The design is presented in the next section.

3. SYSTEM DESIGN OF FLARES

The design of FLARES incorporates three sections that correlate with the sub-tasks of equipment operation and maintenance. (See Figure 1 for a pictorial view of the FLARES design.) FLARES has a Fault Localization section that performs the diagnostic reasoning needed to troubleshoot the equipment. It also has a System Assessment section that determines the operational status of the degraded equipment given the conclusion from the first section. The last section, System Repair, provides maintenance and repair support such as instructions, diagrams, inventory updates, etc. The user can communicate with each of the three sections through a graphical user interface. Upon completion of fault localization, the user has the option of having FLARES perform the system assessment and then, if desired, continue to the repair of the system. Alternatively, the user may have FLARES go directly to the system repair section avoiding the assessment section altogether.

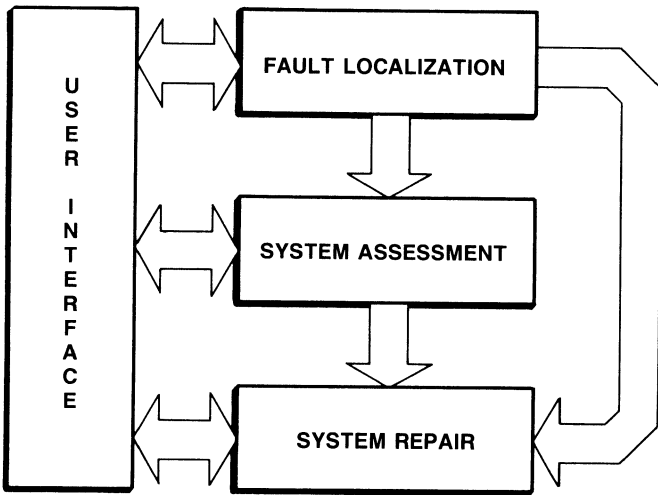


Figure 1. Overview of FLARES Design

FLARES is being developed on a Digital Equipment Corporation VAX 11/780, VMS 4.2. The main development language is DEC's BLISS-based OPS5, Version 1. DEC's DATATRIEVE database formatting language is being used for storage and retrieval

of the diagnostic tables. The user interface consists of a graphics display with a touch screen, an optional voice recognition unit, and possibly a laser videodisc system. FORTRAN 77 is being used for the interfaces between equipment, languages, and tools (see Figure 2).

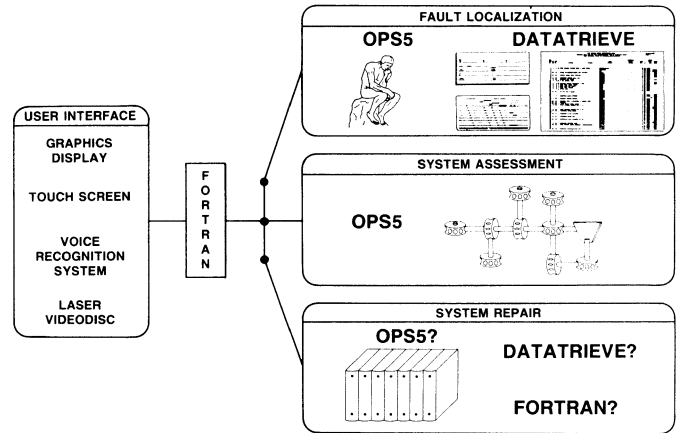


Figure 2. FLARES Implementation Languages and Tools

The knowledge-based portions of FLARES, the Fault Localization and System Assessment sections, are being implemented in OPS5. However, these two sections utilize OPS5 for different types of functions. Within the Fault Localization section, OPS5 is being used to perform interpretation and diagnostic reasoning. Interfacing with DATATRIEVE for supporting data from the diagnostic tables, the OPS5 program will interpret what is currently known about the equipment failure, suggest the next test(s) to be run by the user, and integrate any new information, in an attempt to correctly diagnose the faulted card that is causing the failure.

In the System Assessment section OPS5 is being used to create a model of the equipment. The model traces the functional flow of electronic signals through the equipment. The assessment takes place by running the OPS5 model with those signals that have not been affected by the faulted card, and by examining the output signals generated. These output signals are compared to those required for operational use of the equipment to determine the equipment's functional utility.

The System Repair section of FLARES is still in the design phase. It may be implemented with OPS5, DATATRIEVE, and/or FORTRAN 77, and it may use a laser videodisc system, depending on the extent to which FLARES will aid the user in repairing the equipment.

The extensive use of OPS5 for the development of FLARES has provided a basis upon which to form an

evaluation of the language, and has led to the discovery of five maxims about the use of OPS5. These maxims and the evaluation are presented later in the paper. First, however, a description of the OPS5 language is provided for background to these next topics.

4. THE OPS5 PROGRAMMING LANGUAGE

OPS5 is a forward-chaining, production-rule language. It executes in a cyclic manner of pattern matching, conflict resolution, and rule activation (see Figure 3). An OPS5 program consists of static production rules in an "IF - THEN" format and dynamic working memory elements that indicate the current status of its "world".

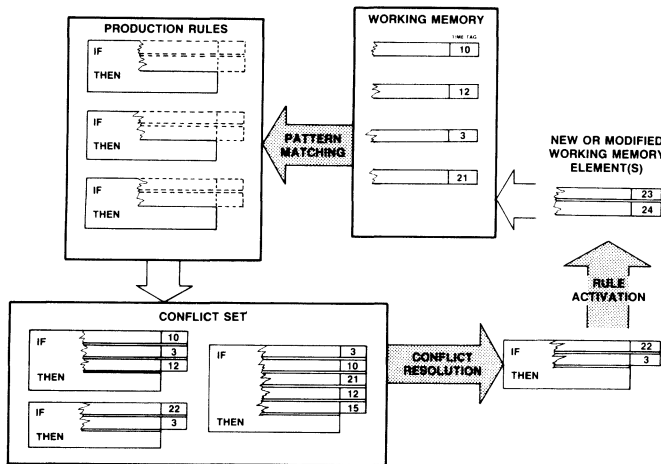


Figure 3. Pictorial View of OPS5

The working memory elements each have an associated time tag. This is a sequential assignment of a number to the working memory element upon its creation. The time tag designates the relative recency of the working memory element. The working memory elements are pattern matched with the conditions in the IF portions of all the production rules. Any production rule that has its complete IF portion satisfied is placed in a conflict set. At the completion of pattern matching, a conflict resolution strategy is invoked to resolve the rules in the conflict set down to one rule that will be chosen for activation. The selected rule is activated; the statements in the rule's THEN portion are executed. These statements usually update the working memory by adding, deleting, and modifying working memory elements. The cycle is then repeated, starting again by pattern matching between the now updated working memory and the production rules.

This cycle is called data driven, or forward chaining, because of the influence the updated working memory elements have on the selection of a production rule for activation. The conflict resolution strategies utilize the recency of working memory elements matched with a rule (indicated by the time tags) for their selection. The strategies also consider the size of the IF portion of an instantiated (or matched)

production rule, and the specificity (number) of the relational tests used during pattern matching. However, the matching working memory elements' recencies are the first basis of selection.

OPS5 offers two conflict resolution strategies, LEX and MEA. (The name LEX comes from the strategy's similarity to lexicographic ordering, and MEA from the term means-ends analysis.) They are basically the same, with the exception that MEA places heavier emphasis on the recency of the working memory element matched with the first condition in the IF portion of each of the instantiated production rules.

5. LESSONS LEARNED THROUGH EXPERIENCE WITH OPS5

Five maxims on the use of OPS5 were discovered during the development of FLARES. Figure 4 lists these maxims. This section of the paper addresses each of these maxims for the benefit of others who are thinking of, or are currently, using OPS5.

1. OPS5 IS DECEIVINGLY SIMPLE
2. CONTROL ONLY ENOUGH TO GET THE JOB DONE
3. OPS5 IS NOT AN ISLAND
4. ALL IS NOT BLISS
5. DETAILED OPS5 DOCUMENTATION IS NEEDED FOR RAPID DEVELOPMENT OF PROGRAMMING TECHNIQUES

Figure 4. OPS5 Maxims

5.1. OPS5 Is Deceivingly Simple

The first maxim discovered is that OPS5 is not as simple as it initially seems. Upon first inspection of OPS5 with its simple cyclic performance of pattern matching, conflict resolution, and working memory update, and its symbolic syntax for production rules and working memory elements, one can conclude that OPS5 is an easy, straightforward language. In a sense, it is, in that it functions in the forward-chaining manner which is readily understood. However, the conflict resolution strategies, which implement the forward-chaining program flow, are more complex than may be originally assumed. The resolution tests employed by LEX and MEA tightly couple the format of the production rules with the rule selection process. For example, such things as the order in which the programmer writes the condition elements (in a rule's IF section), and the order in which the statements in the THEN section are written can affect the outcome of the strategies' tests and therefore the selection of the next rule for activation. The programmer unaware of the subtleties of the resolution strategies may find unexpected results when the program is executed. The causes behind the program's unexpected performance, however, can be revealed through closer examination of the conflict resolution strategies. This examination should be completed before undertaking a major programming effort to avoid frustrating trial-and-error learning.

5.2. Control Only Enough To Get The Job Done

The second maxim addresses the issue of controlling the flow of an OPS5 program: control only enough to get the job done. Theoretically, the beauty of the data-driven program flow lies in the influence of new data. As new data is created from the old data, the program extends what it "knows" by using what it has recently acquired. Placing control within the data-driven program flow will detract from this elegance. The initial selection of OPS5 implies a desire for this type of program flow. However, it is often the case that a programmer needs to influence the flow for efficiency's sake, or to implement some sequentiality. During the development of FLARES, it was found that control of the program flow can be accomplished but that it must be done with care, adding only enough control to produce a manageable application.

By paying close attention to the selection criteria of the chosen conflict resolution strategy, the programmer can influence the activation of specific production rules. This control can be implemented through the careful tailoring of the production rules' IF and THEN sections. Within the IF sections control can be accomplished by altering the number of condition elements, the specificity of the condition elements, and the ordering of the elements to force a production rule to best meet the conflict resolution strategy's criteria.

Control can also be accomplished from the THEN section of the production rules. Because the recency of working memory elements has such an influential role in the conflict resolution strategies, the selection of a production rule can be controlled by controlling the order in which elements are added to working memory. This will insure that a particular working memory element will have the highest time tag during the next cycle. Careful ordering of the MAKE and MODIFY statements in the production rule's THEN section will accomplish this.

Control can also be added to an OPS5 program on a more global scale. OPS5 allows for actions that take place on the whole of working memory, such as 1) saving a copy of working memory in a file, 2) adding a previously saved state to the current working memory, and 3) restoring working memory to a previous state. Alteration of the global state of working memory could change the subset of production rules placed in the conflict set and thus affect the overall performance of the program.

Of course, the most obvious control method is through the choice of a conflict resolution strategy, either LEX or MEA. Be aware, however, the choice of a strategy and the use of the other control methods are not independent. To control an OPS5 program's flow is to pre-plan the selection of a production rule for activation based on the criteria of the strategy being used.

5.3. OPS5 Is Not An Island

The third maxim discovered during the development of FLARES is that OPS5 is not an island, that is,

a program written in OPS5 is not independent of the remaining computing environment. DEC's OPS5 contains routines that are used to communicate with utility routines written in other languages. An OPS5 program can invoke a utility routine, and it can pass working memory elements to and from the routines. Many languages used for knowledge-based systems view the environment as a closed world, making it difficult to work around those aspects that the language cannot perform well. With accessibility to the world outside OPS5, an OPS5 program can take advantage of the capabilities offered by other software (and therefore hardware, such as graphic displays, as well). This broadens the application areas of OPS5 and leads to more productive use.

5.4. All Is Not BLISS

As previously stated, FLARES is being written in DEC's BLISS-based OPS5 (Version 1). A LISP-based OPS5 is also available. While the use of the BLISS-based version has some advantages, there are also associated disadvantages, thus the fourth maxim: all is not BLISS.

The advantages include its speed and the accessibility of interim files. The BLISS-based OPS5 production rules are pre-compiled. This affords a quick running program. The compilation results in separate intermediate object files of the IF and THEN sections of the rules. Through the development of FLARES it was found that interesting use can be made of these intermediate files. For example, one copy of the compiled file of the IF sections of some production rules can be alternately linked with different versions of the THEN sections' interim files, providing variations of a production rule program. Other advantages have been previously mentioned under different topics, for example, the capability to store working memory in a file for future use, and the capability to communicate with the outside computing environment.

One disadvantage of the BLISS-based OPS5 lies in its limited support of mathematical functions. It offers only the integer functions of addition, subtraction, multiplication and modular division; no floating point arithmetic and no high-level functions are supported.

Two other disadvantages deal with the BLISS-based OPS5 treatment of production rules. First, while the working memory elements can be viewed by the user, the production rules cannot. Once the rules are compiled the programmer cannot view them to determine the compilation outcome. To be able to do so would aid in debugging the program. Secondly, it has been stated that the LISP-based version of OPS5 has the capability of creating new production rules from inside a running OPS5 program. This option does not exist with the BLISS-based version. This capability is desirable for the development of more extensive and powerful programs.

The last disadvantage arises in the execution order of the statements in the THEN section of an activated rule. The BLISS-based OPS5 has a pre-defined order in which it executes statements

that alter working memory. Any actions that add to working memory are executed first, then any actions that save working memory are executed, followed by the execution of any actions that delete from working memory. This pre-defined ordering removes any control the programmer had over the order of actions and can cause some unexpected happenings. For example, a specific incident occurred where a production rule was written whose action was to delete some elements from working memory and then to save the state of working memory in a file. It was later discovered that the saving of working memory happened before the deletions, thus saving an unwanted working-memory state. To accomplish the desired result required writing one logical production rule as two separate rules.

5.5. Detailed OPS5 Documentation Is Needed For Rapid Development Of Programming Techniques

The final, and perhaps hardest, lesson learned through experience with DEC's OPS5 is the need for adequate documentation. While the documentation provided by DEC [1,2] is a good dictionary of commands, it is not sufficient for learning how to program with OPS5. The documentation lacks details on the basic workings of OPS5 (i.e., the conflict resolution strategies) and provided an insufficient number of examples. This led to an increase in the amount of time required to become proficient in programming in OPS5, and time lost in the development of application programs. However, a good text book does exist. Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming, by Brownston, et al., [3] is a good resource for learning to program with OPS5. It has numerous examples and sufficient detail for both novices and experienced OPS5 programmers.

6. EVALUATION OF OPS5

An evaluation of DEC's BLISS-based OPS5, Version 1, has been made based on its use during the development of FLARES. Overall, OPS5 is a versatile tool that allows for the rapid prototyping and development of knowledge-based systems.

OPS5 provides the basics of a knowledge-based system: knowledge representation formats and an inference mechanism. Production rules and the working memory elements' structures are the two knowledge representation formats provided by OPS5. Knowledge that fits the "IF-THEN" format is stored in the production rules. Working memory elements are stored in a representation format that allows the grouping of related knowledge in an attribute-value format. OPS5 also provides the inference mechanism of forward chaining; i.e., it supplies the processes needed to perform pattern matching, conflict resolution, and rule activation.

The syntax of the OPS5 language is not overly cumbersome. There is a relatively small set of statements and syntactic requirements. While initial exposure to OPS5 may cause some confusion, simple examples of the use of OPS5 can give a newcomer the confidence to produce

substantial programs in a short amount of time.

A weakness of OPS5, like most artificial intelligence tools, is that it is not a general tool. It does not offer an extensive selection of knowledge representation formats, and it does not offer optional inference mechanisms. It is a tool designed only for forward-chaining, production-rule systems. However, because OPS5 does provide for communication with programs outside of its own environment, there is the possibility of using OPS5 as a portion of a larger, more extensive system.

7. SUMMARY

In summary, FLARES, a knowledge-based expert system is being developed to aid submarine operators in the operation and maintenance of complex electronic equipment. Its design consists of three sections: Fault Localization, System Assessment, and System Repair. DEC's BLISS-based OPS5, Version 1, is the major development language. OPS5 is being used in FLARES to perform the two functions of diagnostic reasoning and electronic equipment modelling. It is a forward-chaining, production-rule language. The major control of execution lies in the conflict resolution strategies, LEX and MEA. Five maxims about OPS5 have been formulated through this work: 1) OPS5 is deceptively simple, 2) Control only enough to get the job done, 3) OPS5 is not an island, 4) All is not BLISS, and 5) Detailed OPS5 documentation is needed for rapid development of programming techniques.

Based on the authors' experience with OPS5, it can be concluded that OPS5 is a versatile tool that supports the rapid prototyping and development of data-driven, production-rule expert systems. The original selection of OPS5 for the FLARES project was done without much knowledge of OPS5's capabilities, however, the choice was a good one. Given the chance to once again select an artificial intelligence language for FLARES there would be no hesitation to re-choose OPS5. The best way to select an artificial intelligence language or tool for the development of a particular knowledge-based system is through a thorough examination of the domain's requirements for knowledge representation and inference techniques and the capabilities offered by the languages and tools that best meet those needs. OPS5 may not be appropriate for application domains that are complex and extensive. Such domains may not adhere to the forward-chaining inference mechanism and may require more complex knowledge representation methods. But, for those domains, such as that of FLARES, that do follow the data-driven flow, that contain knowledge that can be stored in production rules, and that do not require extensive knowledge representation methods, OPS5 is recommended.

REFERENCES

1. Forgy, OPS5 User's Manual, Authorized reproduction by Digital Equipment Corporation, AA-BH00A-TE, 1981.

2. OPS5 For VAX User's Guide, Digital Equipment Corporation, AA-BH99A-TE, March 1984.
3. Brownston, Farrell, Kant and Martin, Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming, Addison-Wesley Publishing Company, Inc., 1985.

BUSINESS APPLICATIONS SIC

DESIGN PRINCIPLES FOR SOFTWARE MANUFACTURING TOOLS

PAUL G. BASSETT

VICE PRESIDENT - RESEARCH
NETRON INC.
TORONTO, ONTARIO (CANADA)

Abstract

A good solution to the reusable code problem turns out to provide a solid technical basis from which to understand and deal with the production, quality, and maintenance issues currently besieging the software industry. To this end, a software manufacturing methodology called CAPTM (Computer Automated Programming) has been developed. CAP is based on Bassett Frame Technology, which uses a functional-programming concept called a 'frame', motivated in turn by the reusable code problem.

The Introduction explains the necessary background ideas about 'frames'. Section 2 analyses the subtle but important distinction between problem-solving and programming. CAP design principles are then developed which show how to build software tools that support problem-solving through open-ended, structured, program manufacturing techniques. The principles are organized around the flow of program specifications from 'under' to 'optimally', to 'over' specified, machine-executable instructions.

The components of an existing CAP system are described in Section 3, and Section 4 discusses the usage of CAP as a manufacturing technique. Statistics from a case study are presented which indicate that: (a) production quality commercial software can be manufactured at rates exceeding 2000 lines of debugged COBOL per man-day (including systems design time), and (b) less than 10 percent of this code needs to be hand-written/maintained.

1.0 Introduction: The Reusable Code Problem

In the software industry's current cottage industry style, it is common practice to build new programs by "cutting and splicing" pieces of old programs together. This approach demonstrates that

- (a) there is a great deal of potentially reusable code available, and
- (b) it is worth the effort to adapt it rather than starting from scratch [16].

Unfortunately [7],

- (a) the programmer does not have any systematic way of isolating just what portions of programs are relevant;
- (b) the customization process is time-consuming, tedious, and prone to error;
- (c) once the process is finished, both old and new programs must be maintained as if each is completely unique, despite the considerable common functionality. Maintenance effort should be proportional to the novelty in the system, not the number of source statements [4].

The central thesis of this paper is that a good

solution to the reusable code problem turns out to provide a solid technical basis from which to understand and deal with the production, quality, and maintenance issues currently besieging the software industry.

1.1 External Subroutines

It is still widely believed that external subroutines form a satisfactory repository of reusable code. Separately compiled and linked subroutines are obviously useful, but they are limited because there is no graceful or systematic means of effecting:

- (a) local customization of an external subroutine to fit each calling program's particular context of use, and
- (b) global evolution of a subroutine when it must change to benefit all future callers of that subroutine without victimizing current callers.

The fundamental problem is that a subroutine is a representation for a single function which is not adaptable at source-program (function) construction time. It may have considerable run-time flexibility, but at the time of actually molding the subroutine into the program that must use it, an external subroutine by its very nature has no flexi-

bility at all.

1.2 Code Generators

Code generators have been around for years (e.g. RPG) and although they are usually very succinct and expressive, they have never enjoyed widespread use [2,10]. The simplest kind of code generators are those that generate "raw" source code. The problem with those generators is that they are basically "one-shot" tools. Because each generator is an expert at only a part of the overall problem [3,17], programmers must supplement and modify the generated source code to suit their own needs. Having adapted the code, they have no means of reusing the generator without destroying all of their manual modifications. To be more useful, a code generator must allow some follow-on mechanism which can adapt the generated source code automatically, thus allowing reuse of the generator without the loss of the customizations.

More sophisticated code generators typically supply "user exits" for handling this problem. These provide linkage to separately compiled, external subroutines which can usually be written in a variety of general purpose languages. The trouble is that:

- (a) this is always an additive technique; there is no way to change or remove generated functionality;
- (b) predefined interfaces often omit information that is essential in the customization (the "black box" effect);
- (c) all non-procedural parts of the generated code, such as data declarations, are simply unavailable for customization.

A proper solution requires generators to provide for automatic customization of generated code (not just run-time communication with generated modules).

1.3 The Frame Methodology

A frame [13,14] methodology has been developed to address the reusable code problem from the perspectives of both programmers and code generators [3]. A frame is a machine-processable representation of an abstract data type [9], with "abstract" meaning functional [1,3]. Because the data operators are functionals, not functions, frames can accommodate both local customization into an individual program, and global evolution to benefit all future embedding programs. Frames are implemented as files containing a mixture of source code (e.g. COBOL) and (pre-processor) macro commands but quite unlike the proposals of Backus [1] or Evans [8]. This mixture is called frame text.

There are just four macro commands whose essential role is to automate the "cutting and splicing" of programs:

COPY-INSERT allows a frame hierarchy to be copied into a program (by naming the frame at the root of the hierarchy), and causes customizing frame text to be INSERTed anywhere into that hierarchy.

BREAK-DEFAULT defines a named "breakpoint". Breakpoints mark arbitrary places in a frame

where custom frame text can be INSERTed to supplement and/or replace DEFAULT frame functionality.

REPLACE systematically substitutes a specific code string for a generic one (throughout a frame hierarchy). For example, field names, picture clause elements, etc. are generic if they tend to vary from program to program.

SELECT incorporates into a program one frame text module from a set of modules in the frame. SELECTs are like CASE statements (with arbitrary nesting) which operate at text construction time. An important use of SELECT is to automate version control (global evolution).

Frames are written by both analysts and CAP tools. Having code generators produce frames solves the problem of destroying subsequent modifications by automating the "cutting and splicing" of the customizing frame text into the generated frame text.

All customizing frame text for one program is localized into a SPECIFICATION or SPC frame. An SPC governs the entire process of building the compilable source program from its frame components. As will be seen, a methodology incorporating frames at its heart offers a potential for:

- (a) fill-in-the-blanks program specifications (rapid prototyping),
- (b) automation of the process of reusing previously built, high quality software (both human and machine written),
- (c) automatic customization in context,
- (d) maintenance of only what is unique in a program,
- (e) evolution without obsolescence (elimination of unnecessary retrofits),
- (f) painless enforcement of good programming technique (standards).

1.4 Software as a Manufacturing Enterprise

In the next section principles for designing software construction tools are analyzed from the abstract perspective of function spaces. It should be borne in mind, however, that CAP is fundamentally a practical manufacturing paradigm, in which standard frames are the standard sub-assemblies, various code generation steps are the processing operations on basic components (raw materials) to produce fabricated parts, and the CAP text processor operating on the SPC frame is the process of final assembly with any custom options.

2.0 CAP Design Principles

In order to focus on the proper roles to be played by people and machines in the software production process, it is important to understand what is appropriate for the various actors. When higher level languages such as Assembler and FORTRAN were first invented, it was proclaimed that "self-programming" computers had arrived (remember IBM 1401 AUTOCODER ?). In what sense, if any, does a software construction tool differ from a programming

language? Is it really possible to automate programming, or will software tool designers be caught in the same mental trap as the pioneers of higher level languages?

2.1 Problem Solving Versus Programming

Problem solving and programming are related but distinct concepts, and the distinction is critical to the proper design of tools. The job of a problem solver is to find a good function: one which accepts the information specified by the problem, and provides results consistent with the problem's goals and constraints. Concurrently, the problem solver often jumps to the meta-problem of reshaping the problem - and this is precisely the role of the system's analyst.

Sometimes finding a good function can be reduced to matching the problem information to a list of already available functions. None would claim that selecting from a menu is programming. It is the very antithesis: an effective way for non-programmers to obtain the functionality they need, but cannot program. However, for professional problem solvers, life is seldom so kind. Usually many functions must be combined in some non-obvious way to create the desired function.

If a problem can be solved by simply grouping the names of some sub-functions under a new function name, without regard to the order in which these sub-functions are performed, and without regard to how these sub-functions must communicate with each other, then it remains plausible to claim that programming is not involved. This function grouping approach to problem solving turns out to be quite powerful. But first the technique must be further clarified and formalized.

Most problems do not exist in isolation. Recall that a function must be consistent with a problem's constraints in order to qualify as a solution. By varying the constraints in meaningful ways, different but related problems are created which are solved by different but related functions. Each variable constraint is called a degree of freedom. A function space is then implicitly defined to be the set of functions which solve a set of problems which are related to each other by their common degrees of freedom.

Thus degrees of freedom can be used to characterize otherwise implicit function spaces. A degree of freedom is usually specified by expressing one of a (possibly infinite) set of optional sub-functions (constants and variable parameters are simple cases of this). Then any formal notation which allows us to create a function by simply referencing a sub-function from within each degree of freedom independently is, in effect, a language for solving problems without programming.

Conversely, a language rich in irrelevant degrees of freedom (those which are unrelated to problems for which solutions are needed), and poor in relevant degrees of freedom, forces programming to be a part of the problem solving process. Most general purpose computer languages restrict their usage to problem solvers who are also programmers. FORTRAN is a non-programming language to the extent that algebraic expressions solve problems; otherwise, programming must be done.

Now, it would be ideal if problem solvers could always have notations at their disposal which have just the right degrees of freedom for the problems needing solutions. Programming could be completely relegated to the machine. Perhaps when Artificial Intelligence creates a meta-notation with which the machine can develop its own notations for new problem classes, we can all humbly retire. For now I have attacked the more realistic meta-problem of designing tools which eliminate programming for known, highly useful function spaces.

Any given program (function) must usually combine sub-functions from various automated function spaces with sub-functions which are custom-built for the problem. Here it is vital that the "black box" effect be avoided. Black boxes, whose actions are imprecisely understood and have difficult or uncontrollable side effects, are the bane of programmers!* Thus, integrating the automatically produced code with manually produced customizing code must be a convenient, effective process. Computer Automated Programming derives its name from the importance attached to this tool design philosophy.

*Unfortunately many so called fourth generation languages use black boxes [11].

2.2 The Role of Languages

Our industry continues unabated to proliferate languages, and this is both necessary and desirable [17]. The creation of each language is motivated by a desire to reduce the effort of solving, in computer executable form, some class of problems. By distinguishing problem solving from programming, it becomes possible, with respect to a given class of problems, to group languages into three levels: over-specified, optimally-specified, and under-specified.

2.2.1 Optimal-Specification

A language is said to optimally-specify a function space (and hence an associated problem class) if and only if:

- (a) the language is isomorphic to the function space; that is, each distinct function is denoted by only one distinct expression, and only the functions in the space are expressible;
- (b) the degrees of freedom are independent, optimally-specified sub-spaces (of constants, variables, or functions);
- (c) the language's well-formed expressions are the "most compact" (see next paragraph) with respect to all languages satisfying (a) and (b).

In practice, this definition is weakened as follows: (a) is approximated by first designing the language to be virtually one-to-one, then assuming the function space (implied by the language's semantics) to be what was "really meant" by the solutions of the original, unformalized problem class; (b) is approximated first by striving for as much independence as possible, then by applying as many context-sensitive error tests as are practical to any remaining dependent degrees of freedom; while

(c) is ignored as long as the language users are happy.

In practice, such "weak optimally-specified" languages are a realistic approach to problem solving without programming. Functions can usually be defined by simply grouping the names of some sub-functions under a new function name, without regard to the order in which these sub-functions are performed and without regard to how these sub-functions must communicate with each other. Their compilers are called code generators because each generator plays the role of a programmer, converting a declarative, optimal specification into procedural, over-specified code, which itself must be compiled. As has been noted (c.f. Sections 1.3, 2.2.3 and 2.3.2) CAP design principles require the generated code to be in the form of frames.

CAP design strives to optimize the syntax burden for both the human user of an optimally-specified language, and the tools which must also read and write in the language. For people, a special purpose editor should be written. Its special purpose is to be the friendly interface (translator) between the problem solver and the optimized-for-internal-use form. Accordingly, it presents a syntax-suppressed, problem-oriented view of the function space, it provides user-resettable defaults for all the functional parameters (degrees of freedom), and it checks for inconsistent parameter settings whenever possible.

2.2.2 Under-specification

An under-specified language is like an optimally-specified one except that the relationship of well-formed expressions in the language to the possible solution functions is one-to-many. There may be many degrees of freedom which play a secondary or lesser role in the structure of the overall function space. There may be several functions, each expressible in a different language, which must be combined, but whose degrees of freedom intersect or are inter-dependent. In these situations, an under-specified language can be used to quickly "broad brush" the major functional features of the solution. The code generator then employs heuristics to specify one solution function at the optimal level, which is reasonable, and consistent with any overlapping degrees of freedom.

Whereas an optimally-specified language is typically used in a declarative (what, not how) mode, an under-specified language is typically used in a prescriptive mode. That is, the special purpose editor engages in a dialogue of questions and answers, and actively prescribes sub-sets of the degrees of freedom to be specified, depending on answers to previous questions. The code generator then uses heuristic logic (i) to specify all the minor degrees of freedom, creating one or more optimally-specified expressions, and (ii) to specify an SPC frame containing any context-sensitive functionality. The code in (ii) may be necessary to properly combine the functions being expressed in (i). (Thus a generator operating on under-specified expressions may write some over-specified code too.)

Clearly, using an under-specified language is even further from programming than using an optimally-specified language. Unfortunately the one-to-many nature of the language means that the result is

seldom the exact function wanted. However, for this approach to be viable, the result must be an excellent first approximation. That is, the problem solver must be able to spend less time by starting at the under-specified level, then altering the optimally-specified and over-specified results to arrive at the specific function wanted, than by simply starting at the optimally-specified and over-specified levels. One drawback is that inexperienced users of under-specified language tools can all too easily err on the optimistic side, discovering only with hindsight that they would have been better off to begin at the optimally-specified level. On the other hand, providing a rich set of under-specified "front ends" enhances the problem solving power of a CAP system and reduces this drawback.

2.2.3 Over-specified Languages

In an over-specified language, the relationship of well-formed expressions to functions is many-to-one, and properties (b) and (c) of an optimal language do not hold even weakly. Over-specified languages are ubiquitous. For example, every computer's binary or assembly language lacks the syntax to express directly the right degrees of freedom for most of the problem classes to which the machine is applied. And so programming (often done by a compiler) is inevitable at this final stage of problem solving.

(But, with respect to the function space with which the computer hardware can directly deal, assembly language is not over-specified. In this context it is probably an optimally-specified language, or even an under-specified one, if the machine supports micro-programming. Care must be taken not to confuse the issue of which function space best spans a given problem class with the issue of which language best spans a given function space. The function spaces of all useful computers are isomorphic to the (universal) class of all algorithmically solvable (in finite time and space) problems. The assembly language example is really a special case, since each machine architecture forces the consideration of a specific function space. Above the hardware context, each "reasonable" problem class induces, in principle, a "reasonable" (associative, acceptable performance) function space in which to compose solutions for the problems in the class.)

General purpose languages are usually, though not inevitably, over-specified with respect to most formally characterizable function spaces, even universal function spaces. On the other hand, a general purpose language may not be over-specified for several restricted problem classes, which vindicates the language's design. Despite having tools which support under-specified and optimally-specified languages, the source language used in frame text must be a general purpose language in order to permit custom functions to be defined. If a CAP system builder has the luxury to design his own language (which I did not), then I believe it is possible to design one which is a good approximation to being optimally specified with respect to some universal function space.

To sum up the role of languages (see also Sect 2.4), whenever a useful function space can be defined by an optimally-specified language, program-

ming can be relegated to the computer. To further enhance problem solving leverage, multiple under-specified, front-end editor-generator pairs can be built which create optimally-specified expressions. These expressions are processed in turn by editor-generator pairs which create programs at the over-specified level, but maintain them at the optimal level. Any special purpose, custom functionality is kept in the SPC frame which directs the CAP processor in its final assembly tasks of (re)-building the complete source program, then compiling and linking it into executable form.

2.3 The Role of Frames

Frames are used to formalize the common intermediate stage in the program construction process, prior to the frames being combined and customized into a single program (function). There are two reasons for having this stage. First, recognizing the open-ended nature of problem solving, an extensible library of standard frames and templates (see Sect. 2.3.1), together with generated frames, can support custom programming for any problem. Secondly, the ability to mechanize the assembly of a program, given the diversity of its components, depends on bringing them to a common notation.

2.3.1 Standard Frames

As problems are discovered to be related to each other, a standard frame can be evolved to span the implicit function space. Each frame denotes a functional, whose domain defines (using the COPY, and REPLACE commands) the degrees of freedom appropriate to the class of related problems, and whose range (all possible instantiations of the frame text) is the corresponding function space. By fixing those degrees of freedom in various ways, various problems in the class can be solved without programming. This is not to say that programming has been eliminated. Usually real problems refuse to confine themselves to neat, predefined classes. Accordingly, a frame's BREAK points and SELECT clauses constitute open-ended degrees of freedom, where solutions can be arbitrarily extended, if necessary.

Standard frames are used whenever the function space is too limited in scope or usage to warrant a new optimally-specified language. This approach to problem solving is implemented by using templates. A template is an uncustomized SPC frame, and usually spans a hierarchy of frames. It collects in one linear list (a file) all degrees of freedom appropriate for a useful class of problems. The replaceable strings, sub-function selection choices, and insertion points for the frames in the hierarchy constitute a fill-in-the-blanks method of customizing the program. Thus templates and frames together permit problems to be solved in a manner which progressively reduces traditional programming to a minimum, given the open-ended nature of real problems.

To the degree that system design expertise can be stored inside the system, the SPC frame can itself be created by "designer" tools working at the under-specified level (see Section 3.1).

2.3.2 Generated Frames

Certain function spaces have degrees of freedom

which are too dynamic to be represented by fixed, standard frames. Well known examples are screen/-keyboard interfaces and report definitions. For these cases, optimal languages can be developed in association with frame-writing generators.

By generating frames instead of raw source code, open-ended (programming) degrees of freedom become available. Such degrees of freedom are required in the overall problem class, but should be suppressed in the various optimal specification languages. Further customizing can be specified via an SPC without the hand editing or restrictive user exits associated with conventional generators. Basically what has happened is that the editing that would otherwise be necessary to properly customize the generated code has been mechanized. In so doing, we gain both an assembly line style of constructing programs and an ability to maintain the program using its optimally defined pieces (rather than its over-specified code).

2.4 Anatomy of a CAP tool

The following diagram depicts the flow of specifications from the under-specified or Designer level, through the optimally-specified or Customizer level, down to the over-specified or Source and Object levels. Life cycle maintenance is performed with the Customizer (special purpose) editors. Please note that where it refers to screen and report specifications, these are examples of optimally-specified languages with respect to the problems of commercial data processing. A CAP tool may use either, both, or neither of these languages, as well as other notations, if the problems warrant.

3.0 An Actual CAP System

At Netron Inc. a CAP system has been developed for use on a VAX/VMS system and also for the WANG VS computer systems, applied to commercial data processing using COBOL. The following reflects current functionality and some soon-to-be released tools.

3.1 Underspecified level tools

CAPinput - for building interactive file maintenance and data entry programs.
CAPoutput - for building report programs based on general data selection criteria.
CAPfile - for building general files-to-files transforms and interfaces.

These three tools are each structured as shown in Fig. 1. Specification of a complete program requires that an analyst answer a small number of questions (most of which have defaults). The heart of each tool is a frame hierarchy which covers most of the "nooks and crannies" of a formalized problem space. The tool writes a small SPC frame which references the hierarchy and defines the specific function wanted. As well, each tool writes several (weakly) optimal specifications to handle screen/-keyboard interactions and reports. These specifications are generated using heuristics (designed by human analysts) which produce acceptable (if not inspired) specifications.

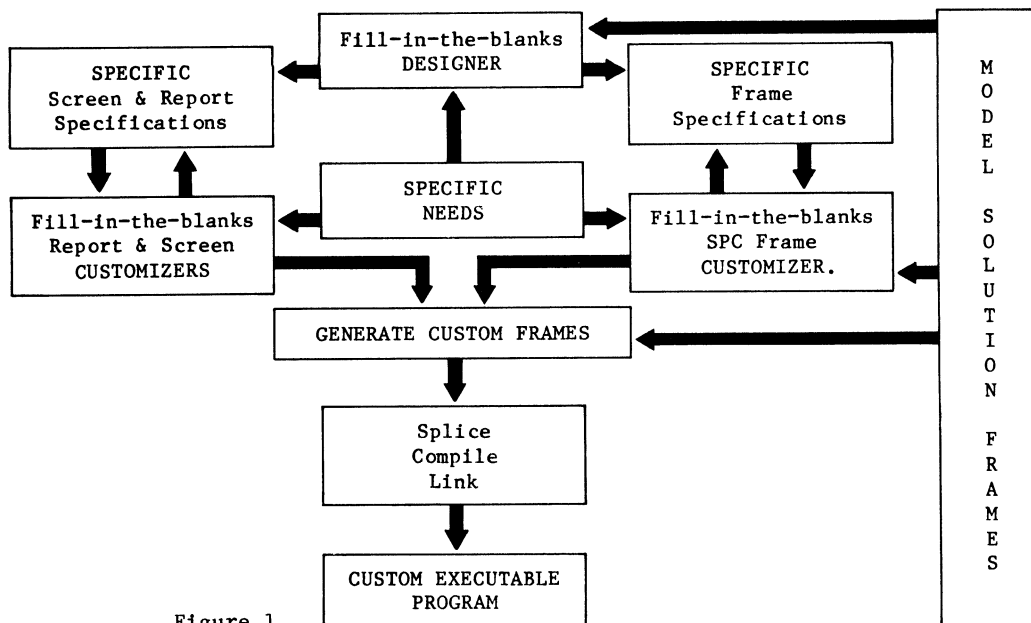


Figure 1

3.2 Optimal Specification level tools

CAPscreen - for designing and maintaining interactive screen/keyboard functionality.

CAPreport - for designing and maintaining report functionality.

The (weakly) optimal notations are used by designer tools and by analysts, either in conjunction with underspecified level tools or independently. CAPinput, CAPoutput, and CAPfile create consistent specifications directly in the optimal notations, whereas people interface indirectly via special purpose editors which suppress syntax and error-check dependent degrees of freedom.

A complete description of these languages is beyond the scope of this paper. Very briefly, independence of degrees of freedom is typified by having screen (report) layout facilities which are completely independent of the attributes of each screen (report) variable. On the other hand, some degrees of freedom are not completely independent. For example, if a variable on a screen is declared as having run-time error checks, and is declared as not being assigned to an internal variable after the operator enters it at run-time, then these two degrees of freedom are in conflict (and must be resolved).

The tools themselves generate frames from the optimal specification. These frames in turn make extensive use of the hierarchy of available CAP frames. Because the frames are written using general purpose (but sadly overspecified) COBOL, the programmer has exact control over the "fine tuning" which his particular application may need in order to convert a functional into the required function.

3.3 Standard Frames

Netron provides an open-ended library of frames, ranging from simple abstract data types to frames which create complete, multiple records per screen, interactive file maintenance programs. These frames are "application independent"; Netron's cus-

tomers add their own frames according to the applications and standards they require. CAPeditor is a special purpose editor for customizing SPC frames based on this library.

Here is a small, ad hoc sample of some standard frames:

- File-maintenance frame
- General batch data entry screen handling frame
- Search screen for CAPinput
- Frame for preparing CAPinput logfile reports
- COBOL FD to data dictionary translator; allows several alternate keys
- Abstract data type for reports using CAPreport
- Multiple screen-frame integrator
- Screen definition abstract data type
- Screen-variable edits
- Left justify strings
- Concatenate two strings, remove trailing blanks from first string
- Set tabs stops on screen
- Abstract data type for all screen attributes
- Allow large numbers of screen attributes to be manipulated at one time
- Convert system time and date to any of three different formats
- Interface to other system data (e.g. USERID, libraries, volumes etc.)
- Check if file exists and optionally create it
- Divisions and section headers needed for COBOL program.
- COBOL SELECT declarations within the INPUT-OUTPUT section
- Provide the COBOL necessary for the FD declaration within the FILE SECTION
- For creating menus with CAPscreen
- For scanning/pattern-matching
- Abstract data type for indexed files
- Abstract data type for sequential files

The CAPframes are the heart of the CAP system. Each frame implements a useful function space whose patterns have been recognized by their appearance in several programs. The frames are organized into a taxonomy which guides the problem solver to the relevant functionality.

4.0 Discussion of Tool Usage

4.1 Types of Users

The consistent application of the "under-optimal--over" design principle offers access potential to the industry's three major user groups: end-users, analysts, and programmers. In CAP's current implementation, it is an analyst-oriented software manufacturing system. The focus has been to provide tools which aid in the manufacture of larger, more complex systems.

CAP could be designed for non-programmers, but few are inclined to cope with open-ended applications building/maintenance which is CAP's main strength. Most people like driving cars and some even enjoy fixing or rebuilding them. But who wants to design and manufacture them?

Because CAP is a manufacturing paradigm, most of the benefits stemming from the organization of a conventional manufacturing enterprise become available to DP shops. In particular, the frame engineering department is quite analogous to a conventional engineering department. A useful division of labor is created. Those responsible for designing and maintaining the organization's inventory of standard software components (frames) can work independently from those charged with getting the application software products out the door. The benefit of having centralized standards control is obvious.

4.2 Rapid Prototyping

Conventional wisdom, stemming from the software disasters of the 60's and early 70's, has firmly entrenched the hedging policies of preparing exhaustive feasibility studies, formal requirements definitions, structured walk-throughs, and the like. Often, the time and costs to plan a system are greater than the costs of building it. In turn, the specifications are usually out of date by the time they are finally approved, and the end-users still don't really know what they are getting, or if what they get is what they need. Another danger is that it is so easy to specify features which turn out to be much more difficult to implement than they are worth to the user. In short, the institutionalized policies of large DP groups are no small contributor to the enormous applications backlog.

Conventional wisdom can now be made wiser [5,6,11,12,15]. CAP tools can write formal specifications which are understood by both people and computers, then convert the spec's to equivalent programs. We can now adopt the attitude of "what you see is what you get", and even let small prototypes constitute part of the design spec.

End-users can "kick its tires" and iteratively guide the specifications. The implementation team can provide specific, detailed arguments as to why certain features should or should not be in the system, and can more accurately cost-estimate the system's implementation based on deviations from the organization's current frame inventory.

4.3 Productivity and Quality

Using a tool such as CAPinput typically requires

that the user spend a few minutes at the under--specified level. Without further customization, an executable program is available shortly thereafter. The following is the summary from a detailed case study which analyzes the actual usage of CAP.

4.3.1 CASE STUDY: The Manufacture of the CANADIANA Requisition System

CANADIANA OUTDOOR PRODUCTS INC. is a subsidiary of NOMA INDUSTRIES LTD. In March 1983, Canadiana employed Netron Inc. to create a computerized Requisition system to replace Canadiana's manual Requisition system.

The system was created using CAP and is run on a WANG VS computer using interactive terminals. The system allows requisitions to be created, maintained, displayed, searched, authorized, ordered, recorded and reported upon.

The Requisition system was built by a student analyst during his first work term leave from the University of Waterloo. After the first week, enough of the system had been prototyped that Canadiana users recognized serious design problems. The system was redesigned and put into production by the end of the third week.

Sixteen programs were created using CAP tools, to create and control the interaction of the 22 screens and 3 reports through which the Requisition system is operated. CAP tools enabled the author to create the Requisition system by writing less than 10% of the total COBOL lines needed.

One method of judging COBOL program production with and without CAP tools is to compare the total number of lines of submitted source code in the entire Requisition system with the number of hand-written lines. Purely comment lines were discarded.

The results show more than a 10:1 productivity gain by this measure. Of 34,000 lines of submitted code contained in the 16 programs of the Requisition system, only 3,000 lines were written by hand.

The following table shows, for each of the 16 programs forming the Requisition system, the number of lines (i) hand written in the SPC frame, (ii) in the generated frames, (iii) in standard frames, and (iv) in the total submitted to the COBOL compiler.

4.3.2 Quality

Of course, the issue here is not merely to show a capability of producing in excess of 2000 lines of production COBOL per man day (including design time). Further analysis of the manufactured programs will show that they are more consistent with respect to user-interface and structured program style, more reliable, more functionally complete, much more easily maintained, and no less efficient than conventional, hand written programs. The reason is that the standard frames and frame generators are highly seasoned components in the course of whose evolution many improvements and optimizations have been made. The cumulative effects are capital assets (no pun intended) which yield a return on investment in every incorporating program. Programs handwritten from scratch have no chance to acquire the quality and thoroughness that is the hallmark of a good frame [15].

Number of Code Lines

Program Name	Main CAPTool	Total Source	SPC Frame	Generated Frames	Standard Frames
PREQ1	CAPinput	2979	56	1731	1192
PREQ2	CAPinput	2130	71	1264	795
PREQ3	CAPinput	2318	78	1013	1227
PREQ4	CAPinput	1721	62	869	790
PREQ5	CAPinput	3440	421	1904	1115
PREQ6	CAPinput	2776	157	1766	853
PREQ7	CAPinput	1510	40	673	797
PREQ8	CAPinput	3018	206	1806	1006
PREQ9	CAPinput	3238	281	1910	1047
PREQA	CAPinput	3659	436	2223	1000
PREQI	CAPinput	3399	436	1916	1047
PREQF	Frame Lib.	274	187	0	87
PREQG	Frame Lib.	223	136	0	87
PREQR	CAPreport	954	140	198	616
PREQS	CAPreport	1086	226	216	644
PREQT	CAPreport	1152	179	290	683

Figure 2

4.4 Life Cycle Support

Maintenance is one of CAP's strongest features. By storing all source code customizations in one spot, factored away from both standard and generated frames, typical program maintenance is collapsed from 50 - 60 pages of source listing to two or three pages. By having the code generators emit frame code which can be automatically customized, the declarative specifications also support the life cycle maintenance of the programs in a very convenient manner.

4.4.1 Frame Maintenance

As with all software, frames change through time. Standard frames tend to be relatively stable since they rapidly become seasoned through frequent reuse. But additionally, because they are functionals, they are able to absorb arbitrary amounts of change (including complete rewrites) without risking any previously written program. It is easy to arrange that the range (function space) of a new version of a functional be a superset of the previous version's range. Simply provide a version control parameter governing a SELECT clause. This allows the improved functional to still recreate all old functional versions. An old program's SPC, unaware of subsequent changes, references the frame hierarchy with its old version symbol (if any!), and gets exactly the same code it has always gotten, even though new programs may get something quite different (the Template always contains the latest version symbol).

This does not mean that frames and libraries become more cluttered than in conventional shops. Conventionally, complete copies are kept of all versions (using distinct names), even though only small changes might have been made. Frames keep an automatic audit trail of the version differences, with

only occasional rewrites done to eliminate clutter. The obsolete (but still active) rewritten versions are placed in a separate library, again to eliminate clutter. Internal version references automate the retrieval of the correct version. Thus a single external name is common to all versions and less space overall is actually required.

4.5 Conclusion

CAP grew out of a need in the world of business data processing to solve the reusable code problem. The resulting design principles are quite general and are applicable in any application area which can be factored into recurring problem classes. To build the tools and work-in-process inventory of a CAP software factory for a new problem domain, the following is required.

First, a CAP preprocessor is built for the currently used, general purpose programming language. Standard frames can then be written to form the reusable components of new programs. Often, well-structured, existing programs can serve as models for creating these frames. If the degrees of freedom for any function spaces can be formalized, then special purpose declarative languages can be defined at the optimal level. By building an editor-generator pair for each language which emits frames, further automation can take place. Finally, if several function spaces (currently spanned by a combination of standard frames and code generators) often need to be combined to produce needed programs, then designer front-ends can be built which operate at the under-specified level to provide rapid prototyping of complete programs.

Software has been very successful in automating conventional manufacturing. It is now possible for our own industry to gain the same benefits.

References

1. BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM, 21, 8 (August 1978), 196-206.
2. BALZER, R. An alternative approach to software automation. In Research Directions in Software Technology, P. Wegner (Ed.), MIT Press, Cambridge, Mass., 1979, pp. 851-856.
3. BASSETT, P.B. and GIBLON, J. Computer Automated Programming (Part I). In proceedings of IEEE conference on Software Tools and Techniques (Soft Fair), Washington D.C., July 1983.
4. BASSETT, P.B. and RANKINE, S. The Maintenance Challenge. Computerworld In Depth, May 16, 1983.
5. BIANCHI, M.H., MASHEY, J.R. Rapid Prototyping on UNIX™. Presented at the Software Engineering Symposium: Rapid Prototyping, Columbia Maryland, April 19-21 1982.
6. BLATTNER, M., FROBOSE, R. Prototyping and the Life Cycle of Software. Presented at the Software Engineering Symposium: Rapid Prototyping, Columbia Maryland, April 19-21 1982.
7. CHEATHAM, T.E. The Harvard PDS Project: an Overview. Presented at the Software Engineering Symposium: Rapid Prototyping, Columbia Maryland, April 19-21 1982.
8. EVANS, M. Software Engineering for the Cobol Environment. Communications of the ACM, 25, 12 (December 1982), 874-882.
9. GOGUEN, J.A., THATCHER, J.W., and WAGNER, E.G. An Initial Algebra Approach to the specification, correctness and implementation of abstract data types. In Current Trends In Programming Methodology, vol 4, R. Yeh (Ed.). Prentice-Hall, 1979 pp. 80-149.
10. HAMMER, M., RUTH, G. Automating the Software Development Process. In Research Directions in Software Technology, P. Wegner (Ed.), MIT Press, Cambridge, Mass., 1979, pp. 767-790.
11. HOUGHTON, R.C.jr. Rapid Prototyping Tools: What can we Learn From the MIS World ? Presented at the Software Engineering Symposium: Rapid Prototyping, Columbia Maryland, April 19-21 1982.
12. MASON, R.E.A., CAREY, T.T. Prototyping Interactive Information Systems. In CACM Vol. 26 No. 5 p 347
13. MINSKY, M. A Framework for Representing Knowledge. In The Psychology of Computer Vision, P. Winston (Ed.), McGraw-Hill Inc., U.S.A., 1975, pp. 211-277.
14. RICH J. Inspection Methods In Programming, Ph.D. Thesis M.I.T. technical report AI-TR-604, June 1981
15. TAYLOR, T., STANDISH, T.A. Initial Thoughts on Rapid Prototyping Techniques. Presented at the Software Engineering Symposium: Rapid Prototyping, Columbia Maryland, April 19-21 1982.
16. WASSERMAN, A.I., GUTZ, S. The Future of Programming. Communications of the ACM, 25, 3 (March 1982), 196-206.
17. WULF, W.A. Some Thoughts on the Next Generation of Programming Languages. In Perspectives on Computer Science, Academic Press, New York, New York, 1977, pp. 217-234.

DECISION SUPPORT SYSTEMS
AND
DEC-MICROS

Dr. Kuriakose Athappilly, Associate Professor
BIS Department
Western Michigan University
Kalamazoo, Michigan

ABSTRACT

This paper deals with Decision Support Systems (DSS) and their impact in the dec-micros. Since DSS is not a very well known concept today, the paper attempts to explain briefly what DSS is and then illustrates the existence of several hardware/software configurations that are entitled as Decision Support Systems in the Dec-family.

INTRODUCTION

Decision Support Systems, better known as DSS have become the hallmark of the modern executive. It has been growing in popularity since the 1970s. It has evolved from EDP, MIS, and MS, but it differs from all of them as its main function is to aid managers directly and quickly in their decision making process. There are many different views on how to approach the question, "What is a DSS?". It is generally understood as "an interactive system that provides the user with easy access to decision models and data in order to support semi-structured and unstructured decision making tasks." (1) The intent of DSS is not to automate the decision making process, but to provide information and add insight as support for managers' decision making process. (See Chart 1) With a DSS the Manager can combine its benefits with his analytical skills and judgment to reach the optimal solution. Parallel to the growth and development of DSS as a concept, we witness an evolution in the family of Dec-systems toward the development of many hardware and software configurations for several general and specific Decision Support Systems.

General Characteristics of DSS

There are many different and unique decisions a manager is faced with and for this reason a DSS must contain some unique qualities to make it an effective tool. They are flexibility, inter-activeness, discovery orientation, and easy-to-learn. Flexibility allows a manager to create different models, manipulate data in a variety of ways, and to match information to the problem at hand. Interactiveness is important as it facilitates the manager's communication with the system, for quick and clear results. Discovery orientation allows managers to probe trends, isolate problems and ask new questions. Easy - to -learn feature helps managers use the system without learning its technical aspects. The focus of all these features is on the user which enable the DSS to follow his or her thought process.

". . . Decision support systems rely on the decision makers' insights and judgment at all stages of problem solving--from problem formulation to choosing the relevant data to work with, to picking the approach to be used in generating solutions, and on to evaluating the solutions presented to the decision maker." (2)

Chart 1: A comparative view of the three systems; MIS, OR/MS, and DSS in terms of their impact, payoff, and relevance.

	IMPACT	PAYOFF	RELEVANCE
MIS	STRUCTURED TASKS	EFFICIENCY	INDIRECT REPORT
OR/MS	STRUCTURED PROBLEMS	SOLUTIONS	RECOMMENDATIONS & SOLUTIONS
DSS	DECISIONS	EFFECTIVENESS	TOOL -OWN CONTROL -DIRECT -NO AUTOMATION

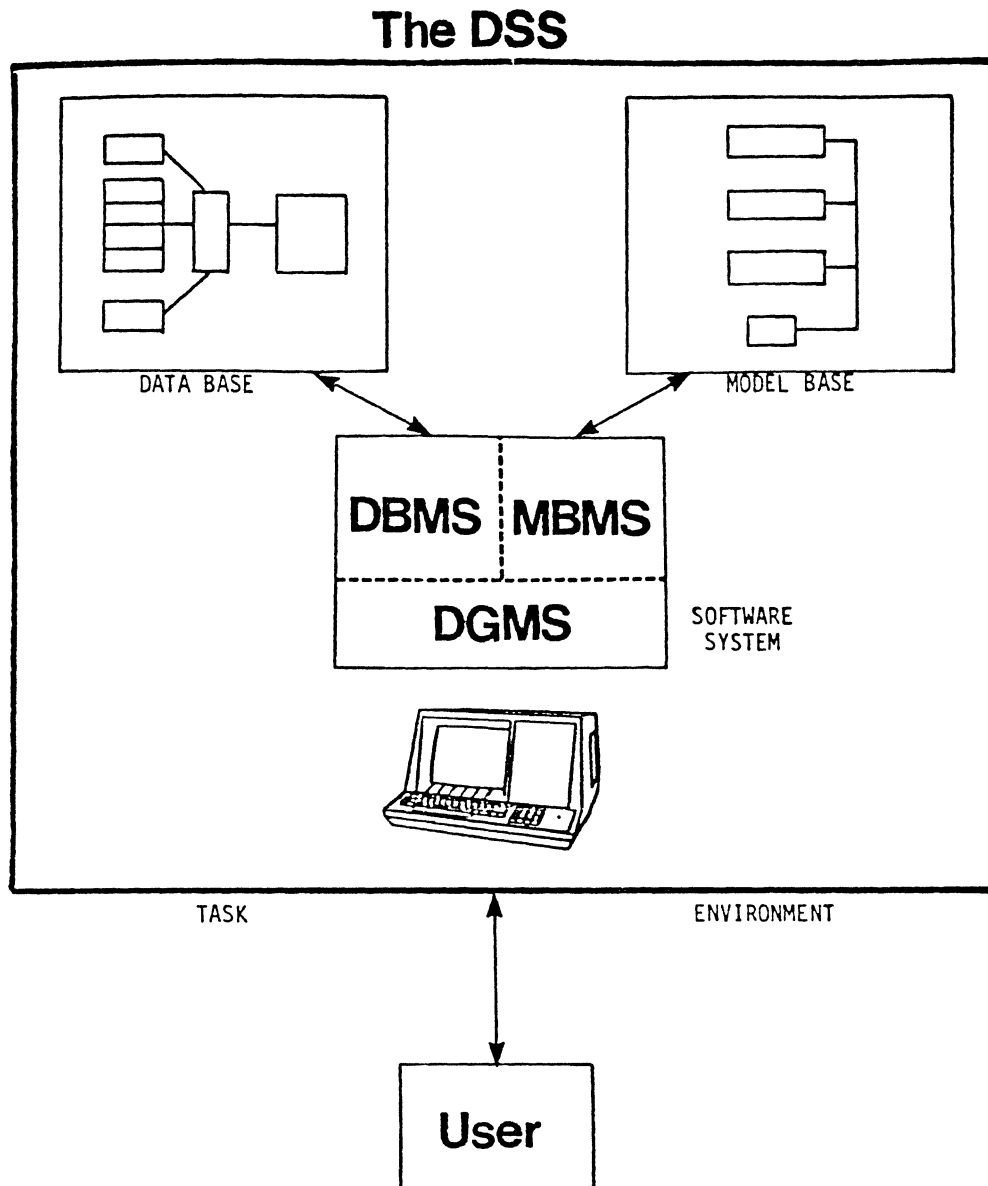
Technical Characteristics of DSS

The general characteristics reveal what DSSs are supposed to do, but the question "What is a DSS?" remains to be answered for a complete understanding.

It is generally agreed that a software program to qualify as a DSS must have three general capabilities: Database management, model base management, and dialog generation as shown in Figure 1.

The database management within the DSS which basically consists of data insertion, extraction, and retention processes is essential for an efficient and effective DSS generating phase. This component provides for the memory requirements in decision support. Considerations for database management include security procedures, a data dictionary, ease of data entry, multiple access availability, and audit trail capability. Because the accuracy, integrity and reliability of the information provided by the DSS depends on the accuracy, integrity and reliability of the data

Figure 1: COMPONENTS OF THE DECISION SUPPORT SYSTEM



from Sprague & Carlson, Building Effective Decision Support Systems, Prentice-Hall, 1987.

used in the database, it's important that the data-base management system be considered an important function.

The model base management system makes DSS a unique software system. Modeling is the primary function of DSS because it can be used to create ad-hoc models and scenarios that represent real world situations. Those scenarios help the manager explore alternatives and examine the consequences of such alternate decisions on the computer before implementation in the real world. This capability of exploring and testing many alternatives and getting answers to several "what if" questions is the unique strength of DSS. Naturally, since this modeling capability reveals the strengths and weaknesses of the design structure the user manager can create appropriate designs by repeated what if test questions to suit the real world decision environment in a natural and logical format.

The model base management system includes also several statistical and mathematical manipulations which offer DSS a good deal of analytical ability and flexibility. Forecasting, ranking, sorting, simultaneous equation solving, totaling decision analysis, optimization, and averaging all aid in model building. Most of these manipulations are offered in many of the advanced spreadsheets. With spreadsheets it is easy to introduce new variables and instantly calculate their effect through a chain of events and a multitude of products. Consequently user managers can create one or more models, expand or reduce them, and

modify them to help explore more and more alternatives.

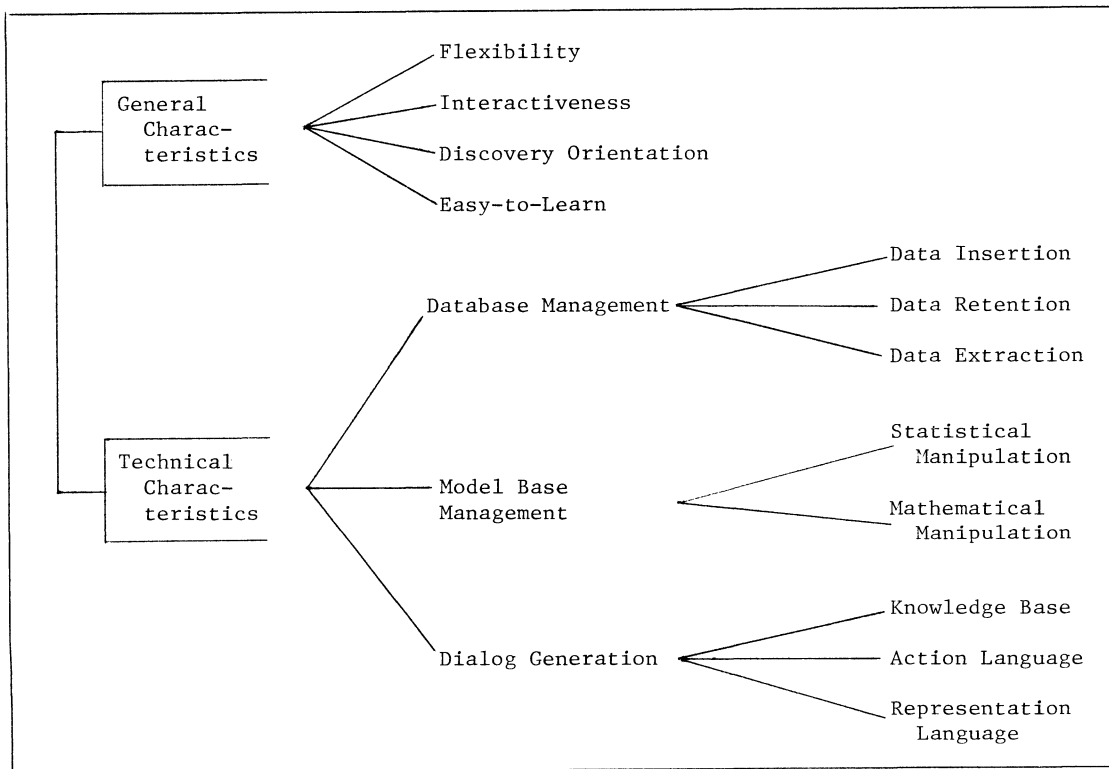
The dialog component of a DSS is the means through which the users communicate with DSS. Much of the power, flexibility and ease-of-use characteristics of a DSS depends on the efficiency of the dialog component. The communication with DSS depends on three factors. They are: i) the knowledge base, ii) the action language, and iii) the representation language. The knowledge base consists of what the user manager needs to know in order to use the system efficiently. The action language indicates what the user can do communicating with the system. The representation language indicates what the user manager sees. This third characteristic, is perhaps, the most important of the three cited above, especially because of the developments in graphics display.

Graphic display enhances DSS by displaying graphs, reports and charts or other forms of communicating commands and results to the user. Spreadsheets that can demonstrate DSS qualities must have the ability to generate personalized reports with standard formats.

Each of the functions described are not separate features when contained in DSS, but are combined to give flexibility to do many things at once (see Chart 2 for summary of the DSS characteristics). They are all integrated to give the software a dynamic and paradigmatic quality. Ideally a decision support system is "a comprehensive computer software system encompassing everything

Chart 2

DSS CHARACTERISTICS



a manager needs: Highly flexible; adaptable database management; powerful modeling capabilities; a wide range of easily accessible statistical and mathematical techniques; presentation - quality graphics; and report writing." (3)

Decision Aid and Decision Modeling

There are two types of decision support programs, decision-aid and decision-modeling. Decision-aid programs allow users to evaluate alternate options by assigning weighted values to each factor in a decision. Then the computer calculates the highest score and reports the recommended course of action. Since the assigning of values is subjective, this type of program can bring an outcome that matches a decision already made.

The second type of program, decision-modeling programs, are usually considered more useful because of their ability to use many variables and answer "what if" questions. This type of software aids the user in choosing the most effective strategy for carrying out a decision. Both decision-aid and modeling programs give managers many features to choose from.

DEC-Micros Towards DSS

With the introduction of Digital Office Workstations, Micro VAX-11, and Micro PDP-11 series in

the hardware area and numerous specific and generic DSS packages in the software area, the Decision Support Systems have become a conspicuous feature in the Dec-family. Figure 2 shows the office work stations with VAXes, indicating the different machines and their capabilities. Figure 3 displays the range of solutions that a manager can work out with VAXes in his/her decision making process. Figure 4 shows the integrated system, a typical DSS environment, that can be created using Dec-micros.

There are several DSS application software available on the market that are compatible with Dec-systems. Chart 3 shows the available software in their rank order in different functional areas. Chart 4 lists several of the existing DSS software that are compatible with the Dec-micros.

Figure 2: Office Workstations with VAXes

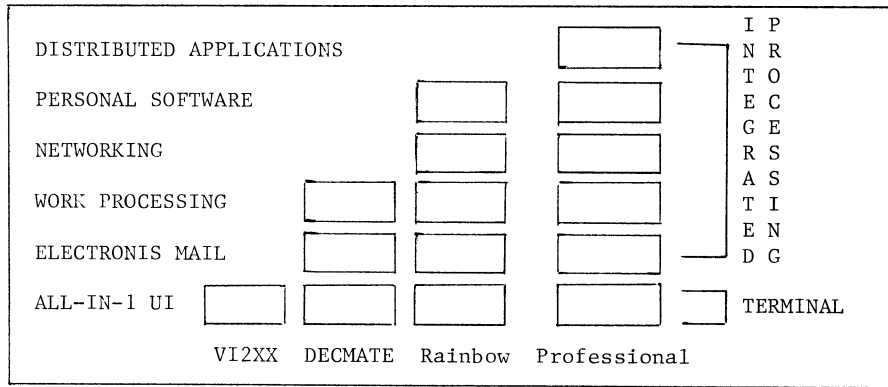


Figure 3: Range of Solutions with VAXes

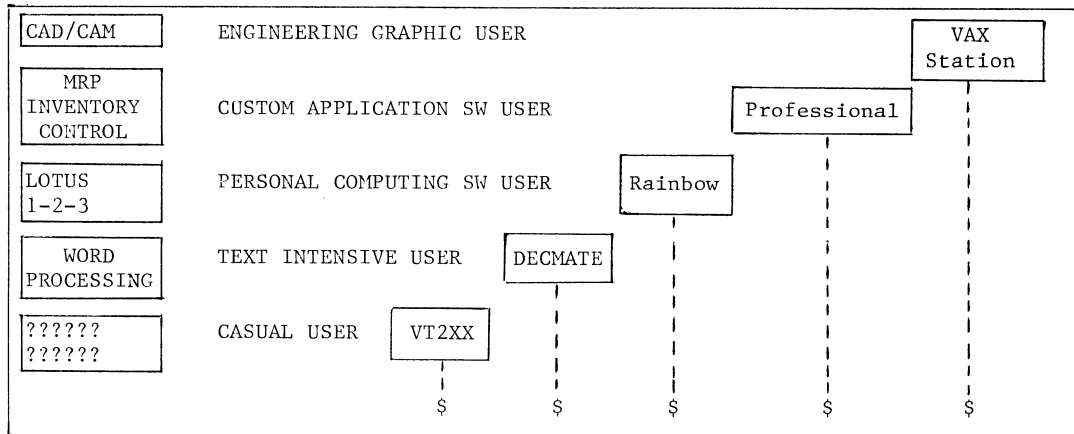


Figure 4: Integrated Office System

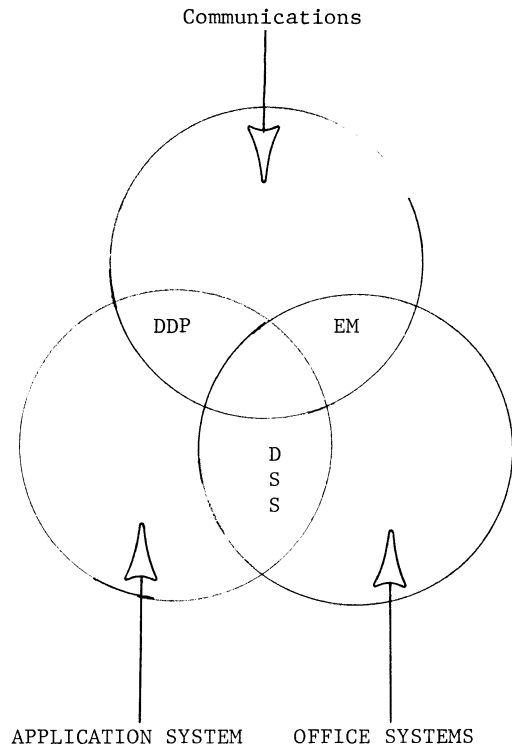


Chart 3: DSS application software according to the rank of order of availability.

(Items ranked by order from most availability to least availability)

MAXIMUM	
↑	
	1. Financial Management
	2. Management
	3. Information Management
	4. Accounting
	5. Office Systems
	6. Sales and Marketing
	7. Education
	8. Manufacturing
	9. Real Estate
	10. Distribution
	11. Construction
	12. Engineering
	13. Law
	14. Personal Computing
↓	
MINIMUM	

Chart 4: DSS Application Software in different functional areas.

- Accounting
 - MAPS/AP----- A.P.
 - MAPS/GL----- G.L.
 - MAPS/MODEL-- FP/FA
- Construction
 - SUPERVISOR-- CPM/PERT
- Distribution
 - DACMS----- Inventory
 - CDIS----- Forecasting
- Education
 - DSMTUTOR---- FP, Simulator
 - FINAID----- FP
- Engineering
 - CADAT----- CAD/CAM
- Financial Management
 - ASSET-LIABILITY system
 - BANKMASTER
 - BUSINESS MODELER
 - CALC-II/CALC-II PLUS
 - DATAMODEL
 - DATA CALC
 - DECCALC
 - DIGICALC
 - FLOWCALC
 - INVESTMASTER
 - MONEYMAN
 - MODELER
 - OPTIQUBE
 - PLANPLUS
 - UNICALC 3-D
 - XSP
- Information Management
 - ACCENT R
 - GRS
 - MAPS/DB
 - READER
 - XPLAN PLUS
- Law
 - INMAGIC
- Management
 - AIM BENCHMARK -- Suites I and II
 - DYNAMO
 - LP
 - MISTER
 - MOSAIC
 - RISKAN1/2
 - SIBYL/RUNNER
- Manufacturing
 - COMETS
 - HS/LP
- Office Systems
 - MATRIX
 - PLESSEY-CALC
 - THEMIS
 - VC
- Personal Computing
 - SUPERCOMP-TWENTY

DATA ACQUISITION, ANALYSIS, RESEARCH, AND CONTROL SIG

LISREL

An application, An explanation

Interfaced with SPSSX on a VAX/VMS 11/780

Leanne Whiteside

University of Arkansas at Little Rock

Little Rock, Arkansas

ABSTRACT

LISREL VI is a computer program for estimating the unknown coefficients in set of **L**inear **S**tructural **R**ELationships. It is accessed on the VAX/VMS computer via the statistical package SPSSX (Statistical Package for the Social Sciences). The purpose of this paper is to explain the application of LISREL to actual data and to provide a guide to new users of LISREL. First, a discussion of the LISREL program and its model will be presented, followed by two examples. The first example, taken from the LISREL VI User's Guide, contains both latent (unobservable) and manifest (observable) variables. The output is examined in detail. The second example is the analysis of data provided by a study in progress at the Center of Child Development and Education at the University of Arkansas at Little Rock. This second example contains only observed variables. Throughout this paper the LISREL VI User's Guide by Karl G. Joreskog and Dag Sorbom is heavily used as a source of information.

1 Terminology and Conventions

To discuss LISREL, it is necessary to review related terminology and path diagram conventions. LISREL can provide estimates for equations involving both **manifest** (observed) and **latent** (unobserved) variables. Any variable whose variability is assumed to be determined by other causes outside the causal model is called **exogenous**. When a variable's variation is explained by exogenous variables or other variables in the system, the variable is called **endogenous**.

The path coefficient indicates the direct effect of a variable upon another variable. The coefficient usually has a double subscript indicating the cause and effect variables.

For example, if η_3 has a direct effect upon η_4 , the path

coefficient would be named $\beta_{4,3}$. All observable variables are represented by squares; all unobservable variables are represented by circles. The correlation between exogenous variables is indicated by curved lines with arrowheads at both ends. Straight lines with one arrow are drawn from the variables taken as causes (exogenous, independent) to the variables taken as effects (endogenous, dependent).

2 Introduction

In 1966 Karl G. Joreskog presented an efficient numerical method for the maximization of functions of many variables. This led to the development of the LISREL model. It has since played such a major role in the application and analysis of structural equation models that such models are often referred to as "LISREL" models. Models of this kind are also

know as simultaneous equation systems, linear causal analysis, path analysis and dependence analysis.

Joreskog's LISREL program is based on a general model that may include:

- 1) a measurement model specifying the relations between the observed variables and the unobserved (latent) variables including measurement errors
- and
- 2) a linear structural equation model specifying causal relationships among the unobserved variables with possible reciprocal causation and random disturbance terms.

LISREL can provide estimates for models that include one or both of these sets of equations.

LISREL will not only estimate the unknown coefficients in the structural equations but will allow for errors in equations (residuals, disturbances) and errors in variables (measurement, observation). The variance-covariance matrices for these errors will also be estimated. The LISREL model can handle correlated errors and residuals. LISREL can estimate unknown parameters by any of five different methods for fully identified models. (Identification of models is discussed below.) The five methods include two-stage least squares, unweighted least squares and generalized least squares. The method of maximum likelihood estimation is the default method in LISREL. This method is known as a "full information" approach since it uses all the information in the data and parameters are estimated simultaneously. All methods give consistent estimates for fully identified models.

3 The General LISREL Model

In the general model, the relationship among a set of observed variables and a set of unobserved variables is examined. The observed variables are taken to be measures of the unobserved variables. The relationships among the observed variables are contained in the covariances among the observed variables. Let Σ be the matrix of the covariance for the population. The sample covariance matrix, S , is input by the researcher. LISREL uses the sample covariance (or correlations) matrix S and constraints on the parameters to compute an estimate of Σ . In the process of this estimation the unknown parameters are estimated. Actually LISREL is not restricted to the covariance or correlation matrix as input, but the examples in this paper uses only the

covariance and correlation matrices.

The measurement model is a set of equations that link the observed variables to the unobserved variables. The relationship of the unobserved variables to each other are contained in the structural model. The problem is then one of estimating the unknown parameters and determining the goodness of fit of the parameters and model.

To illustrate the general LISREL model, consider the following path diagram in figure 1. Observed independent and dependent variables, x_1 and y_1 , are indicated with boxes. Latent (unobserved) independent and dependent variables ξ_1 and η_1 are enclosed in circles. The observed variables x_1 and x_2 are measures of the unobserved variable ξ_1 . λ_5 and λ_6 are the regression coefficients of x_1 and x_2 on ξ_1 . The error of measurement of x_1 and x_2 is indicated by δ_1 and δ_2 . Similarly, y_1 and y_2 are measures of η_1 . ϵ_1 and ϵ_2 are errors of measurement of y_1 and y_2 and λ_1 and λ_2 are regression coefficients of y_1 and y_2 on η_1 .

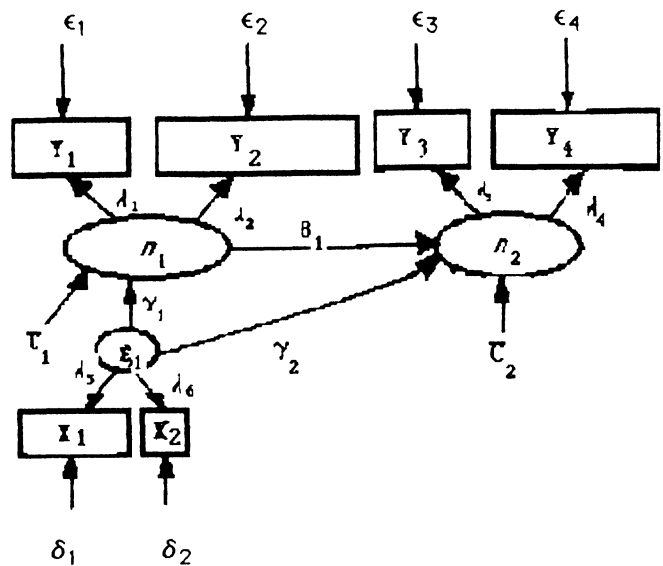


Figure 1

The relationships described above can be summarized in the following system of equations called the measurement model.

$$y_1 = \lambda_1 \eta_1 + \epsilon_1 \quad x_1 = \lambda_5 \xi_1 + \delta_1$$

$$y_2 = \lambda_2 \eta_1 + \epsilon_2 \quad x_2 = \lambda_6 \xi_1 + \delta_2$$

$$y_3 = \lambda_3 \eta_2 + \epsilon_3$$

$$y_4 = \lambda_4 \eta_2 + \epsilon_4$$

Or in matrix form:

$$Y = \lambda_y \eta + \epsilon \quad X = \lambda_x \xi + \delta$$

Where the elements of λ_x and λ_y are regression coefficients. The vectors ϵ and δ are the errors of measurement of the observable variables. Two matrices related to ϵ and δ computed by LISREL are called θ_ϵ and θ_δ . θ_ϵ contains on its diagonal the variances of the ϵ_i errors of measurement of y_i . The off diagonal elements are the covariances of the ϵ_i variables. Similarly, θ_δ is the variance / covariance matrix for the δ_i errors of measurement of x_i .

The structural model describes the relationships between the unobservable (latent) variables. In this example the independent latent variable ξ has a direct causal effect on the latent variables η_1 and η_2 . This model also indicates that the latent variable η_1 has a direct causal effect on η_2 . These effects are indicated in the coefficients β_1 , γ_1 and γ_2 . ζ_1 and ζ_2 are the errors (random disturbances) of the structural equations. The structural equations are:

$$\eta_1 = \gamma_1 \xi_1 + \zeta_1$$

$$\eta_2 = \beta_1 \eta_1 + \gamma_2 \xi_1 + \zeta_2$$

Or in matrix form:

$$\eta = \beta \eta + \gamma \xi + \zeta$$

where β and γ are coefficient matrices and ζ is a vector of errors in equations (random disturbance terms). Notice that the matrix β contains information about the effects of η

variables on other η variables. The diagonal of β is always zero since η variables are not allowed to directly "cause" themselves. γ contains information about the effects of latent exogenous variables (ξ) on latent endogenous variables (η). By definition exogenous variables are those that cause other variables and whose variability is assumed to be determined by other causes outside the causal model. Endogenous variables variation is explained by exogenous variables or other variables in the system.

Two additional matrices estimated by LISREL are ϕ (PHI) and ψ (PSI). The first ϕ (PHI) is the variance/covariance matrix of the latent exogenous variables (ξ). The second (ψ) is the variance/covariance matrix of the errors in the equations (ζ variables).

3.1 Assumptions

Joreskog makes five assumptions in the LISREL model. From the LISREL User's Guide they are:

- 1) ζ is uncorrelated with ξ
- 2) ϵ is uncorrelated with η
- 3) δ is uncorrelated with ξ
- 4) ζ , ϵ , δ are mutually uncorrelated
- 5) β has zeros on the diagonal and I-BETA is non-singular.

The first four assumptions involve the correlation of error terms with latent variables or each other. To see why these assumptions might be reasonable look at the path diagram in figure 1 or the equations above. The first assumption states that the disturbance terms in the structural equations are uncorrelated with the exogenous variables in the structural equations model. The second states that the errors of measurement of the observed dependent variables are uncorrelated with the latent independent variables. The third states that the errors of measurement of the observed independent variables are uncorrelated with the latent dependent variables. The fourth states that the errors of measurement and the disturbances in the structural equations are uncorrelated. Notice that the error terms can be correlated among themselves. For example ϵ_1 could be correlated with ϵ_2 . The last assumption simply states that none of the equations in the model are redundant.

In addition, all variables are assumed to be measured as deviations from their means. This implies that the expectation of the variable is zero and the expectation of a product is a variance or a covariance. This is not a limitation to the model since it involves only a change in origin. This assumption will prove useful in the equations used to prove the model is identified. If the method of Maximum Likelihood (ML) is used the additional assumption is that the input correlation matrix S is positive definite. Although the manual is not clear, the program does seem to check the input matrix to verify that it is positive definite. If it is not the analysis is canceled by the program and a message is issued.

The elements of the eight matrices contained in the full LISREL model can be controlled by the researcher in three ways. They can be fixed, constrained or freed. Free parameters are unknown and are estimated by LISREL. Constrained elements are unknown but set equal to one or more other parameters in the model. Fixed parameters have been assigned a value (zero or otherwise) by the researcher and are not estimated by LISREL. Each of these eight matrices have a default mode, either fixed or free. That is, if the researcher does not indicate the mode of a particular matrix it is given a default of fixed or free. Each matrix has a default form, either a full non-symmetric matrix, zero matrix, symmetric matrix or diagonal matrix. Table 2 in the first example gives the defaults for each matrix.

3.2 Estimation

It can be shown that the elements of the population correlation matrix of the observed variables Σ are functions of these eight matrices, β , γ , λ_x , λ_y , θ_ϵ , θ_δ , ψ and ϕ . For some models the elements of these matrices will be fixed to zero or the identity matrix. Estimation of the free parameters of these matrices is accomplished by an iterative scheme (an application of the Davidon-Fletcher-Powell method) that produces an estimate of Σ . When the method of maximum likelihood is used the fitting function below is minimized by the iterative procedure until the solution converges, that is, until the estimated Σ is "close" to the true Σ . Since Σ is not known the sample correlation matrix S is used. The scheme must have initial estimates (provided by LISREL or the researcher) and must satisfy the constraints that have been imposed on the model. It is possible for the iterative scheme to converge to a local minimum. This would result in

incorrect estimates for parameters. However, this problem is thought to be rare. (Long, 1983).

The fitting function for the ML estimator is defined by:

$$F(S, \Sigma^*) = \text{tr}(S \Sigma^{*-1}) + [\text{LOG} |\Sigma^*| - \text{LOG}|S|] - (p+q)$$

where Σ^* is the estimated Σ , Σ^{*-1} is the inverse of Σ^* , and $\text{LOG} |\Sigma^*|$ is the LOG of the determinant of Σ^* . As Long (1983) points out it is not hard to see intuitively how this function reflects the distance between S and Σ^* . If S and Σ^* have elements similar in value, then their inverses also. As S and Σ^* become close ($S \Sigma^{*-1}$) become closer to a $(p+q)$ identity matrix, since the dimensions of both S and Σ^{*-1} is $(p+q)$ by $(p+q)$. The trace of a $(p+q)$ by $(p+q)$ identity matrix is equal to $p+q$, so the first term of the fitting function goes to $p+q$ as S and Σ^* become close. Also as Σ^* and S become closer their determinants (and the logs of their determinants) become closer and the second term of F approaches zero. The last term of $p+q$ is to cancel the first $p+q$ in the fitting function so that F approaches zero as S approaches Σ^* .

3.3 Identification

During the calculations, LISREL assumes the model is identified. A model is identified if the choice of the model and the constraints (or lack of constraints) on the matrix elements result in one and only one estimate of Σ . Unfortunately there is not a general and practical way to determine if a general model is identified. (Some special cases of the LISREL model have necessary and sufficient conditions defined.) LISREL will attempt to analyze any model regardless of the question of identification, but results can not be trusted unless it is known the model is identified. In some cases warnings will be issued by LISREL indicating parameters that may not be identified. Other LISREL results that indicate an unidentified model will be discussed in the following output. A necessary (but not sufficient) condition for identification is the following inequality.

$$t < (1/2)(p+q)(p+q+1)$$

where t is the total number of independent parameters to be estimated, p is the number of x_i variables and q the number of y_i variables.

An effective (although time-consuming) way to determine that a model is identified is to show that each parameter can be solved in terms of the population variances and covariances of the observed variables. In this way parameters are proven identified individually. This method is illustrated in the following examples. If all the parameters are identified the model as a whole is identified. If a parameter can be solved in more than one way the parameter is over-identified and the model as a whole is said to be over-identified. Note that individual parameters can be identified but the model as a whole is not identified unless ALL the parameters are identified. However these identified parameters can be estimated even though the estimated for the unidentified parameters can not be used. If a parameter can not be solved in terms of the variances and covariances of the observed variables then the parameter is unidentified and the model is unidentified.

In summary, identified and over-identified models can be completely estimated, while unidentified (also called under-identified) models can not.

If a model is not identified all is not lost! Additional limitations can be imposed on the coefficients linking measured variables or the researcher can make assumptions about the correlation among residual terms.

LISREL does not constrain estimates to be within any boundary range, this means that the program may give estimates of negative variances, correlations larger than one in absolute value, etc. Solutions of this kind can occur if the model is not identified. However it can also occur when the model does not fit the data or the sample size is too small. Another situation that can cause unacceptable estimates is the case of many missing values in the data. Researchers often handle missing data by pair-wise deletion in the calculation of the covariance (or correlation) matrix. This means that each correlation coefficient is calculated with a different population. In this case the correlation matrix should be used in the analysis by LISREL.

3.4 LISREL Data Input

As was mentioned before LISREL can take several forms of data as input. These input forms include raw data, the moment matrix, the covariance matrix or the correlation matrix. You may choose to analyze any one of the three matrices. LISREL can compute the matrix to be analyzed regardless of what has been read as input. Where necessary LISREL will use default values of zero for the means and one for the standard deviations. The decision of which matrix to use as input for LISREL and in the analysis by LISREL can be a difficult one for the researcher. The tables below should prove useful in this task. Parts of the following tables are taken from the notes of a workshop conducted by Karl Joreskog (see also Joreskog, 1984).

Table 1

INFORMATION RETAINED AND LOST

<u>Matrix</u>	<u>Retained</u>	<u>Lost</u>
Raw data	all	none
Moment Matrix	means, standard deviations, correlations	-
Covariance Matrix	standard deviations, correlations	means
Correlation Matrix	correlations	means, standard deviations

MATRIX TO BE USED IN ANALYSIS

LISREL Name	Details
CM Covariance	-this should be used in general -use if it is desired to retain the original units of measurements in the observed variables
KM Correlation	-use if the model is scale-free and the units of measurement of the observed variables are arbitrary or irrelevant -use when pairwise deletion of missing values has occurred
MM Moment	-use if the model contains intercept terms and/or mean values of the latent variables

4 Example 1, involving Latent and Manifest Variables

The following example will serve to illustrate the LISREL control cards necessary to perform an analysis of a model and to explain the meaning of the LISREL output. This example found in the LISREL manual (1984) was originally taken from Wheaton et al (1977). The data was collected over three points in time: 1966, 1967 and 1971. Two observed variables, Education and SEI (Socioeconomic Index) are used to measure the unobservable variable SES (Socioeconomic Status). The observable variables, the Anomia subscale and the Powerlessness subscale, are used in the model to measure the unobservable variable Alienation. Data were collected from 932 persons. This example uses data from 1967 and 1971 only. The path diagram for this model is the one discussed in the beginning of this paper (figure 1).

The observable exogenous variables in the model are:

x_1 = Education

x_2 = SEI (Socioeconomic Index)

They are indicators of the latent exogenous variable:

ξ_1 = SES (Socioeconomic Status)

The observable endogenous variables in the model are:

y_1 = Anomia 67 y_3 = Anomia 71

y_2 = Powerlessness 67 y_4 = Powerlessness 71

They are indicators of the latent endogenous variables:

η_1 = Alienation 67 η_2 = Alienation 71

Table 2

MATRICES IN LISREL				
Matrix	Lisrel	Default		
Symbol/Name	Name	Form/mode	Description	
λ_y	LAMBDA-Y	LY	FU/FI	Factor matrix of y_i on η_i
λ_x	LAMBDA-X	LX	FU/FI	Factor matrix of x_i on ξ_i
β	BETA	BE	ZE/FI	Coefficient matrix for dependent variables
γ	GAMMA	GA	FU/FR	Coefficient matrix for independent variables
ϕ	PHI	PH	SY/FR	Covariance matrix of ξ_i 's
ψ	PSI	PS	SY/FR	Covariance matrix of ζ_i errors
θ_ϵ	THETA-EPSILON	TE	DI/FR	Covariance matrix of ϵ_i errors
θ_δ	THETA-DELTA	TD	DI/FR	Covariance matrix of δ_i errors

All eight matrices in the LISREL model have parameters to be estimated in this example. These matrices are listed along with their LISREL name and a brief description. The default form and mode refer to the specification of the matrix that LISREL assumes if the researcher does not define them. The default forms are FU (full, non-symmetric matrix), ZE (a matrix of zeros), SY (symmetric matrix which is not diagonal), and DI (diagonal matrix i.e. off diagonal elements are zero). The default modes are FI (fixed i.e. not to be estimated) or FR (free i.e. to be estimated).

4.1 The Equations

The structural and measurement equations for the model follow. The measurement error terms ϵ_1 and ϵ_3 are constrained to be correlated in the model. Remember that ϵ_1 is the error of measurement of y_1 . It is reasonable that the error of measurement of a variable observed at two points in time would be correlated. Since η_1 and ξ_1 are unobserved they do not have a definite scale. The 1's assigned in each column of λ_x and λ_y assigns the scale of the unobserved variables to be the same as x and y respectively. In matrix form the equations are:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \lambda_2 & 0 \\ 0 & 1 \\ 0 & \lambda_4 \end{bmatrix} \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ \lambda_6 \end{bmatrix} \xi_1 + \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix}$$

$$\begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ \beta_1 & 0 \end{bmatrix} \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} + \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} \xi_1 + \begin{bmatrix} \zeta_1 \\ \zeta_2 \end{bmatrix}$$

or

$$\begin{aligned} \mathbf{Y} &= \lambda_y \boldsymbol{\eta} + \boldsymbol{\epsilon} \\ \mathbf{X} &= \lambda_x \boldsymbol{\xi} + \boldsymbol{\delta} \\ \boldsymbol{\eta} &= \boldsymbol{\beta} \boldsymbol{\eta} + \boldsymbol{\gamma} \boldsymbol{\xi} + \boldsymbol{\zeta} \end{aligned}$$

4.2 Identification

The question of identification must be answered before the LISREL model can safely be applied. The first test is the necessary (although not sufficient) condition of $t < 5(p+q)$ ($p+q+1$) when t is the total parameters to be estimated, p is the number of observed x variables and q is the number of observed y variables. The count of t includes $\lambda_2, \lambda_4, \lambda_6, \beta, \gamma_1$ and γ_2 , as well as the freed elements of $\Phi, \Psi, \Theta_\epsilon$ and Θ_δ . Which elements of these matrices are free to be estimated are determined by the researcher in the specification of the model. In this example Φ , the matrix of

the variance of the latent variable ξ_1 , is free to be estimated. Ψ , the matrix of variance / covariance of the ζ_1 error terms, has two elements free. The ζ_1 error terms are assumed to be uncorrelated so the matrix contains the variance of ζ_1 and ζ_2 on the diagonal and zeroes elsewhere. Θ_δ has two elements to be estimated, these are the variance of the δ error terms. The correlation of ϵ_1 and ϵ_3 will be estimated along with the variances of each ϵ_j . Therefore, Θ_ϵ has 4 variance elements on the diagonal and one off diagonal covariance element to be estimated. Again, the parameters that are free to be estimated are determined in the model by the researcher but are summarized by LISREL in the output labeled PARAMETER SPECIFICATIONS (see following output).

In this example there are a total of 16 independent parameters to be estimated (i.e. $t=16$). Then with $p=2$ and $q=4$ the necessary condition is satisfied. That is, $t = 16 < (1/2)(2+4)(2+4+1) = 21$. This does not mean that the model is identified! This condition is necessary but not sufficient. As stated earlier there is not an easy way to determine if a model is identified. There has been work done to show sufficient conditions to verify that certain types of models are identified. Some of these types and conditions are outlined by Long (1983, Covariance Structure Models page 34). In general, to show that a model is identified it must be possible to solve for all parameters to be estimated in terms of the variance and covariance of the observed variables. To do this, first working with the measurement equations solve for all parameters and variance / covariance of latent variables in terms of the variance/covariance of the observed variables. With these parameters identified, work with the structural equations to solve for the elements of β, γ and Ψ in terms of the variance/covariance of the latent variables.

The following equations come from the measurement equations by multiplying an equation by itself (or by another equation) and taking expectations. Notice the terms that are zero by assumption. These terms are zero because the researcher assumed no correlation or no correlation was assumed by the model in general. For example, since it is assumed that the δ error terms, the ζ error terms and the ϵ error terms are uncorrelated with each other the expectation of the product of any two of these is zero. That is, $E[\delta \zeta^T] = E[\delta \epsilon^T] = E[\zeta \delta^T] = 0$.

Table 3

Starting with the first measurement equation, multiply the equation by itself to obtain $y_1 y_1 = \eta_1 \eta_1 + 2 \eta_1 \epsilon_1 + \epsilon_1 \epsilon_1$. Take expectations and use the assumptions above to see that the next two equations are equivalent.

$$E[y_1 y_1] = E[\eta_1 \eta_1] + E[2 \eta_1 \epsilon_1] + E[\epsilon_1 \epsilon_1]$$

$$VAR(y_1) = VAR(\eta_1) + 0 + VAR(\epsilon_1)$$

The idea is to solve all parameters to be estimated in terms of the variance and covariance of the observed variables. We now have one equation and two unknowns. Continuing with the remaining measurement equations we have the following set of new equations. The equations are numbered on the left. The first six are obtained by multiplying an equation by itself. The last 15 come from multiplying one equation by another. The terms that are zero by assumptions in the model (either in general or set by the input model) are omitted.

- 1) $VAR(y_1) = VAR(\eta_1) + VAR(\epsilon_1)$
- 2) $VAR(y_2) = \lambda_2^2 VAR(\eta_1) + VAR(\epsilon_2)$
- 3) $VAR(y_3) = VAR(\eta_2) + VAR(\epsilon_3)$
- 4) $VAR(y_4) = \lambda_4^2 VAR(\eta_2) + VAR(\epsilon_4)$
- 5) $VAR(x_1) = VAR(\xi_1) + VAR(\delta_1)$
- 6) $VAR(x_2) = \lambda_6^2 VAR(\xi_1) + VAR(\delta_2)$
- 7) $COV(y_1 y_2) = \lambda_2 VAR(\eta_1)$
- 8) $COV(y_1 y_3) = COV(\eta_1 \eta_2) + COV(\epsilon_1 \epsilon_3)$
- 9) $COV(y_1 y_4) = \lambda_4 COV(\eta_1 \eta_2)$
- 10) $COV(y_1 x_1) = COV(\eta_1 \xi_1)$
- 11) $COV(y_1 x_2) = \lambda_6 COV(\eta_1 \xi_1)$
- 12) $COV(y_2 y_3) = \lambda_2 COV(\eta_1 \eta_2)$
- 13) $COV(y_2 y_4) = \lambda_2 \lambda_4 COV(\eta_1 \eta_2)$
- 14) $COV(y_2 x_1) = \lambda_2 COV(\eta_1 \xi_1)$
- 15) $COV(y_2 x_2) = \lambda_2 \lambda_6 COV(\eta_1 \xi_1)$
- 16) $COV(y_3 y_4) = \lambda_4 VAR(\eta_2)$
- 17) $COV(y_3 x_1) = COV(\eta_2 \xi_1)$
- 18) $COV(y_3 x_2) = \lambda_6 COV(\eta_2 \xi_1)$
- 19) $COV(y_4 x_1) = \lambda_4 COV(\eta_2 \xi_1)$
- 20) $COV(y_4 x_2) = \lambda_4 \lambda_6 COV(\eta_2 \xi_1)$
- 21) $COV(x_1 x_2) = \lambda_6 VAR(\xi_1)$

We want to use these equations to verify that each parameter to be estimated in the model can be solved for in terms of the variance or covariance of the x_i and/or y_i 's. Using the identified parameters we must then consider the structural equations and their parameters. Make a list of the parameters to be identified and solve for each. The following is a table of the parameters to be identified and the equations used.

	Equation used		Equation used
λ_2	10,14	λ_4	17,19
λ_6	10,11	$VAR(\xi_1)$	21 and λ_6
$VAR(\epsilon_1)$	1, 7 and λ_2	$VAR(\epsilon_2)$	λ_2 and 7, 2
$VAR(\epsilon_3)$	$\lambda_4, 16, 3$	$VAR(\epsilon_1)$	16, 4, λ_4
$COV(\epsilon_1 \epsilon_2)$	$\lambda_4, 8, 9$	$VAR(\delta_1)$	$\lambda_6, 21, 5$
$VAR(\delta_2)$	$\lambda_6, 21, 6$		

This proves all the parameters from the measurement equations are identified. There are still 5 parameters left to check ($\beta_1, \gamma_1, \gamma_2$, and two terms from the variance of the error terms). These come from the structural equations:

$$\eta_1 = \gamma_1 \xi_1 + \zeta_1 \quad [\text{eq st1}]$$

$$\eta_2 = \beta_1 \eta_1 + \gamma_2 \xi_1 + \zeta_2 \quad [\text{eq st2}]$$

To identify γ_1 and $VAR(\zeta_1)$ we will obtain two equations with two unknowns. Multiply [eq st1] by itself for one equation. For the second, multiply [eq st1] by ξ_1 , take expectations of both equations. The two equations have only γ_1 and $VAR(\zeta_1)$ as unknowns, all other parameters are already proved identified. To identify the remaining three parameters we will work with [eq st2]. First multiply [eq st2] by η_2 and take expectations to obtain the equation $COV(\eta_1 \eta_2) = \beta_1 VAR(\eta_1) + \gamma_2 COV(\xi_1 \eta_1) + E[\eta_1 \zeta_2]$. We do not have an assumption that sets the last term equal to zero. To see that it is zero, multiply [eq st1] by ζ_2 and take expectations. The right side of the equation is zero by previous assumptions. Now multiply [eq st1] by ξ_1 and take expectations. Use this and the equation above to solve two equations in two unknowns. This identifies the parameters β_1 and γ_2 . The last parameter to identify is $VAR(\zeta_2)$. This can be done by multiply [eq st2] by itself and taking expectations. Since all parameters are identified the model is identified.

4.3 LISREL Commands

The following LISREL program includes the SPSSX statements necessary for execution. LISREL will always print the LISREL commands again on page 2 of the output. The line numbers on the left side of the page are generated by SPSSX and will be used in the following discussion to explain each line of the program.

For VMS V4.1 UNIV. OF AR. AT LITTLE ROCK
 License Number 18805

```

1 TITLE Lisrel example - STABILITY OF ALIENATION
2 SUBTITLE DATA FROM WHEATON ET. AL.(1977) - LISREL
3 SET WIDTH=80
4
5 INPUT PROGRAM
6 NUMERIC DUMMY /*DUMMY VARIABLE
7 END FILE /* DUMMY FILE
8 END INPUT PROGRAM
9
10 USERPROC NAME=LISREL
  
```

```

THERE ARE 2913947 BYTES OF MEMORY AVAILABLE.
THE LARGEST CONTIGUOUS AREA HAS 2331157 BYTES.
11 Lisrel ALIENATION MODEL B Lisrel Manual III.54-63
12 DA NI=6 NO=932
13 CM
14
15 11.834
16 6.947 9.364
17 6.819 5.091 12.532
18 4.783 5.028 7.495 9.986
19 -3.839 -3.889 -3.841 -3.625 9.61
20 -21.899 -18.831 -21.748 -18.775 35.522 450.288
21 LA
22 'ANOM 67' 'POWL 67' 'ANOM 71' 'POWL 71' 'EDUC' 'SEI'
23 MO NY=4 NX=2 NE=2 NK=1 BE=SD PS=DI TE=SY
24 FR LY 2 1 LY 4 2 LX 2 1 TE 3 1
25 ST 1 LY 1 1 LY 3 2 LX 1 1
26 LE
27 'ALIEN 67' 'ALIEN 71'
28 LK
29 'SES'
30 OU ALL
31 END USER
  
```

The TITLE and SUBTITLE are SPSSX commands are used for documentation. Since the data will be read in LISREL, SPSSX will not have an active file defined. Lines 5 thru 8 are used to set up the dummy active file and dictionary necessary for SPSSX to execute. USERPROC NAME = LISREL calls LISREL and all commands after this statement are LISREL commands.

LISREL commands may be abbreviated to two characters. Blanks are used to separate different keywords and may not be used within keywords or commands and may not be placed before or after an equal (=) sign. LISREL commands are not required to begin in column 1 (SPSSX commands must begin in column 1). The User's Guide is not clear but upper and lower case seem to matter. All commands and keywords must be upper case. All parameters are expected to be on the same line as the command although command lines can be continued by ending a line the a C.

The first statement in LISREL must be the title. More than one line is allowed for the title, to continue the title to more than one line type any character (non-blank) in column 80. The next command (DA) on line 12 gives the specification of the data. NI=6 indicates that there are six observable variables and NO=932 sets the sample size as 932.

Now the input matrix will be read. LISREL can read data in several forms. The data could be raw data, a moment matrix, a covariance or correlation matrix. CM indicates that the data that follows to be read is a covariance matrix. The data is entered in the program in free form with spaces acting as delimiters. The matrix is assumed to be ordered with endogenous (y) variables first then exogenous (x) variables. If this is not the case the SE command can be used to reorder the variables. In this example the matrix is in the correct order.

The LA card (line 21) assigns 8 character labels to the observed variables as represented in the input matrix (or Raw data if used). On lines 26 thru 27, the LE and LK cards are used to assign labels to the unobservable independent (ξ) and dependent (η) variables respectively. Labels are necessary for documentation and readability only, although they may be used with the SE card to reorder the variables.

The model to be estimated is specified with the MO command. In this case on line 23, the number of y (NY) variables is 4, the number of x (NX) variables is 2, the number of ξ (NK) is 2 and there is only 1 η (NE) variable. The coefficient matrix β is, by default in LISREL, defined to be fixed to zero. In this example BE=SD defines β to be subdiagonal. This means the diagonal and upper triangle are fixed to zero and the subdiagonal is free to be estimated. The covariance matrix ψ is defined as diagonal. This implies that there is no covariance of the errors of the structural equation. TE=SY sets θ_e (Theta-Epsilon, the covariance matrix of the measures of errors for y variables) as a symmetric matrix. It is by default diagonal and free.

The previous commands defined the structure of the whole matrices. It is also possible to relax or put constraints on individual elements of a matrix. On line 24 the elements (2,1) and (4,2) of λ_y are freed. Line 25 gives the elements (1,1) and (3,2) of λ_y the starting value of 1. The (3,1)

element of θ_{ϵ} is freed so that the correlation between ϵ_1 and ϵ_3 will be estimated.

The OU command is used to indicate the output desired from LISREL. LISREL can obtain estimates by five different methods: IV (instrumental variables), TSLS (two-stage least squares), ULS (unweighted least squares), GLS (generalized least squares) and ML (maximum likelihood). All methods give consistent estimates for fully identified models. The default is ML in the LISREL program. Joreskog (1984) page II.26 and II.27 give details of the types of output that can be obtained.

4.4 Output and Explanation for Example 1

The following output generated by LISREL gives a summary of the specifications of the model and the output requested.

```

LISREL VI - VERSION 6.6
Lisrel  ALIENATION MODEL B  Lisrel Manual III.54-63

NUMBER OF INPUT VARIABLES  6
NUMBER OF Y - VARIABLES    4
NUMBER OF X - VARIABLES    2
NUMBER OF ETA - VARIABLES  2
NUMBER OF KSI - VARIABLES  1
NUMBER OF OBSERVATIONS    932

OUTPUT REQUESTED
TECHNICAL OUTPUT          YES
STANDARD ERRORS           YES
T - VALUES                YES
CORRELATIONS OF ESTIMATES YES
FITTED MOMENTS             YES
TOTAL EFFECTS             YES
VARIANCES AND COVARIANCES YES
MODIFICATION INDICES      YES
FACTOR SCORES REGRESSIONS YES
FIRST ORDER DERIVATIVES  YES
STANDARDIZED SOLUTION    YES
PARAMETER PLOTS           NO
AUTOMATIC MODIFICATION    NO

```

LISREL always prints the matrix that is to be analyzed. In this example it is the same as the input matrix. It is possible that the matrix to be analyzed is not the same type as the input matrix. Also if the variables were reordered by a SE card the new reordered matrix would be printed.

The determinant can be used as a measure of "ill conditioning" of the matrix. If the determinant is small relative to the magnitude of the diagonal elements, it is possible that there is a strong linear relationship between one or more of the observed variables. One or more of these variables should be excluded from the model, or use the ULS (Unweighted Least Squares) method instead of ML. By checking the determinant we can see that this is not the case in this example.

COVARIANCE MATRIX TO BE ANALYZED

	<u>ANOM 67</u>	<u>POWL 67</u>	<u>ANOM 71</u>	<u>POWL 71</u>	<u>EDUC</u>	<u>SEI</u>
ANOM 67	11.834					
POWL 67	6.947	9.364				
ANOM 71	6.819	5.091	12.532			
POWL 71	4.783	5.028	7.495	9.986		
EDUC	-3.839	-3.889	-3.841	-3.625	9.610	
SEI	-21.899	-18.831	-21.748	-18.775	35.522	450.28

DETERMINANT - 0.608057D+07

The output titled PARAMETER SPECIFICATIONS indicates the constraints placed on each element of each matrix involved in the model. A zero entry indicates that the element is not to be estimated, that is, it is fixed. An integer indicates that the entry is to be estimated. If two entries have the same integer index, these two entries have to be constrained to be equal. Check the parameter specification to verify that you have defined the model as you intended!

PARAMETER SPECIFICATIONS

LAMBDA Y

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ANOM 67	0	0
POUL 67	1	0
ANOM 71	0	0
POUL 71	0	2

LAMBDA X

	<u>SES</u>
EDUC	0
SEI	3

BETA

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ALIEN 67	0	0
ALIEN 71	4	0

GAMMA

	<u>SES</u>
ALIEN 67	5
ALIEN 71	6

PHI

	<u>SES</u>
SES	7

PSI

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
	8	9

THETA EPS

	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>
ANOM 67	10			
POUL 67	0	11		
ANOM 71	12	0	13	
POUL 71	0	0	0	14

THETA DELTA

	<u>EDUC</u>	<u>SEI</u>
	15	16

Initial estimates or starting values are automatically generated by LISREL. They are obtained from a non-iterative scheme and can therefore be obtained with a minimum of computer resources. These values are true estimates and can be used in the early stages of development of a model to save computer time. Starting values (if set by the researcher) and fixed values are not estimated.

INITIAL ESTIMATES (TSLS)

LAMBDA Y

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ANOM 67	1.000	0.000
POUL 67	1.003	0.000
ANOM 71	0.000	1.000
POUL 71	0.000	0.963

LAMBDA X

	<u>SES</u>
EDUC	1.000
SEI	4.975

BETA

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ALIEN 67	0.000	0.000
ALIEN 71	0.591	0.000

GAMMA

	<u>SES</u>
ALIEN 67	-0.571
ALIEN 71	-0.242

PHI

	<u>SES</u>
SES	7.140

PSI

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
	4.598	3.774

THETA EPS

	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>
ANOM 67	4.907			
POUL 67	0.000	2.397		
ANOM 71	1.736	0.000	4.751	
POUL 71	0.000	0.000	0.000	2.767

THETA DELTA

	<u>EDUC</u>	<u>SEI</u>
	2.470	273.564

Looking at β the elements should be interpreted as follows:

$\beta_{i,j}$ indicates that a unit change in η_j results in a change of $\beta_{i,j}$ units in η_i , with all other variables held constant. If the variables have been standardized, a standard deviation change in η_j results in $\beta_{i,j}$ standard deviation change in η_i , with all other variables held constant. However, because of indirect effects other variables are usually effected by a change in one variable. For this reason it is informative to examine the TOTAL EFFECTS of one variable on the others. LISREL will calculate the TOTAL EFFECTS (see later output). A similar interpretation can be made for γ .

LISREL ESTIMATES (MAXIMUM LIKELIHOOD)

LAMBDA Y

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ANOM 67	1.000	0.000
POUL 67	1.027	0.000
ANOM 71	0.000	1.000
POUL 71	0.000	0.971

LAMBDA X

	<u>SES</u>
EDUC	1.000
SEI	5.163

BETA

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ALIEN 67	0.000	0.000
ALIEN 71	0.617	0.000

GAMMA

	<u>SES</u>
ALIEN 67	-0.550
ALIEN 71	-0.211

PHI

	<u>SES</u>
SES	6.880

PSI

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
	4.705	3.866

THETA EPS

	<u>ANOM 67</u>	<u>POWL 67</u>	<u>ANOM 71</u>	<u>POWL 71</u>
ANOM 67	5.065			
POWL 67	0.000	2.215		
ANOM 71	1.887	0.000	4.812	
POWL 71	0.000	0.000	0.000	2.683

THETA DELTA

	<u>EDUC</u>	<u>SEI</u>
	2.730	266.896

SQUARED MULTIPLE CORRELATIONS FOR Y - VARIABLES

<u>ANOM 67</u>	<u>POWL 67</u>	<u>ANOM 71</u>	<u>POWL 71</u>
0.572	0.764	0.616	0.731

TOTAL COEFFICIENT OF DETERMINATION FOR Y - VARIABLES IS 0.952

SQUARED MULTIPLE CORRELATIONS FOR X - VARIABLES

<u>EDUC</u>	<u>SEI</u>
0.716	0.407

TOTAL COEFFICIENT OF DETERMINATION FOR X - VARIABLES IS 0.762

SQUARED MULTIPLE CORRELATIONS FOR STRUCTURAL EQUATIONS

<u>ALIEN 67</u>	<u>ALIEN 71</u>
0.306	0.501

TOTAL COEFFICIENT OF DETERMINATION FOR STRUCTURAL EQUATIONS IS 0.343

There are several indicators of the goodness-of-fit of the model. The squared multiple correlations and the coefficient of determination are calculated for the observed variables (x and y) and the structural equations when initial estimates or the maximum likelihood estimates are printed. For the ith observed variable the squared multiple correlation is defined as:

$$1 - \theta^*_{i,i} / s_{i,i}$$

where $\theta^*_{i,i}$ is the estimated error variance of the ith observed variable and $s_{i,i}$ is the observed variance of the ith variable. The coefficient of determination is defined:

$$1 - |\theta^*| / |S|$$

where $|\theta^*|$ is the determinant of the matrix containing the estimated error variance and $|S|$ is the determinant of the

covariance matrix. For the structural equations the squared multiple correlation is defined as:

$$1 - \text{var}(\zeta_i) / \text{var}(\eta_i)$$

and the coefficient of determination for the structural equations is defined as:

$$1 - |\Phi| / |\text{Cov}(\eta)|$$

In both cases the squared multiple correlations and the coefficient of determination should be between zero and one, with large values indicating a good model. LISREL will not restrict the values between zero and one, negative values are an indication of a bad model.

The multiple correlations can be viewed as a measure of the strength or reliability of variable. In this example the variables ANOM 67 and POWL 67 are measures of the latent variable ALIEN 67, of these POWL 67 is the more reliable indicator since it has the largest squared multiple correlations. For ALIEN 71, POWL 71 is the more reliable indicator. For the exogenous variable SES, EDUC is the more reliable indicator with a squared multiple correlation of .716 for the Maximum Likelihood estimate.

For the structural equations the squared multiple correlations can be interpreted as the proportion of variance in each η_i variable explained by the ζ_i variables. In this example there is only one ζ variable, SES. ALIEN 71 has a squared multiple correlation of .501 which is higher than for ALIEN 67 (.306). That is, SES is a better indicator of the variance of ALIEN71 than ALIEN67.

The total coefficient of determination can be viewed as the strength of several relationships jointly. For example the value of .952 for the Y variables indicates that all the Y variables jointly serve as a good measure of the η_i latent variables. In this case these are ALIEN71 and ALIEN67.

MEASURES OF GOODNESS OF FIT FOR THE WHOLE MODEL:
 CHI-SQUARE WITH 5 DEGREES OF FREEDOM IS 6.33
 (PROB. LEVEL = 0.275)
 GOODNESS OF FIT INDEX IS 0.998
 ADJUSTED GOODNESS OF FIT INDEX IS 0.990
 ROOT MEAN SQUARE RESIDUAL IS 0.754

Three measures are used to judge how well the model "fits". The χ^2 along with it's degrees of freedom, the goodness-of-fit index and the root mean square residual. These measures indicate the overall fit of the model. It can happen that the overall fit is "good" but individual relationships are modeled poorly.

The χ^2 measure is $N-1$ times the minimum value of the fitting function. The degrees of freedom is $\frac{1}{2} k(k+1) - t$ where k is the number of observed variables ($k=p+q$) and t is the number of independent parameters to be estimated. Notice t should be equal to the largest index in the parameter specification. Under some situations the χ^2 measure can be viewed as a test statistic for testing a hypothesis against an alternative, but the ideal conditions for this to be valid seldom exist. To use the χ^2 in this way it is assumed that the observed variables have normal distribution, the sample size must be large enough to justify the asymptotic properties of the χ^2 test, and the analysis must be based on the sample covariance matrix, not the correlation matrix. Instead the χ^2 measure should be used as a measure of the goodness of fit with large values (relative to the degrees of freedom) indicating a bad fit and small values indicating a good fit.

The goodness-of-fit is a measure of the relative amount of variances and covariances jointly accounted for by the model. The adjusted goodness-of-fit index is adjusted for the degrees of freedom for the model. Both of these values should be between zero and one, although LISREL does not restrict them to any range. The root mean square residual can be thought of as an average of the residual variances and covariances. Both goodness of fit indexes and the root mean square residual can be used to compare the fit of two different models for the same data. The goodness-of-fit index can also be used to compare the fit of the model for different data.

Other indicators of the fit of the model include an examination of the parameter estimates. Unreasonable values indicate that the model may not be identified or the sample size too small. High correlation of parameter estimates may indicate a "nearly" non-identified model. The correlation matrix for the estimates is computed and can be printed by requesting PC on the OU command line.

If the indicators of the goodness of fit of the model are not acceptable, the next question is how to improve the model. Sorbom suggests selecting the parameter with the largest partial derivative of the fitting function with respect to the fixed parameter. This seems logical since the derivative indicates change in the fitting function. However it can be shown that freeing a parameter with a large derivative does not always correspond to a large change in the Chi-square.

A better indicator, introduced by Joreskog and Sorbom, of which parameter to free, is called the modification index. For each fixed and constrained parameter, the modification index is defined as $N/2$ times the ratio between the squared first-order derivative and the second-order derivative. Freeing the parameter with the largest modification index will result in a reduction of the Chi-square at least as great as the modification index. Free only **one** parameter at a time and only if it makes sense in the model to estimate that parameter. A modification index of zero indicates that the model will not be identified if the parameter is freed or that the parameter is already freed.

In this example the largest modification index is for element (4,2) of the matrix θ_ϵ (Theta-epsilon). Freeing this parameter would indicate that the error terms for the measure of y_2 (POWL 67) and y_4 (POWL 71) are correlated. We are guaranteed that the χ^2 will drop at least by 1.59 and a loss of one degree of freedom.

MODIFICATION INDICES

LAMBDA Y			
	<u>ALIEN 67</u>	<u>ALIEN 71</u>	
ANOM 67	0.000	0.700	
POWL 67	0.000	0.700	
ANOM 71	0.534	0.000	
POWL 71	0.534	0.000	

LAMBDA X	
	<u>SES</u>
EDUC	0.000
SEI	0.000

BETA		
	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ALIEN 67	0.000	0.000
ALIEN 71	0.000	0.000

GAMMA	
	<u>SES</u>
ALIEN 67	0.000
ALIEN 71	0.000

PHI		PSI	
	<u>SES</u>	<u>ALIEN 67</u>	<u>ALIEN 71</u>
SES	0.000	0.000	0.000

THETA EPS				
	<u>ANOM 67</u>	<u>POWL 67</u>	<u>ANOM 71</u>	<u>POWL 71</u>
ANOM 67	0.000			
POWL 67	0.000	0.000		
ANOM 71	0.000	0.534	0.000	
POWL 71	0.700	1.591	0.000	0.000

THETA DELTA	
<u>EDUC</u>	<u>SEI</u>
0.000	0.000

MAXIMUM MODIFICATION INDEX IS 1.59
FOR ELEMENT (4, 2) OF THETA EPS

The next two sections of output contain the STANDARD ERRORS and the T-VALUES. The standard errors are used to calculate the T-values, but they can also be used to detect a bad model. Extremely large standard errors may indicate that the model is nearly non-identified (some of the parameters cannot be estimated).

T-VALUES are calculated by dividing the parameter estimate by its standard error. T-VALUES greater than two (in absolute value) indicate that the parameter is different than zero. If the fit of the model is not good, consider fixing (to zero) parameters with T-VALUES less than 2 in magnitude. (Be cautious of the T-VALUE when the sample size is small.) In this example all parameters have significant T-VALUES.

STANDARD ERRORS

LAMBDA Y

	<u>ALLEN 67</u>	<u>ALLEN 71</u>
ANOM 67	0.000	0.000
POUL 67	0.053	0.000
ANOM 71	0.000	0.000
POUL 71	0.000	0.049

LAMBDA X

	<u>SES</u>
EDUC	0.000
SEI	0.421

BETA

	<u>ALLEN 67</u>	<u>ALLEN 71</u>
ALLEN 67	0.000	0.000
ALLEN 71	0.050	0.000

GAMMA

	<u>SES</u>
ALLEN 67	0.053
ALLEN 71	0.049

PHI

	<u>SES</u>
SES	0.658

PSI

	<u>ALLEN 67</u>	<u>ALLEN 71</u>
	0.433	0.343

THETA EPS

	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>	THETA DELTA	
					<u>EDUC</u>	<u>SEI</u>
ANOM 67	0.371				0.516	18.193
POUL 67	0.000	0.318				
ANOM 71	0.240	0.000	0.395			
POUL 71	0.000	0.000	0.000	0.330		

T-VALUES

LAMBDA Y

	<u>ALLEN 67</u>	<u>ALLEN 71</u>
ANOM 67	0.000	0.000
POUL 67	19.320	0.000
ANOM 71	0.000	0.000
POUL 71	0.000	19.647

LAMBDA X

	<u>SES</u>
EDUC	0.000
SEI	12.253

BETA

	<u>ALLEN 67</u>	<u>ALLEN 71</u>
ALLEN 67	0.000	0.000
ALLEN 71	12.420	0.000

GAMMA

	<u>SES</u>
ALLEN 67	-10.293
ALLEN 71	-4.292

PHI

	<u>SES</u>
SES	10.457

PSI

	<u>ALLEN 67</u>	<u>ALLEN 71</u>
	10.863	11.256

THETA EPS

	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>	THETA DELTA	
					<u>EDUC</u>	<u>SEI</u>
ANOM 67	13.650				5.287	
14.670						
POUL 67	0.000	6.972				
ANOM 71	7.864	0.000	12.175			
POUL 71	0.000	0.000	0.000	8.133		

Correlations of parameter estimates are calculated for the ML method only. Parameters with high correlations are indications of a model that is nearly non-identified.

CORRELATIONS OF ESTIMATES

	<u>LY 2 1</u>	<u>LY 4 2</u>	<u>LX 2 1</u>	<u>BE 2 1</u>	<u>GA 1 1</u>	<u>GA 2 1</u>
LY 2 1	1.000					
LY 4 2	0.232	1.000				
LX 2 1	0.000	0.000	1.000			
BE 2 1	0.100	-0.297	-0.112	1.000		
GA 1 1	0.375	0.125	-0.517	0.145	1.000	
GA 2 1	-0.055	0.108	-0.278	0.572	0.173	1.000
PH 1 1	0.000	0.000	-0.708	0.118	0.554	0.296
PS 1 1	-0.605	-0.215	-0.157	-0.114	0.002	0.024
PS 2 2	-0.105	-0.528	-0.029	0.082	-0.040	-0.043
TE 1 1	0.546	0.117	0.000	0.057	0.204	-0.032
TE 2 2	-0.669	-0.063	0.000	0.047	-0.247	0.130
TE 3 1	0.238	0.210	0.000	-0.126	0.095	-0.047
TE 3 3	0.133	0.589	0.000	-0.174	0.073	0.065
TE 4 4	-0.068	-0.664	0.000	0.168	-0.052	-0.104
TD 1 1	0.000	0.000	0.763	-0.150	-0.589	-0.329
TD 2 2	0.000	0.000	-0.577	0.071	0.317	0.171

CORRELATIONS OF ESTIMATES

	<u>PH 1 1</u>	<u>PS 1 1</u>	<u>PS 2 2</u>	<u>TE 1 1</u>	<u>TE 2 2</u>	<u>TE 3 1</u>
PH 1 1	1.000					
PS 1 1	0.165	1.000				
PS 2 2	0.031	0.130	1.000			
TE 1 1	0.000	-0.325	-0.055	1.000		
TE 2 2	0.000	0.231	-0.015	-0.531	1.000	
TE 3 1	0.000	-0.136	-0.025	0.503	-0.230	1.000
TE 3 3	0.000	-0.125	-0.351	0.145	-0.033	0.476
TE 4 4	0.000	0.101	0.208	-0.026	-0.057	-0.189
TD 1 1	-0.738	-0.211	-0.039	0.000	0.000	0.000
TD 2 2	0.427	0.099	0.019	0.000	0.000	0.000

CORRELATIONS OF ESTIMATES

	<u>TE 3 3</u>	<u>TE 4 4</u>	<u>TD 1 1</u>	<u>TD 2 2</u>
TE 3 3	1.000			
TE 4 4	-0.556	1.000		
TD 1 1	0.000	0.000	1.000	
TD 2 2	0.000	0.000	-0.544	1.000

The matrix labeled FITTED MOMENTS is the fitted sigma. Remember that LISREL is estimating parameters so that the fitted sigma is "close" to the matrix S. The FITTED RESIDUALS is the matrix of S - Σ. The size of the residuals should be small relative to the size of the elements of S. This may be hard to determine if S is not a correlation matrix. It will be easier to inspect the NORMALIZED RESIDUALS matrix. A

simple test of how well the model accounts for the element $s_{i,j}$ is to check the magnitude of the elements of the NORMALIZED RESIDUAL matrix. If an element is larger than |2| the corresponding $s_{i,j}$ element is not accounted for well in the model.

FITTED MOMENTS AND RESIDUALS

FITTED MOMENTS

	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>	<u>EDUC</u>	<u>SEI</u>
ANOM 67	11.850					
POUL 67	6.965	9.364				
ANOM 71	6.876	5.121	12.559			
POUL 71	4.843	4.972	7.522	9.986		
EDUC	-3.783	-3.883	-3.790	-3.680	9.610	
SEI	-19.528	-20.046	-19.566	-18.997	35.522	450.28

FITTED RESIDUALS

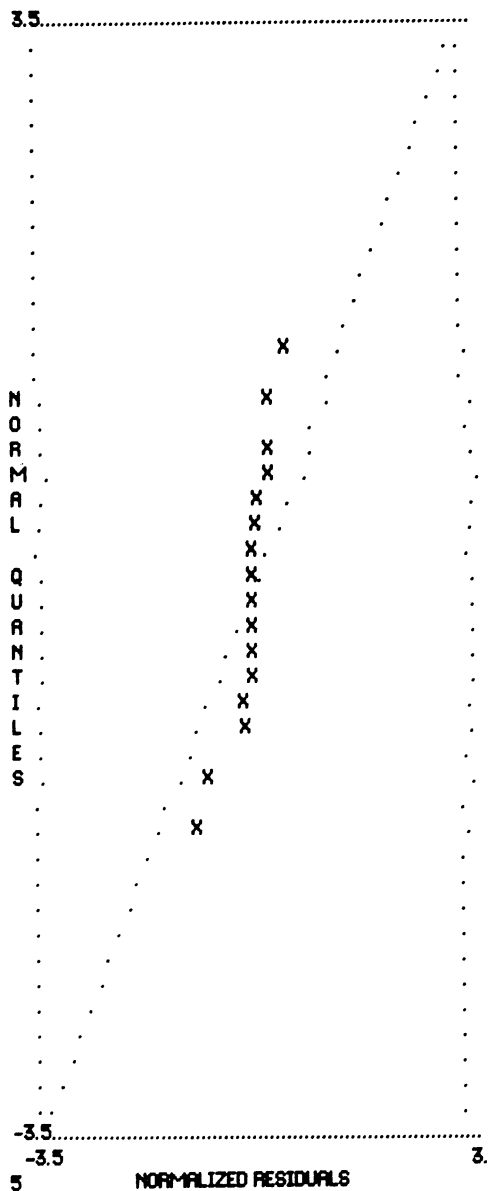
	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>	<u>EDUC</u>	<u>SEI</u>
ANOM 67	-0.016					
POUL 67	-0.018	0.000				
ANOM 71	-0.057	-0.030	-0.027			
POUL 71	-0.060	0.056	-0.027	0.000		
EDUC	-0.056	-0.006	-0.051	0.055	0.000	
SEI	-2.371	1.215	-2.182	0.222	0.000	-0.001

NORMALIZED RESIDUALS

	<u>ANOM 67</u>	<u>POUL 67</u>	<u>ANOM 71</u>	<u>POUL 71</u>	<u>EDUC</u>	<u>SEI</u>
ANOM 67	-0.029					
POUL 67	-0.043	0.000				
ANOM 71	-0.123	-0.075	-0.046			
POUL 71	-0.154	0.158	-0.060	0.000		
EDUC	-0.152	-0.018	-0.134	0.159	0.000	
SEI	-0.957	0.546	-0.857	0.097	0.000	0.00

A visual summary of the fit of the model as indicated by the normalized residuals is shown in the following QPLOT. Single points are labeled with an X, multiple points with an *, and a 45 degree line by small dots. Fit a straight line through the X's and *'s. Use the 45 degree line to determine if the line you have fitted has slope larger than one (good fit of model), nearly equal to one (fair fit of model), or less than one (poor fit of model). In our example the slope is greater than one indicating a good fit of the model. If the line you fit is non-linear, the model may be incorrectly specified.

Q PLOT OF NORMALIZED RESIDUALS



From the path diagram it is obvious that there are both direct and indirect effects of the ξ_1 variable SES on η_1 and y_1 variables. The direct effects are found in the β and γ matrices. The total effects shown in the following matrices are the sum of the indirect and direct effects. For example examine the matrix TOTAL EFFECTS OF KSI (SES) ON ETA. The total effect of SES on the η_1 variable ALIEN 67 is simply the value of γ_1 (-.550), since there are no indirect effects. The total effect of SES on η_2 is the sum of the direct effects ($\gamma_2 = -.211$) and the indirect effects ($\gamma_1 = -.55$ times $\beta_1 = .617$). In this example the TOTAL EFFECTS OF ETA ON ETA is exactly the same as the β matrix. It is interesting to note that the total effects of SES on each η_1 variable is nearly equal.

TOTAL EFFECTS

TOTAL EFFECTS OF KSI ON ETA

<u>SES</u>	
ALIEN 67	-0.550
ALIEN 71	-0.551

TOTAL EFFECTS OF KSI ON Y

<u>SES</u>	
ANOM 67	-0.550
POUL 67	-0.564
ANOM 71	-0.551
POUL 71	-0.535

TOTAL EFFECTS OF ETA ON ETA

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ALIEN 67	0.000	0.000
ALIEN 71	0.617	0.000

LARGEST EIGENVALUE OF (I-BETA)*(I-BETA)-TRANPOSED (STABILITY INDEX) IS 0.381

TOTAL EFFECTS OF ETA ON Y

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ANOM 67	1.000	0.000
POUL 67	1.027	0.000
ANOM 71	0.617	1.000
POUL 71	0.599	0.971

Various variances and covariances may be requested. These are the matrices listed below.

VARIANCES AND COVARIANCES

ETA - ETA

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ALIEN 67	6.785	
ALIEN 71	4.988	7.747

ETA - KSI

<u>SES</u>	
ALIEN 67	-3.783
ALIEN 71	-3.79

Y - ETA

	<u>ALIEN 67</u>	<u>ALIEN 71</u>
ANOM 67	6.785	4.988
POUL 67	6.965	5.121
ANOM 71	4.988	7.747
POUL 71	4.843	7.522

Y - KSI

<u>SES</u>	
ANOM 67	-3.783
POUL 67	-3.883
ANOM 71	-3.790
POUL 71	-3.680

X - ETA

	ALIEN 67	ALIEN 71
EDUC	-3.783	-3.790
SEI	-19.528	-19.566

X - KSI

	SES
EDUC	6.880
SEI	35.522

Other output not printed in this paper includes the first order derivatives, the factor scores regressions and the standardized solution. The factor scores are the regression coefficients of unobserved variables on observed variables. The standardized solution is the solution in which η_1 and η_2 are scaled to unit variance.

4.5 Summary

The analysis of this model indicates that it is a good fit as indicated by the χ^2 of 6.33 with 5 degrees of freedom. Also the largest modification index is 1.59. Indicating that the only improvement to the χ^2 would result from freeing element (4,2) of θ_{ϵ} . The path diagram with the estimated coefficients is shown in figure 2.

In summary to assess the fit or detect the lack of fit you should examine:

- 1) Parameter estimates
- 2) Standard errors (for ML method only)
- 3) Squared multiple correlations

4) Coefficients of determination

5) Correlations of estimates (for ML method only)

If any of these seem unreasonable (too large, negative variances, correlations which are larger than one in magnitude, ect.), either the model is fundamentally wrong or the data is not informative. To adjust the model a specification search can be performed. This involves eliminating parameters that are not significant as indicated by the T-values. That is, if the free parameter is not judged to be different from zero, fix the parameter to zero. Another option is to free the parameter with the largest modification index. But only if it makes sense to the model to estimate the parameter. When these changes to the model are bases on the fit of the data it is necessary to verify the model by testing it against another independent data set and each new model must be tested for identification.

Measures of the overall fit of the model are found by examining:

- 1) Chi-square, Degrees of freedom (ML method only)
- 2) Goodness-of-fit index
Adjusted goodness-of-fit index
- 3) Root mean square residual

And for a more detailed assessment of fit the following parameters should be examined.

- 1) Residuals
- 2) Normalized residuals
- 3) Q-plot of normalized residuals
- 4) Modification indices

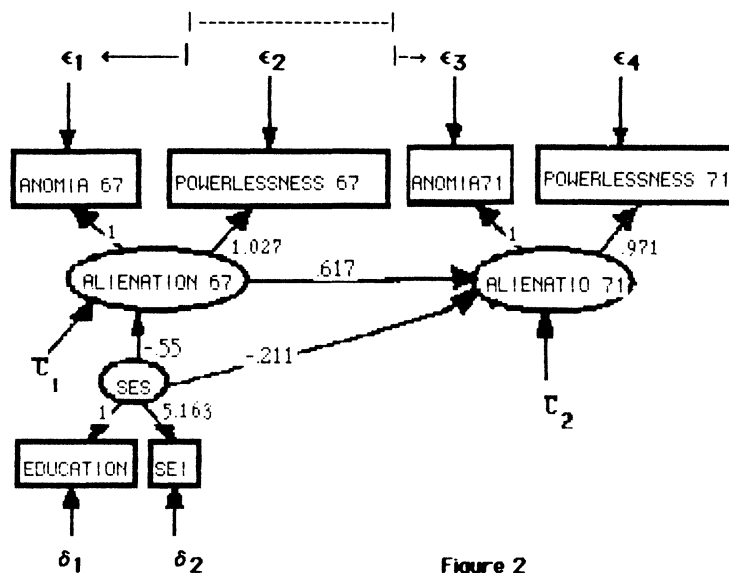


Figure 2

5 Example 2 - involves only observable variables

The following analysis uses data from a study in progress at the Center of Child Development and Education at the University of Arkansas at Little Rock. The data were collected by eleven researchers from six sites in North America. The study involves a total of 930 children beginning in the 70's. The data gathered at each site was similar, however because the data was pooled after the fact, some sites are missing data for certain variables. The six sites are Washington (N=193), Ontario (N=121), North Carolina (N=84), Texas (N=255), California (N=129) and Arkansas (N=148). The seven variables that will be examined in this analysis are mothers education, an indicator of the child's IQ at 12, 24, and 36 months of age, and an indicator of the home environment at each point in time. The information for mothers education was nearly complete with a total only 14 cases missing.

The measure, used at all six sites, of the child's IQ at both 12 and 24 months was the Bayley Scales of Infant Development. This measure was available on 810 children at 12 months and 652 children at 24 months. The Stanford-Binet Intelligence Test was given at the age of 3 years. IQ data were available on 628 children.

The Home Observation for Measurement of the Environment Inventory was used to determine the three home environment variables. The HOME Inventory is designed to assess the quality of stimulation and support available to a child in the home environment. HOME's at age one were available on 865 families, at age two on 507 families and at age three on 559 families. HOME's at age two were not available from two sites, California and Ontario. HOME's at age three were not available from the Texas site.

5.1 Missing Data

The correlation matrix used by LISREL was created using pair-wise deletion of missing data. This means that when data were missing the variables involved were deleted in the calculations of that particular cell of the matrix. The result is a matrix with coefficients calculated with different values of N. In the LISREL model the χ^2 and the modification index are directly related to the value of N. Remember, the χ^2 is defined as N-1 times the minimum value of the fitting function and the modification index is defined as N/2 times the ratio between the squared first-order derivative and the second-order derivative. As long as we choose a reasonable

value for N and keep it constant in all the models we can use these two indicators as we have in the past. A low χ^2 relative to the degrees of freedom will have to be used cautiously, but changes in the χ^2 from one model to the next will still be an indication of improvement in the fit of the model. A large drop compared to the difference in degrees of freedom, indicates that the change made in the model represents a real improvement. A drop in χ^2 close to the difference in degrees of freedom indicates that the improvement in fit is obtained by "capitalizing on chance", and the added parameters may not have real significance and meaning (Joreskog, 1984). The modification index can be used as before, since the use of the modification index depended on the values of the index relative to each other and the value of the χ^2 . The correlation matrix and each cells value for N follows.

Table 4

	momed	MDI12	HOME12	MDI24	HOME24	IQ36
HOME36 momed	1.00 (0)					
MDI12	.283 (798)	1.00 (0)				
HOME12	.464 (851)	.253 (764)	1.00 (0)			
MDI24	.526 (643)	.482 (633)	.505 (610)	1.00 (0)		
HOME24	.496 (501)	.336 (453)	.614 (471)	.582 (408)	1.00 (0)	
IQ36	.495 (621)	.440 (605)	.530 (590)	.696 (572)	.618 (376)	1.00 (0)
HOME36	.463 (554)	.301 (535)	.623 (544)	.552 (507)	.738 (297)	.5936 (506) 1.00 (0)

5.2 LISREL Specifications

Since all of the variables are observed some of the matrices in the LISREL model will be set to zero and the x_i and y_i variables will be set equal to the ξ_i and η_i variables, respectively. In this model there is only one independent variable, mother's education. All variables in this analysis are:

x_1 = mother's education

y_1 = MDI at 12 months y_4 = HOME at 24 months

y_2 = HOME at 12 months y_5 = IQ at 36 months

y_3 = MDI at 24 months y_6 = HOME at 36 months

The following path diagram illustrates the first attempt to define a model for this data. To simplify the drawing the paths are not labeled. Mother's Education is allowed to directly effect all 6 dependent observed variables. Path coefficients are to be calculated from each y variables to the variables in the next time period. For example the direct effect of MDI12 is calculated for both MDI24 and HOME24. The covariances among the errors in equations are allowed and indicated with curved lines. Covariance between the errors at the same period of time might occur if the model were misspecified by the exclusion of variables that affect both the indicator of HOME and IQ.

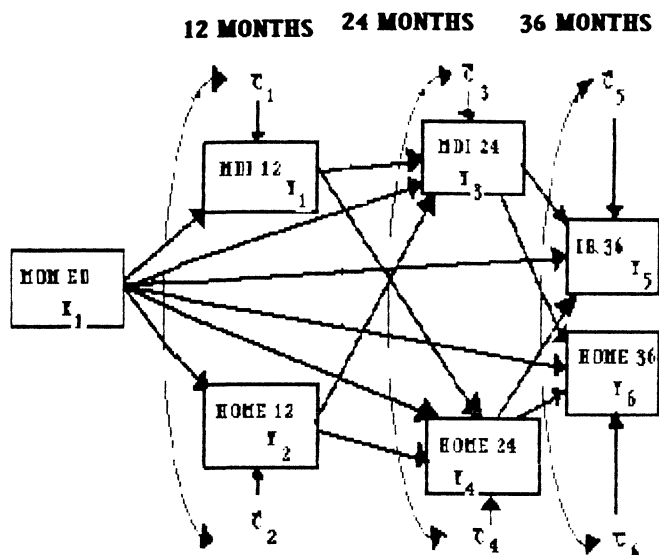


Figure 3

The structural equations for this path diagram are:

- 1) $y_1 = \gamma_{11} x_1 + \zeta_1$
- 2) $y_2 = \gamma_{21} x_1 + \zeta_2$
- 3) $y_3 = \beta_{31} y_1 + \beta_{32} y_2 + \gamma_{31} x_1 + \zeta_3$
- 4) $y_4 = \beta_{41} y_1 + \beta_{42} y_2 + \gamma_{41} x_1 + \zeta_4$
- 5) $y_5 = \beta_{53} y_3 + \beta_{54} y_4 + \gamma_{51} x_1 + \zeta_5$
- 6) $y_6 = \beta_{63} y_3 + \beta_{64} y_4 + \gamma_{61} x_1 + \zeta_6$

Since all the variables in this analysis are observed, we set the x_i and y_i variables equal to the ξ_i and η_i variables. In terms of the LISREL model this means that the x_i and y_i variables are perfect indicators of the ξ_i and η_i variables. We will therefore set both θ_{ϵ} and θ_{δ} to zero and the factor

matrices λ_y and λ_x will each be set to identity matrices. ψ is defined by default in LISREL to be symmetric and free with the variances of the error terms on the diagonal and the covariance of the error terms elsewhere. We will request that only three covariance estimates be calculated and the other off diagonal terms fixed. Since the matrix is by default symmetric we will only need to fix or free the lower elements. β is defined in the LISREL program to be fixed to zero, this means that none of the elements of this matrix will be estimated unless they are freed by the researcher. The appropriate elements of this matrix will be freed as indicated by the equations above. The full matrix γ is by default estimated, therefore we need only fix the elements that should not be estimated. As for ϕ , since we have only one x variable and are using the correlation matrix in the analysis, this variance/covariance matrix will not give any new information and will not be estimated. The LISREL commands for this model follow, the SPSSX commands necessary to execute this program are identical to the first example.

```

DA NI=7 NO=500 MR=KM
LA
'MOM ED' 'MDI 12' 'HOME 12M' 'MDI 24' 'HOME 24M' 'IQ 36' 'HOME 36M'
SE
'MDI 12' 'HOME 12M' 'MDI 24' 'HOME 24M' 'IQ 36' 'HOME 36M' 'MOM ED'
KM SY
      1.000
      .2838      1.000
      .4648      .253      1.000
      .5265      .4821      .5053      1.000
      .4962      .3362      .6145      .5818      1.000
      .4953      .4402      .5301      .6967      .6180      1.000
      .4634      .3011      .6231      .5525      .7379      .5936
      1.000
MO NX=1 NY=6 BETA=FU
PA BE
000000
000000
110000
110000
001100
001100
PA PS
1
11
001
0011
00001
000011
OU TV RS EF MI

```

We have chosen a value of 500 for the number of observations as indicated by NO=500 on the first line. There are 7 input variables and the correlation matrix (MA=KM) is used as input. The variables are labeled with the LA command. The matrix to be analysed should be ordered with the y variables first. This is not the case in this example. The input matrix is reordered with the SE command. The symmetric correlation matrix is read after the KMSY command. Since the program uses spaces to separate elements of the matrix, the values could have been listed across the line instead of in matrix form.

All remaining commands (except the OU command) are to define the free and fixed elements of the various matrices involved in the model. In the first example the FI and FR commands were used to fix or free individual parameters. Sometimes this is inconvenient, instead the PA command can be used. This command is a signal to the program to read a pattern matrix of zeroes and ones, where a zero means a fixed element and a one means a free element. In this example this is done for the BE (β) and the PS (ψ) matrices.

The last command controls the output that is printed and the method of estimation used. Since a method is not indicated on the OU line, ML is used by default. The other output requested includes TV, the t-values, RS, the residuals and Q-plot, EF, the total effects, MI, the modification indices.

5.3 Identification

The next question that must be answered involves that of identification. The parameters must be proved to be identified before the results of the estimation can be used. The parameters to be identified include all the free elements of β , all of γ and the six variances and three covariances of ζ .

Beginning with the first equation multiply by x_1 and take expectations. Since the covariance term in the resulting equation is equal to zero by the specification of the model, γ_{11} is proved identified. Now multiply equation 1 by itself and take expectations to obtain $\text{VAR}(y_1) = \gamma_{11}^2 \text{VAR}(x_1) + 2\gamma_{11} \text{COV}(x_1, \zeta_1) + \text{VAR}(\zeta_1)$. In this equation the covariance term is zero by assumption and γ_{11} is identified, therefore $\text{VAR}(\zeta_1)$ is identified. Equation 2 can be manipulated in a similar way to

identify the parameters γ_{21} and $\text{VAR}(\zeta_2)$. The $\text{COV}(\zeta_1, \zeta_2)$ term can be shown to be identified by multiplying equation 1 and 2, taking expectations and using terms previously identified.

Next use equation 3 to obtain three equations in three unknowns as follows: multiply equation 3 by x_1 for the first equation, by y_1 for the second, and by y_2 for the third, take expectations of all three equations. The last two equations will have four unknowns, this fourth unknown is a covariance term involving y_i ($i=1,2$) and ζ_3 . These terms can be shown to equal zero by multiplying both equation 1 and 2 by ζ_3 . Equations 4, 5 and 6 can be handled in a similar way to identify the remaining parameters. Since each parameter can be solved for in terms of the variances and covariances of the observed variables, the model is identified.

5.4 LISREL Results and Explanation

The results of the ML estimation of this and two additional models are summarized in table 5. The fit of the first model is poor, notice the χ^2 of 53.53 with 4 degrees of freedom and the AGF (Adjusted Goodness of Fit Index) of .799. All of the parameters estimated are judged to be different from zero by the t-values. The largest modification index (35.44) is for β_{46} . Estimating β_{46} would be estimating the direct effect of HOME 36 on HOME 24 which is not reasonable. The second largest modification index (31.874) is for β_{62} , to estimate the direct effect of HOME 12 on HOME 36.

In model 2 β_{62} is freed. Model 2 must also be proved to be identified. Since only one new parameter is added and one equation affected, we need only look at equation 6 again. With this new parameter equation 6 now looks like:

$$y_6 = \beta_{62}y_2 + \beta_{63}y_3 + \beta_{64}y_4 + \gamma_{61}x_1 + \zeta_6$$

Using the same pattern used before, multiply this equation first by x_1 , then by y_1 , then by y_2 , then by y_3 and take expectations of all four equations. These four equations will have the β and γ terms as unknown as well as three covariance terms. These terms involving y_i ($i=1,2,3$) and ζ_6 can be shown to equal zero by multiplying equations 1, 2 and 3 by ζ_6 , taking expectations and using assumptions.

Model 2 is an improvement over model 1 by all indicators. The χ^2 was reduced by 33.5, more than the 31.8 that we were guaranteed. The GF and AGF are both larger and the RMSR is smaller. The t-value for γ_{61} was not large enough to safely assume it be different than zero. The largest modification index (18.03) is for $\psi_{5,3}$. Freeing this parameter would allow the errors for MDI 24 and IQ 36 to be correlated. Because of the nature of these tests, it is reasonable that the error in measurement of these two variables could be correlated.

In model 3 $\psi_{5,3}$ is freed. Again there is significant improvement of the fit of the model. The χ^2 was reduced by 18.61 and the degrees of freedom by only 1, the GF and AGF indicators were increased and RMSR is again smaller. Notice that two parameters had t-values too small to safely assume to be different than zero.

Other results from model 3 follow. The squared multiple correlations for the structural equations is interpreted as the proportion of variance of each dependent variable explained by the independent variables. In this example we have only one independent variable, mother's education. Mothers's education explains more of the variance of the home environment at 36 months than the other variables.

SQUARED MULTIPLE CORRELATIONS FOR STRUCTURAL EQUATIONS

	<u>MDI 12</u>	<u>HOME 12M</u>	<u>MDI 24</u>	<u>HOME 24M</u>	<u>IQ 36</u>
<u>HOME36M</u>	0.081	0.216	0.461	0.455	0.466
	0.603				

The total effects of mother's education on the intelligence and home variables are given in the following results. It is interesting that, except for the MDI at 12 months, the total effect of mother's education is nearly equal on all variables. And finally the complete path diagram using model 3 for the coefficients.

TOTAL EFFECTS

TOTAL EFFECTS OF X ON Y

	<u>MOM ED</u>
MDI 12	0.284
HOME 12M	0.465
MDI 24	0.527
HOME 24M	0.496
IQ 36	0.495
HOME 38M	0.463

Table 5

<u>PARAMETER</u>	<u>MODEL 1</u>	<u>MODEL 2</u>	<u>MODEL 3</u>
β_{31}	.325	.325	.315
β_{32}	.282	.282	.298
β_{41}	.151	.151	.151
β_{42}	.466	.466	.466
β_{53}	.473	.473	.875
β_{54}	.293	.293	.122
β_{62}	-	.219	.229
β_{63}	.158	.121	.120
β_{64}	.607	.511	.506
γ_{11}	.284	.284	.286
γ_{12}	.465	.465	.465
γ_{13}	.303	.303	.299
γ_{14}	.237	.237	.237
γ_{15}	.101	.101	-.026*
γ_{16}	.079	.044*	.043*
ψ_{11}	.919	.919	.919
ψ_{21}	.121	.121	.121
ψ_{22}	.784	.784	.784
ψ_{33}	.541	.541	.539
ψ_{43}	.149	.149	.145
ψ_{44}	.545	.545	.545
ψ_{53}	-	-	-.236
ψ_{55}	.439	.439	.534
ψ_{65}	.069	.055	.054
ψ_{66}	.428	.397	.397
χ^2	53.53	20.03	1.32
df	4	3	2
p	0.0	0.0	.516
GF	.971	.989	.999
AGF	.799	.896	.989
RMSR	.032	.020	.004

where * indicates a t-value less than 2 in magnitude, and - indicates the parameter was not free

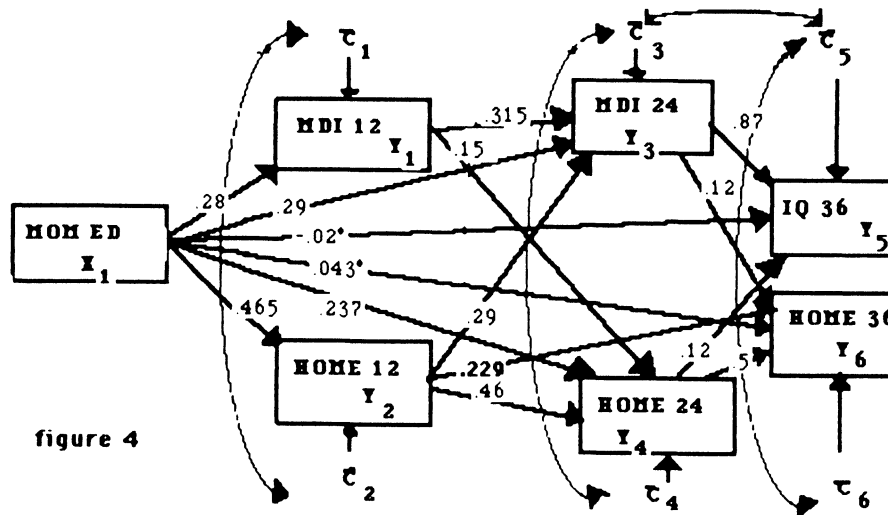


figure 4

REFERENCES

Afifi, A. and Clark, Virginia (1984). *Computer-aided Multivariate Analysis*. London: Lifetime Learning Publications.

Dillon, William R. and Goldstein, Matthew (1984) *Multivariate Analysis: Methods and Applications*. New York: John Wiley & Sons.

Everitte, B. S. and Dunn G. (1983) *Advanced Methods of Data Exploration and Modelling*. London: Heinemann Educational Books.

Heise, David R. (1975) *Causal Analysis*. New York: John Wiley & Sons.

Goldberger, Arthur S. and Duncan, O.D., Eds. (1973) *Structural Equation Models in the Social Sciences*. New York: Seminar Press.

Hogg, Robert V. and Craig, Allen T. (1978). *Introduction to Mathematical Statistics*. New York: Macmillan Publishing Co., Inc.

Joreskog, Karl G. and Sorbom, Dag (1984) *LISREL VI Analysis of Linear Structural Relationships by the Method of Maximum Likelihood*. Uppsala, Sweden: University of Uppsala.

Kenny, David A. (1979) *Correlation and Causality*. New York: Wiley.

Long, J. Scott (1983) *Covariance Structure Models: An introduction to LISREL*. London: Sage.

Long, J. Scott (1983) *Confirmatory Factor Analysis: A Preface to LISREL*. London: Sage.

Marsden, Peter V. Ed. (1981) *Linear Models in Social Research*. London: Sage.

Noble, Ben and Daniel, James W. (1977) *Applied Linear Algebra*. Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Nunnally, Jum C. (1978) *Psychometric Theory*. New York: McGraw-Hill.

Pedhazur, Elazar J. (1973) *Multiple Regression In Behavioral Research: Explanation and Prediction*. New York: Holt, Rinehart and Winston.

SPSS Inc. (1984) *Userproc LISREL: Using LISREL VI within SPSSX*. Chicago: SPSS Inc.

Tatsuloka, Maurice M. (1971) *Multivariate Analysis: Techniques for Educational and Psychological Research*. New York: John Wiley & Sons.

Robert B. Goldstein and Gertrude Stabiner
Eye Research Institute of Retina Foundation
20 Staniford St
Boston, MA 02114

ABSTRACT

RS/1 has been used for system resource chargeback and long-term tracking of system usage. The resource chargeback application (a set of interrelated DCL command files, FORTRAN programs, RPL programs, and RS/1 tables) runs automatically during the first night of each month. By using RS/1 for this purpose we (1) save the cost of an expensive chargeback program, and (2) can tailor the application to our specific needs. Long-term tracking of our the resource usage of our VAX780 and PDP11/70 computers is implemented through RS/1 tables which are maintained manually. The graphs illustrate the evolution of our computer systems and are useful in presenting company DP needs to management.

INTRODUCTION

This paper addresses two issues: system resource chargeback and long-term tracking of system usage. System chargeback is problematic because the DEC accounting utility, under VMS, does not handle disk space usage. Another utility, DISKQUOTA, must be used to obtain this information. Since DISKQUOTA reports usage based on UIC, not on account number, we must combine the results of the DISKQUOTA and ACCOUNTING utilities. Long-term tracking of system usage has been implemented for 6 years, and the resulting graphs illustrate the evolution of our computer systems. We will also show the changes in resource usage that are the result of system tuning efforts.

CHARGEBACK

Although many commercial chargeback packages are available for VMS, we chose to implement our own system for the following reasons:

- o We have greater control over the process.
- o It is more interesting for us to write our own procedures.
- o It gives the Computer Unit at the Eye Research Institute (ERI) an opportunity to use RS/1, which we must know in order to support it.
- o It is tailored to our specific needs.

Implementing our own package was not cheaper than purchasing a commercial package. The cost of the man-hours we spent writing

the procedures and "baby-sitting" the programs has equaled the price we would have paid for a commercial package.

The chargeback procedure is divided into two parts: the chargeback phase and the RS/1 phase. The chargeback phase gathers the information by running the DEC-supplied utilities; the RS/1 phase merges the results produced by the utilities.

Figure 1 shows a flow diagram of the chargeback phase. The steps are:

1. Run ACCOUNTING to produce a report and to start a new accounting file. The reported fields are CPU time, Elapsed time, and Pages printed.
2. Run AUTHORIZE (UAF) to obtain a current user list.
3. Run DISKQUOTA to obtain the disk usage as a function of UIC.
4. Run several FORTRAN programs to perform the following:
 - a. Strip the extraneous headers from the log files produced by the above steps.
 - b. Match the ACCOUNT numbers with the UICs in the disk usage records. At the same time, combine multiple records per account (caused by multiple users having the same account number) into one record per account.

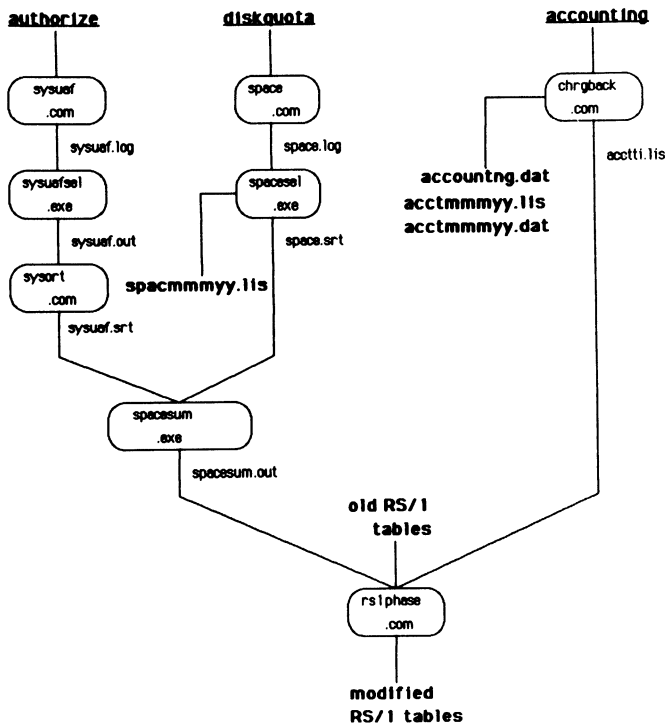


Figure 1. Flow Diagram of Chargeback Phase

c. Produce two final files for input into RS/1: SPACESUM.OUT, which contains the disk usage as a function of account, and ACCTI.LIS, which contains the other usage information (CPU, Elapsed, Pages) as a function of account.

5. Submit the batch procedure to run the RS/1 phase.

The RS/1 phase makes use of a "shell table," which is an empty version of the final report table. Figure 2 shows a portion of the shell table, in which the empty cells are filled by the procedures. Figure 3 shows a flow diagram of the RS/1 phase. The steps are:

Shell For VAX Usage

0	1 Fund Number	2 UIC	3 Elapsed Time	4 CPU Time	5 ERI2 Blocks	6 ERI3 Blocks	7 Pages	8 Charge (\$)
1	Smith	F10008	[70,7]	-	[70,7]			
2	Jones	F10043	[120,4]	-	[120,4]			
3	User X	F10044	[120,5]	-	[120,5]			
4	User Y	F10121	[70,12]	-	[70,12]			
			.					
			.					
			.					
33	Goldstein	F21003	[160,14]	-	[160,14]			
34	Librarian	F21004	[160,13]	-	[160,13]			
35	Medical Database	F21422	[240,1]	-	[240,3]			
36	Low Vision	F21423	[240,2]	-	[240,2]			
37	NDFQ	F704	[160,12]	-	[160,12]			
38	Biguser	F707	[2,5]	-	[220,27]			
39	Totals							

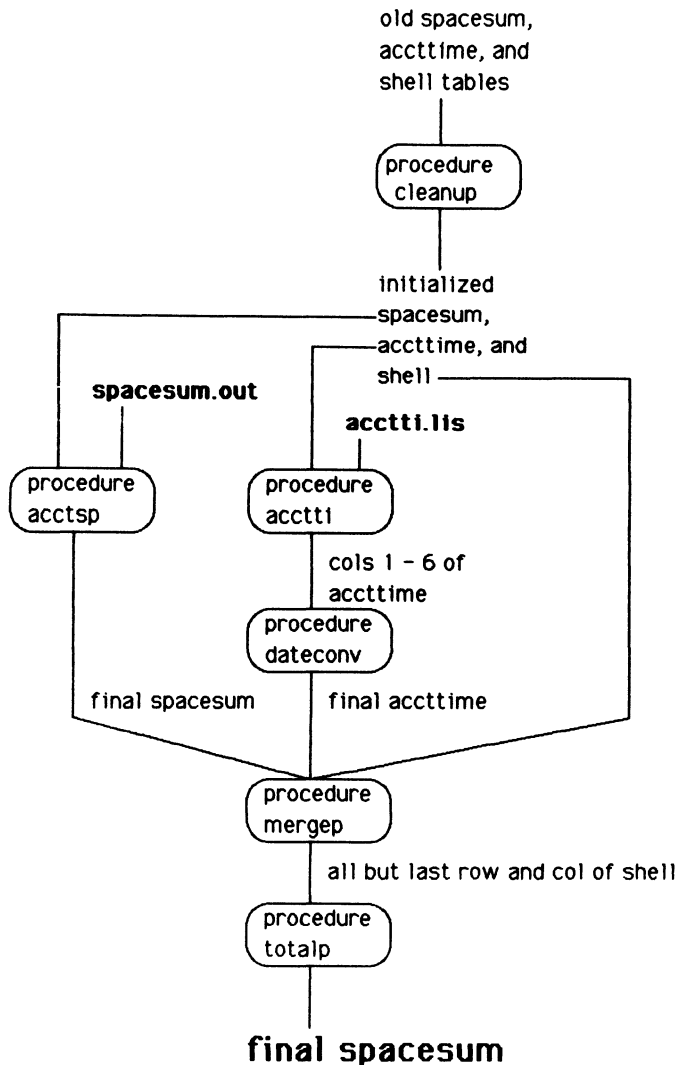
Figure 2. Shell Table

1. Initialize the RS/1 tables. A separate RS/1 table is needed to hold the data from the two files produced in the first phase.
2. Import the files from the chargeback phase.
3. Convert elapsed and CPU times (of the hh:mm:ss type) produced in phase 1 to fractional hours. (For example, 03:15:00 is converted to 3.25.)
4. Merge the accounting and disk usage data into the shell table.
5. Apply the charging formula, which is in the form:

$$\$ = A*\text{cpu hours} + B*\text{connect hours} + C*\text{number of blocks} + D*\text{number of pages} + E$$
6. Total all charges and insert into the last row of the table.

The advantages of this method of performing system resource chargeback were discussed above. Its disadvantages are:

- o The need for constant maintenance. With every new user an entry must be added into the shell table.
- o The programs depend heavily on the formatted output of several DEC utilities. If the output format changes (as it did when VMS went from version 3 to version 4), the programs must be altered.
- o It is not robust. The system fails about once every 3 months for various causes, for example, when a deleted account has not been removed from the shell table, when an incorrect algorithm was used for the end of year change, or when the batch and command procedures were improperly synchronized.



- o The system and all peripherals must be up and running at midnight on the first of the month.
- o RS/1 runs on only one of the machines in the net. Therefore, data must be transferred, via DECnet, from one machine to the other.

LONG-TERM TRACKING

Our site was one of the earliest to use the RS/1 package. We have been recording system usage statistics in RS/1 tables since 1980. The resulting graphs illustrate the evolution of our PDP11/70 computer system and are probably typical of usage patterns of other computer systems.

Figure 4 shows four graphs that plot system resource usage versus time. The "connect time" shown in the upper left panel exhibits a pattern that is repeated in the other three panels. During the first 2 years usage gradually climbed to a maximum, followed by about 1-1.5 years in which usage remained at this level and response time on the PDP11/70 was intolerable.

Idle time during this period (averaged over 7 days/week and 24 hours/day) was only about 30%. In mid-1983 we removed one big application and resource usage dropped, but response time was still very poor. In December 1983 we obtained a VAX780 and started to transfer applications from the PDP11/70 to the VAX. Resource usage dropped to a minimum on the PDP11/70 and has remained low because only two applications are left on the PDP. These are scheduled to be transferred to the VAX soon.

We are currently keeping similar types of statistics on the VAX (Figure 5). Since we are careful not to overload the VAX, the VAX CPU time usage does not show a rising trend, although the VAX elapsed time does show signs of increase.

On both machines disk usage rises rapidly to the critical region (defined as 80% of total disk capacity), and is kept at this level only by constantly reminding users to clean out their disk directories. Figure 6 shows this behavior.

SYSTEM TUNING

We have used RS/1 to display the results of system tuning efforts. Figure 7 shows the number of pagefaults resulting from two different RS/1 commands. The data were obtained by setting the WSDEFAULT, WSQUOTA, and WSEXTENT parameters to the same value for a selected user. Then, when no other users were on the system, we issued the

```
$ rsl
```

command and examined the number of pagefaults. Within RS/1, we issued the

```
# dir
```

command and examined again the number of pagefaults. By means of these measurements we determined the optimum values for the WSDEFAULT, WSQUOTA and WSEXTENT parameters for RS/1 users.

The best measure of system response is user perception. It matters little, for example, that "swapping is at 3%" if the users still think that response time is poor. Therefore, we wrote a program called "Response Logger," which upon user logout queries the user on how well the system responded to his needs. The user is asked to give the system a letter grade of A, B, C, D, or F. A record is then written to a log file containing the grade and other information such as number of users, time of day, and user category. At the end of the month we examine the log file, plot the results, and take appropriate actions if necessary.

Figure 8 shows some of the information that may be obtained with the Response Logger. The upper left panel shows how system response degraded after VMS 4.1 was installed and how it improved after more memory was

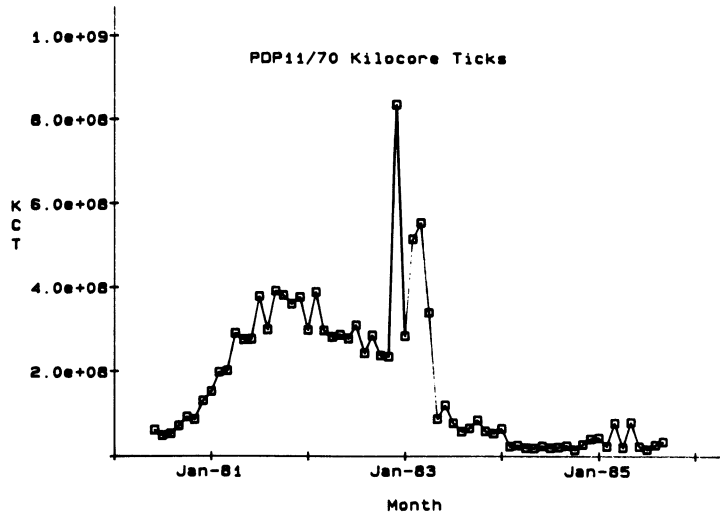
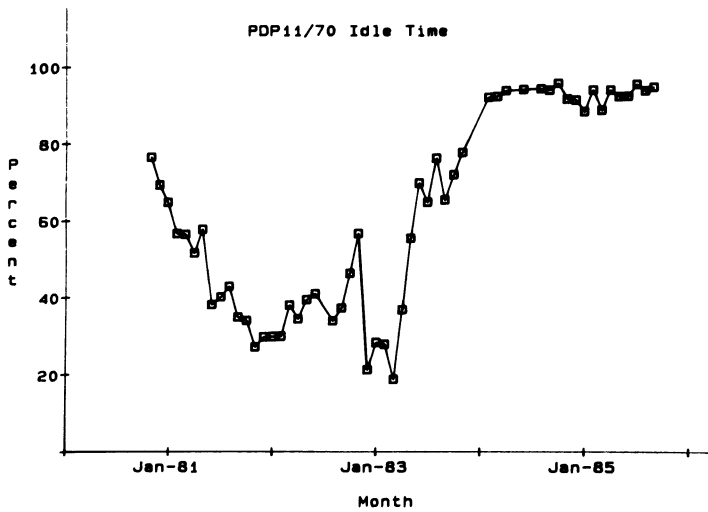
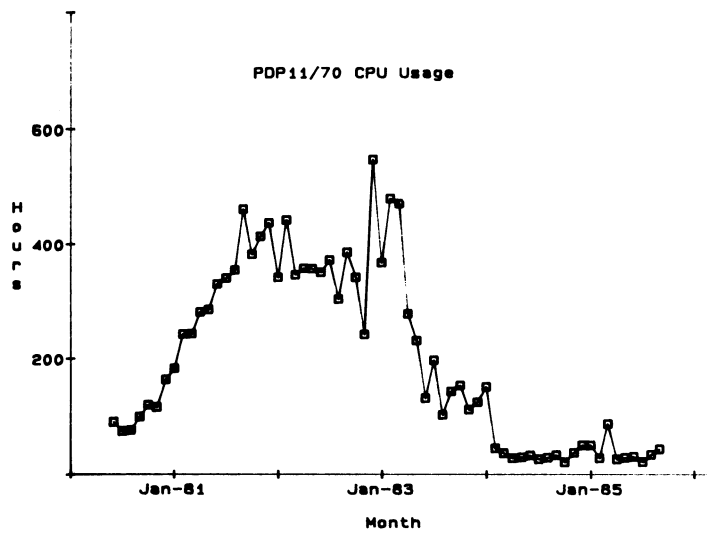
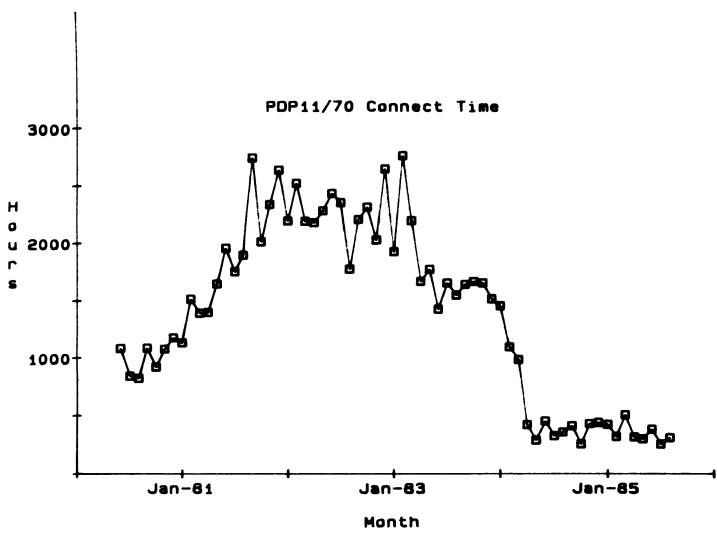


Figure 4. PDP11/70 Usage for 6 Year Period

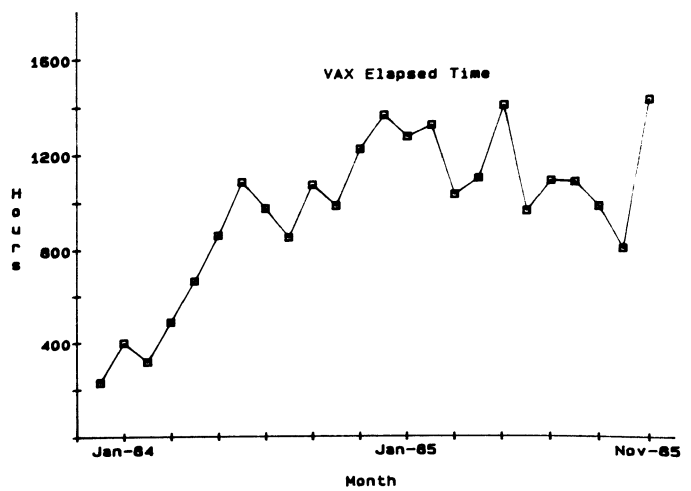
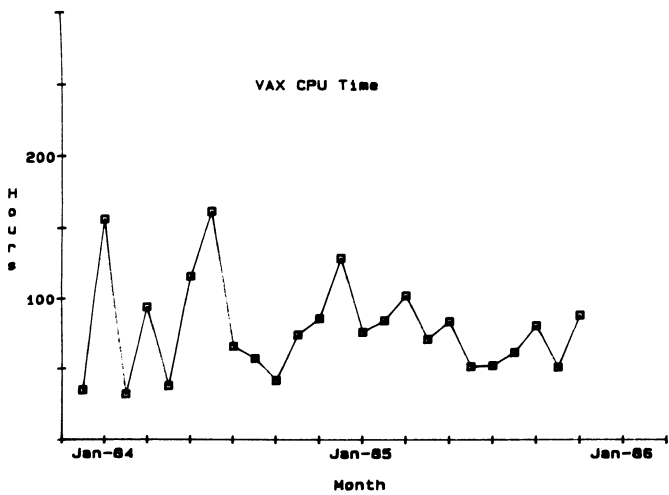


Figure 5. VAX Usage

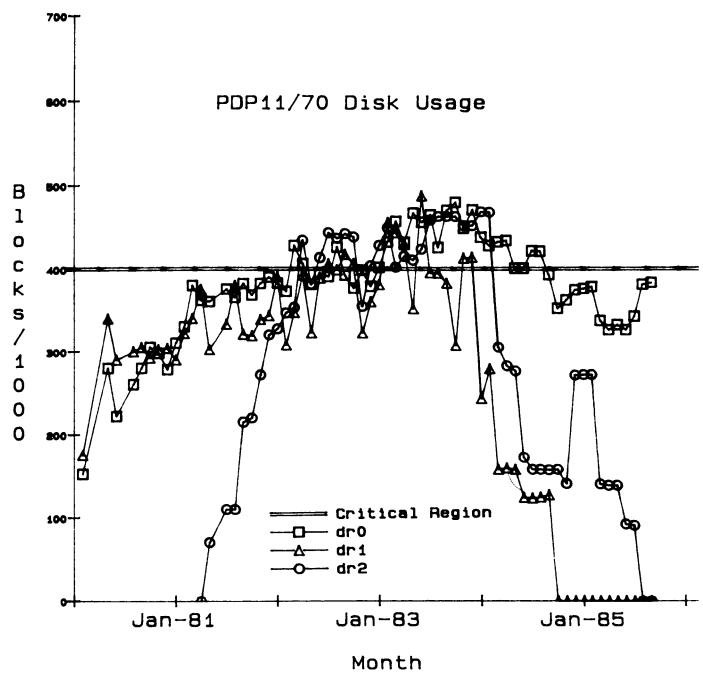
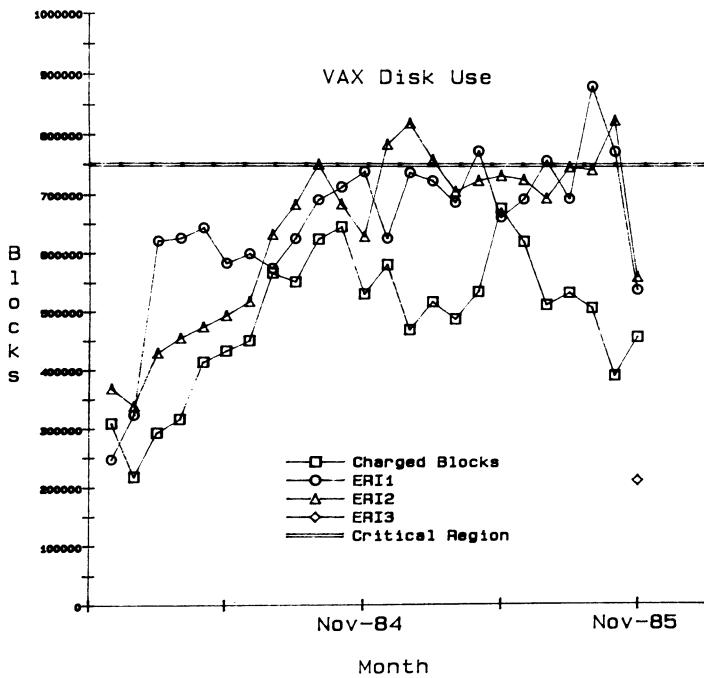


Figure 6. Disk Usage for VAX and PDP11/70

added. The upper right panel shows how system response degrades as the number of users changes throughout the day. One measure of the sensitivity of this program is that the lower left panel exhibits an expected dip at lunchtime. The corresponding improvement in response time is shown in the lower right panel.

CONCLUSION

The chargeback procedure is currently a mixture of command procedures, FORTRAN programs, and RPL procedures. To make it more general, simpler, and more robust, we should attempt to reduce this mixture by replacing the RPL procedures with FORTRAN, or by removing RS/1 from the procedure altogether.

We have been using RS/1 and the Response Logger to obtain information about system utilization, to display tuning results, and to accomplish chargeback of resources. These procedures help us satisfy our users' needs and justify the acquisition of new equipment when necessary.

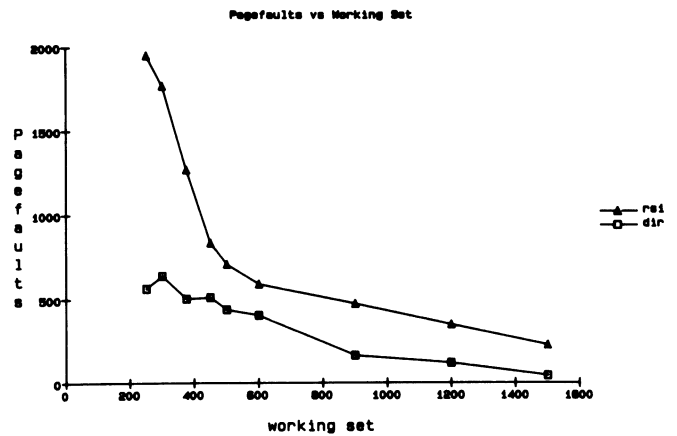


Figure 7. Tuning Measurements for RS/1

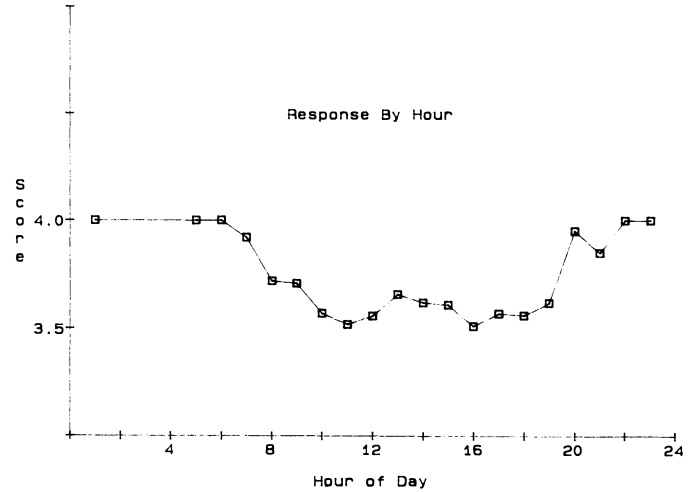
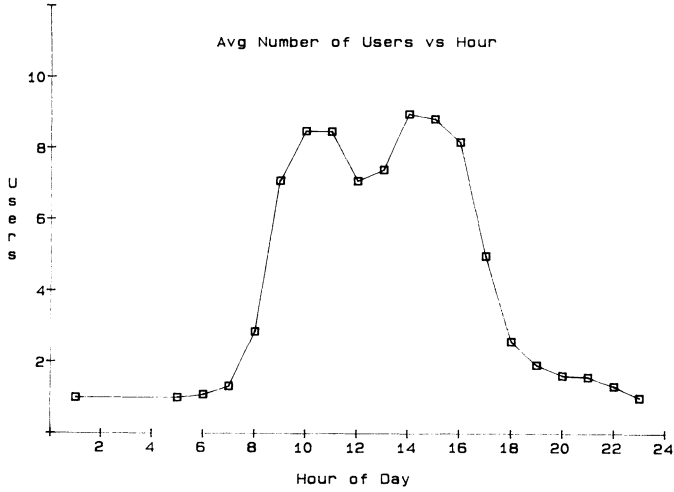
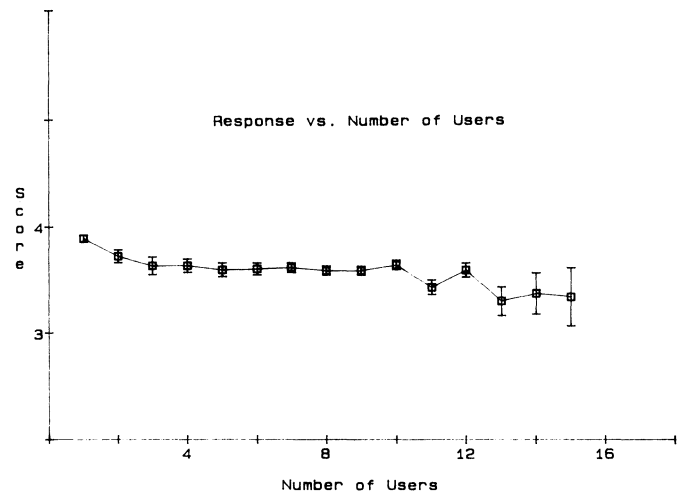
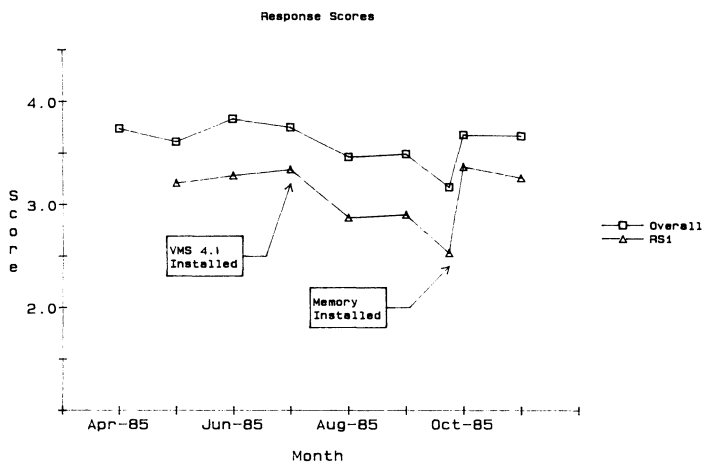


Figure 8. Response Logger Results

Capt Scott B. Eckert
Robert L. Ewing
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

Dr. Gary B. Lamont
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583
Visiting Professor
Wright State University
Dayton OH

Abstract

DICES is a user interactive system which permits the implementation of digital controllers based on the TMS32010 digital signal processing microprocessor for a given single input-single output plant model. DICES partitions the controller design into second-order 3D filter sections and quantizes the coefficients. These coefficients are loaded into a generic filter program written in TMS32010 assembly language, which is then assembled and loaded into the TMS32010 for execution. The controller is then placed in the forward or feedback path of an analog computer system which simulates the plant. Performance data is obtained via IEEE-488 controlled instrumentation using a VAX 11/780 VMS system.

INTRODUCTION

Digital controllers and filters can be designed using a myriad of techniques. At the Air Force Institute of Technology (AFIT), several CAD packages are available to electrical engineering students for designing digital controllers and computer-aided design programs that currently reside on the AFIT VAX 11/780 with Virtual Memory System (VMS) (1,2). However, the controller is typically only simulated in the design/analysis software package using high-precision coefficients. It would be desirable during the design phase to use the designed controller parameters to implement the control algorithm on a microcomputer system operating within an analog computer simulation of the plant. This would permit direct observation of effects of finite word size, the filter coefficients, changes in sampling rates, etc. This controller could then be used as a cascade or feedback controller in a real-world hardware/software environment. The physical plant is modelled using analog transfer functions. A performance evaluation of the implemented controller is normally desired. This performance evaluation consists of determination of the "standard" figures of merit (3) that most designers are interested in obtaining (e.g.) peak overshoot, settling time, peak time, frequency response, etc.). The analysis is typically performed using both the open-loop and

closed-loop transfer function. The implementation of the controller design is tested in a closed-loop configuration since this is the normal operational mode.

SCOPE

The purpose of this work is to investigate the requirements of a digital controller/digital signal processor implementation and performance evaluation system and to implement a basic system which provides current capability while allowing for future expansion. It is not the intent to develop a new test equipment, but rather to integrate a practical system that will assist in the design, coding, test, and evaluation of linear, time-invariant digital controllers. It is assumed that one of several digital design software packages will be used to design the controller/processor algorithms (1,2).

SYSTEM CONCEPT

DICES permits implementation of a digital controller or filter design. The system interfaces with existing software design packages such as TOTAL or ICECAP (1,2) which are available on the AFIT VAX 11/780 VMS system. It will also

allow direct input of DSP or controller algorithms in the form of Z-domain transfer functions. Once the controller transfer function is determined, it can be implemented on DICES using the 16-bit Texas Instrument TMS32010 DSP microprocessor. (4) This, together with a model of the basic plant can be formed into a closed-loop system for evaluation. The performance of this system can then be assessed using test instrumentation programmed via an IEEE-488 instrument bus. The VAX 11/780 contains a National Instruments GPIB11-2 IEEE-488 Interface Controller and highlevel software interface to the VMS device driver (5). The overall system concept is shown in figure 1.

HARDWARE DESCRIPTION

DICES consists of many hardware items which perform the various system functions required to form an automated system. The main hardware items are listed below, each with a brief description of its function.

1. VAX 11/780 and peripherals - Executes the main DICES FORTRAN program and remotely controls the IEEE-488 instruments via the IEEE-488 bus.
2. Bruel and Kjaer 2032 System Analyzer - Performs step response and frequency (magnitude and phase) testing on the closed-loop plant/controller system. Uses Fast Fourier Transform techniques to obtain frequency response data.
3. EAI TR-48 Analog Computer System - Simulates the plant model to allow closed-loop testing of the digital controller.
4. TMS32010 Evaluation Module - Contains the TMS32010 microprocessor, RAM, ROM and executive. A separate board contains the Analog-to-Digital and Digital-to-Analog converter used to interface with the analog computer.
5. Wavetek 172B Signal Generator - Provides the step input used to perform step response testing on the system-under-test.
6. National Instruments GPIB11-2 IEEE-488 Interface Board - Provides the interface between the VAX 11/780 and an IEEE-488 instrument bus.

The above items are used to provide stimulus to the closed-loop system, implement the digital controller, execute the CAD package and DICES software and then perform measurements of the system. These results can then be displayed and used to determine if a design iteration is required for the digital controller.

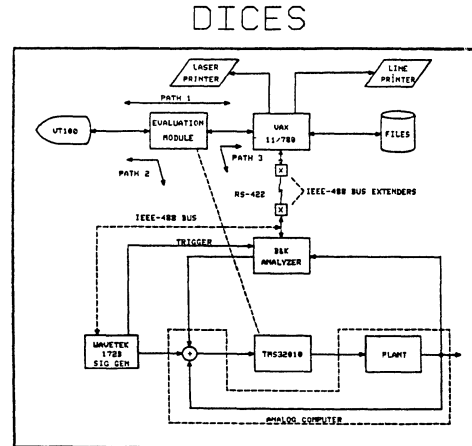


Figure 1. System Concept

SOFTWARE.

DICES software consists of three main modules, DICES main program, VAX VMS device driver, and TMS32010 controller code. Each of these modules is briefly discussed below.

1. DICES Main Program - Coordinates the general flow through the "control problem" (see figure 2). Program presents option menus to user for selection of desired function. This program relies heavily on use of DEC Forms Management System (FMS) for menu generation to select desired steps in the design and simulation of the controller (see figure 2). The main program implements digital controller design by partitioning the desired controller transfer function into second-order sections. Sections are then implemented on a TMS32010 microcomputer. Performance testing is then initiated and controlled using IEEE-488 compatible instruments.

2. VAX VMS IEEE-488 Card Interface Driver - This software provides the interface to the IEEE-488 card. There are three levels of interface (see figure 3), two of which provide a simplified interface to the IEEE-488 instrument bus. The VAX VMS device driver resides as a system driver and performs the detailed handshaking with the IEEE-488 hardware - transparent to the application program.

3. TMS32010 Object Code - This object code is generated by the TMS Assembler from a source file which contains the digital controller parameters. The object code is generated on the VAX and downloaded via an RS-232 link to the TMS32010 to implement the digital controller design.

Example Problem

A simple plant transfer function is given which has unacceptable closed-loop step-response behavior (figures 4 and 5), i.e., peak overshoot is excessive (1.45 units).

$$G_p(s) = \frac{2}{s(s+1)(s+2)} \quad (1)$$

A digital controller is desired which will improve the step-response characteristics of this plant. Following a design session using DICES as the coordinator, a design is obtained which improves the step-response characteristics.

$$G_c(z) = \frac{.3974 (z-.995)}{(z-.998)} \quad (2)$$

The theoretical and actual responses of the simulation are shown in figures 6 and 7.

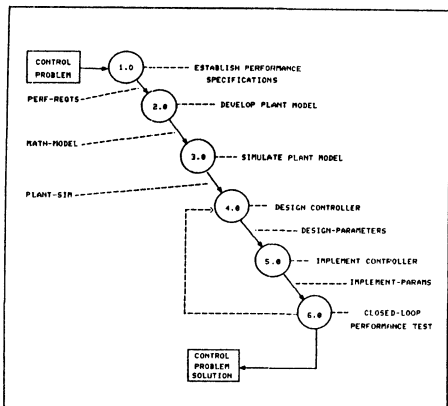


Figure 2. Typical Steps in a Control Problem

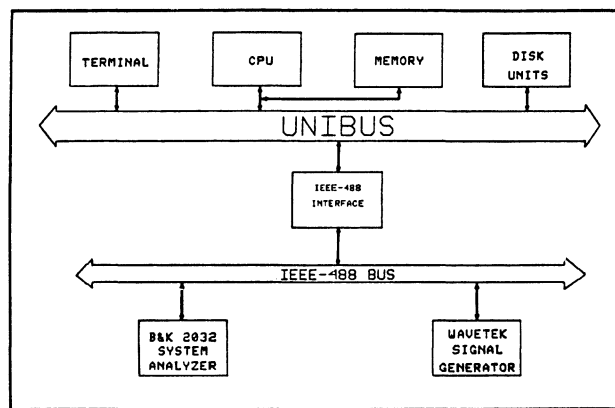


Figure 3. IEEE-488 Software Interfaces.

CONCLUSION

By combining a general purpose computer system with programable instrumentation and flexible interface software, an interactive system for implementing digital controller designs on a state-of-the-art digital signal processing microcomputer has been developed. DICES allows the designer to implement the controller and test it using an analog computer simulation. This assists in determining the effects of finite word-length, quantization errors, filter structure, and other typical effects of implementing a "near infinite precision" design on a finite word-length machine.

REFERENCES

1. Larimer, Stanley J., An Interactive Computer-Aided Design Program for Digital and Continuous Control System Analysis and Synthesis. MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1978.
2. Gembrowski, Charles J., Development of an Interactive Control Engineering Package (ICECAP) for Discrete and Continuous Systems. MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982.
3. Houpis, Constantine H. and Lamont, Gary B., Digital Control Systems Theory, Hardware, Software. New York: McGraw-Hill Book Company, 1985.

4. Texas Instruments, Inc., TMS32010 User's Guide. 1983

5. National Instruments Corporation. GPIB11-2 Operating and Service Manual. Austin TX, July 1982.

6. Bruel and Kjaer. 2032 Instruction Manual, Vols 1, 2 and 3. September 1983.

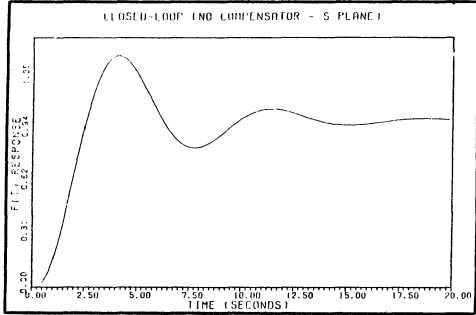


Figure 4.
Theoretical Uncompensated Closed-Loop Response

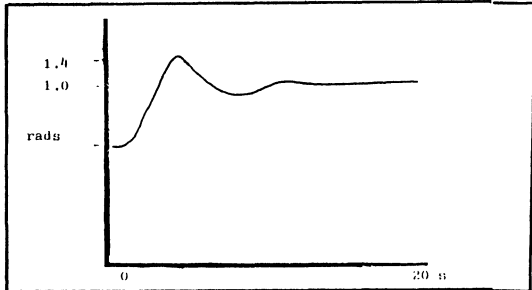


Figure 5.
Actual Uncompensated Closed-Loop Response

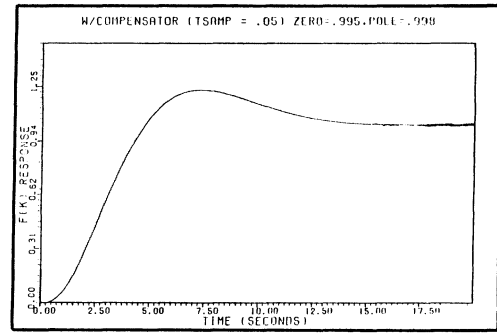


Figure 6.
Theoretical Compensated Closed-Loop Response

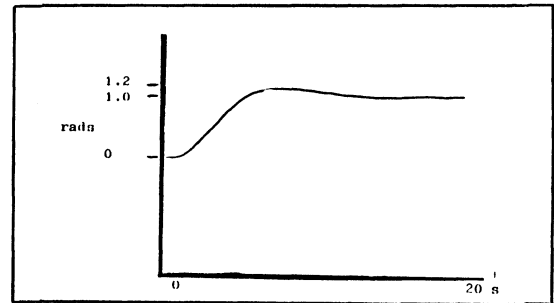


Figure 7.
Actual Compensated Closed-Loop Response

Customizing RS/1 at GA Technologies

ARAM K. KEVORKIAN
GA Technologies Inc.
San Diego, California 92138, U.S.A.

This paper presents an interactive program for temporarily changing the appearance of RS/1 "data objects," and for printing and plotting these data objects in different ways, and on various printers and plotters at different locations, including dedicated devices that may be attached to a terminal. The program is menu-driven and user-friendly and by enabling users to make a variety of choices with simple "yes" or "no" answers to prompts, the task of temporarily changing the appearance of RS/1 data objects and setting up hardcopy output is considerably lightened. The development of this interactive program is a major step toward improving the productivity of computer users at GA through customizing third-party software.

Introduction

RS/1, a software product developed by Bolt Beranek and Newman (BBN) Incorporated, provides capabilities for data management, graphics, electronic spreadsheets, statistical analysis, and applications programming, all integrated in a single easy-to-use system.

A table is the primary means for storing any kind of data in RS/1. Tables are the fundamental data objects in RS/1. They are used for analysis and for generating a variety of other data objects, including bargraphs, piecharts, graphs, three-dimensional plots, models, and procedures.

RS/1 outputs to a large number of printers, plotters, and dedicated devices that may be attached to a terminal. To set up hardcopy output of any RS/1 data object, the system requires the code name of the printer, plotter, or dedicated device, the name of the file to receive the data object, and the appropriate print or plot command. Although all the required information is available in the Information Systems Division (ISD) Users Guide, in the RS/1 manual, and from Systems personnel, the fact that it is to be found piecemeal at different points makes setting up hardcopy output cumbersome and time-consuming for the average user.

In addition, the user may want to choose between horizontal and vertical orientation, regular and pretty printing, or decide between the printers and plotters in the users' and operators' areas.

With the use of modifiers, finally, RS/1 allows temporary changes in the appearance of a data object. For example, the use of the modifier "NOCOLNUMBERS" may

have the effect of displaying a table without column numbers, and the use of the modifier "BOX" will produce box lines around a graph, a bargraph, a piechart, or a three-dimensional plot. Each data object in RS/1 has a different list of modifiers which appears separately in the three-volume RS/1 manual.

All of this know-how from the ISD Users Guide, the RS/1 manual, and the Systems Group has been put into a single menu-driven and user-friendly interactive program. By enabling users to make a variety of choices with simple "yes" or "no" answers to prompts, the task of temporarily changing the appearance of RS/1 data objects and setting up hardcopy output is considerably lightened.

The development of this interactive program is viewed as a major step toward improving the productivity of computer users at GA through the customizing of third-party software.

The Program

The subject of this paper, the interactive program called \$HARDCOPY, is written in RS/1's Research Programming Language (RPL). \$HARDCOPY consists of 14 RPL procedures, including a procedure which can be invoked outside \$HARDCOPY and that provides a glossary of procedures, arguments, variables, and tables used in \$HARDCOPY, as well as a version history. Figure 1, shown below, enumerates the three versions of the program which have been released since May of this year.

Apart from matters of documentation, the most important changes in Version 1.1 and 1.2 had the effect of

shortening the system response time. The latter is defined as the time span¹ between the moment the user enters a command the the moment a complete response is displayed on the screen. This improvement in efficiency was accomplished by eliminating array arguments, and the results have been verified by comparison testing.

VERSION HISTORY

VERSION NUMBER	DESCRIPTION	RELEASE DATE
1.0	Initial version	MAY-85
1.1	Added: Menu of useful hints; glossary of procedures, arguments, variables and tables used in \$HARDCOPY; effective ranges for reducing height and width of graphic data objects for "hardcopy output" and for "display on terminal." Shortened: Response time to some prompts by avoiding use of array arguments in parts of the program.	JUN-85
1.2	Added: \$HARDCOPY version history; capability to exit \$HARDCOPY with original terminal setting; expanded lists of available laser printers, LA50 and LA100 printers column widths, and DEC LP26 lineprinter locations. Included: "Model" in list of data objects to print or plot. Shortened: Response time to most prompts by avoiding use of array arguments throughout the program. Improved: The display of menus.	OCT-85

Figure 1

The testing in this case consisted of a program comprising two RPL procedures, INITIATE and COMPUTE, with the second having five one-dimensional array arguments A, B, C, D, and E, each of length k . The computation performed in COMPUTE involved an evaluation of three conditional branching statements of the type IF ... THEN ... ELSE; one FLOOR and one MIN function evaluations; $2 + 5k$ multiplicative operations (addition, subtraction, multiplication, division); and five ALLOCATE commands. The procedure INITIATE contained the following brief series:

```
X = <FPN>;
TYPE X;
CALL COMPUTE (A,B,C,D,E);
TYPE X;
```

where <FPN> equals any floating point number.

The time span between the two successive displays of the X value is the amount of time, T_1 , needed to call the procedure COMPUTE with the array arguments A, B, C, D, and E. To determine T_1 on a VAX computer, we used an automated system response timer SYSMON which utilizes the built-in clock of an IBM PC to measure response time on the VAX. Table 1, reporting odd-numbered runs, summarizes the results obtained on a VAX-11/785 running under VMS 3.5, for the case where $k = 1$. Table 2, on the other hand, reports even-numbered runs in the same way. Table 2 differs from Table 1 in that array arguments are

excluded in COMPUTE calls. Odd- and even-numbered runs follow each other in order to make comparable pairs.

TIME IN SECONDS (T_1) FOR CALLS OF "COMPUTE" WITH 5 ARRAY ARGUMENTS EACH OF LENGTH = 1

RUN NUMBER (ODD)	DAY 1	DAY 2	DAY 3	DAY 4	DAY 5	DAY 6	DAY 7	OVERALL AVERAGE
1	4.51	7.42	4.11	7.03	4.01	7.96	4.94	
3	4.40	4.61	3.63	5.27	2.91	5.77	3.73	
5	4.62	5.21	4.78	6.20	4.12	7.91	3.57	
7	4.89	5.50	3.79	8.62	3.96	4.56	3.24	
9	4.61	4.56	2.91	9.56	3.79	5.55	4.40	
11	4.62	4.84	3.07	10.16	3.19	6.21	5.66	
13	5.10	3.90	3.18	8.40	3.52	5.55	10.71	
15	4.94	4.34	3.41	8.46	3.52	5.88	4.01	
17	5.00	4.12	2.86	6.42	4.06	4.78	4.56	
19	4.72	4.23	2.86	5.98	3.79	5.55	3.73	
AVERAGE	4.74	4.87	3.46	7.61	3.69	5.97	4.86	5.03

Table 1

TIME IN SECONDS (T_2) FOR CALLS OF "COMPUTE" WITH NULL ARRAY ARGUMENTS

RUN NUMBER (EVEN)	DAY 1	DAY 2	DAY 3	DAY 4	DAY 5	DAY 6	DAY 7	OVERALL AVERAGE
2	0.50	0.77	0.66	0.71	0.55	0.72	1.10	
4	0.49	0.49	0.50	0.55	0.66	0.49	0.55	
6	0.55	0.50	0.60	0.50	0.66	0.50	0.55	
8	0.50	0.49	0.49	1.37	0.55	0.77	0.55	
10	0.50	0.50	0.55	0.55	0.60	0.50	0.55	
12	0.55	0.49	0.49	0.49	0.55	0.49	0.83	
14	0.55	0.50	0.55	0.60	0.55	0.61	0.61	
16	0.55	0.50	0.60	0.55	0.66	0.55	0.66	
18	0.55	0.54	0.50	0.66	0.54	0.49	0.61	
20	0.49	0.55	0.49	0.55	0.60	0.61	0.71	
AVERAGE	0.52	0.53	0.54	0.65	0.59	0.62	0.67	0.59

Table 2

The results show that by eliminating array arguments, an eightfold improvement in average response time is achieved (5.03 versus 0.59 seconds).

Typical Series of Prompts

After entering RS/1, the program is invoked with

```
CALL $HARDCOPY
```

from # prompt. Figure 2 shows a typical series of prompts that follows the invocation of \$HARDCOPY. The first response identifies the current version of the procedure. The response regarding LOCAL HELP in the USEFUL HINTS menu reminds the user that the context-sensitive nature of local help in RS/1 system has been extended into this program.

\$HARDCOPY deals with the seven data objects listed in the PRINT/PLOT menu in different ways as appropriate. The choice of BARGRAPH from the PRINT/PLOT menu in Figure 3a leads to the prompts shown in Figures 3a through 3c.

```
# CALL $HARDCOPY <RET>
*****
$HARDCOPY Version 1.2
*****

Do you wish to see introductory screen or menu of useful
hints? [YES] <RET>
    Introductory screen? [YES] N <RET>
    Menu of useful hints? [YES] <RET>

        USEFUL HINTS
        1 How to EXIT AT ANY TIME
        2 DEFAULT CONVENTION
        3 How to obtain LOCAL HELP
        4 EXIT
        Enter choice: [1] 3 <RET>

            LOCAL HELP

To obtain local ( context-sensitive ) help to any prompt,
enter "?". For example, when you are at the following prompt

    Enter ROWHEIGHT: [2]

and it is not clear to you what ROWHEIGHT means, simply
enter "?" and press <RET> and you will get the following
assistance.

    Expands cell height by number of rows indicated by expression

After this message the same prompt is repeated.

        USEFUL HINTS
        1 How to EXIT AT ANY TIME
        2 DEFAULT CONVENTION
        3 How to obtain LOCAL HELP
        4 EXIT
        Enter choice: [1] 4 <RET>
```

Figure 2

In Figure 3b it may be noted that \$HARDCOPY provides the user with effective ranges for reducing height and width of graphic data objects for "hardcopy output" and for "display on terminal" in anticipation of equipment limitations.

All figures and tables in this paper were created using ISSCO's graphics software package TELLAGRAF, whereas the text portion was processed using the computer typesetting software TeX.

Acknowledgments

The author wishes to acknowledge the invaluable assistance received from Jim Binder and Dave Rapp, both of ISD. To the first the author is indebted for advice on documentation techniques, and to the second he owes much thanks for developing the SYSMON measuring tool for monitoring system response and applying it for this study.

Reference

¹ *The Economic Value of Rapid Response Time*, IBM publication GE20-0752-0 (November 1982).

```
DATA OBJECTS TO PRINT/PLOT
1 TABLE
2 MODEL
3 PROCEDURE
4 BARGRAPH
5 PIECHART
6 GRAPH
7 THREED
8 EXIT
Enter choice: [1] 4 <RET>

Display list of bargraphs? [YES] N <RET>
Enter name of bargraph: ABCEE <RET>
Is your terminal in the following list of high resolution terminals?
    VT125
    GRAPHON
    TEKTRONIX 4010
    TEKTRONIX 4105
    TEKTRONIX 4662
If yes press <RET> else type no (n): [YES] N <RET>

Display on VT100 series (excluding VT125) of bargraphs and
graphs with less than optimal quality is possible. For piecharts
the quality of display will be only marginal, and for threads it
is unusable.

Is your terminal in VT100 series? [YES] <RET>
    Display bargraph? [YES] N <RET>
    Modify default bargraph? [YES] <RET>

        BARGRAPH MODIFIERS THAT CAN BE CHANGED
        1 TITLE or HEADER (Exclude)
        2 HEIGHT or WIDTH (Modify)
        3 BOX (Include)
        4 NOTES (Exclude or Postpone)
        5 BRAGRAPH KEY (Exclude or Postpone)
        6 BARGRAPH KEY (Change Location)
        7 I or D TICKS (Display either one)
        8 I and D TICKS (Display both or neither)
        9 EXIT
        Enter choice: [1] 2 <RET>
```

Figure 3a

Effective range for hardcopy output:
 0.40 <= HEIGHT, WIDTH <= 1
 Effective range to display on terminal:
 0.65 <= HEIGHT, WIDTH <= 1
 Enter HEIGHT: [1] 0.3 <RET>
 HEIGHT may not be less than 0.40. Otherwise program
 will fail to plot or display the bargraph
 Enter HEIGHT: [1] 0.5 <RET>
 Do you intend to display bargraph? [YES] <RET>
 HEIGHT may not be less than 0.65 if you intend to
 display the bargraph
 Enter HEIGHT: [1] 0.7 <RET>
 Enter WIDTH: [1] 0.8 <RET>

BARGRAPH MODIFIERS THAT CAN BE CHANGED

- 1 TITLE or HEADER (Exclude)
- 2 HEIGHT or WIDTH (Modify)
- 3 BOX (Include)
- 4 NOTES (Exclude or Postpone)
- 5 BRAGRAPH KEY (Exclude or Postpone)
- 6 BARGRAPH KEY (Change Location)
- 7 I or D TICKS (Display either one)
- 8 I and D TICKS (Display both or neither)
- 9 EXIT

Enter choice: [1] 9 <RET>

Display modified bargraph? [YES] N <RET>

AVAILABLE OUTPUT DEVICES

- 1 LASER
- 2 PRINTRONIX
- 3 CALCOMP
- 4 DEC LP26
- 5 PRINTER AT TERMINAL

Enter choice: [1] <RET>

Vertical orientation? [YES] N <RET>

Figure 3b

With horizontal orientation, there are two possibilities:

- o Character size: 1/16 inch (regular printing)
- o Character size: 1/8 inch (pretty printing)

Regular printing? [YES] <RET>

AVAILABLE LASER PRINTERS

- 1 In users area
- 2 In operators area
- 3 In 15-125 (for permission call x 3986)
- 4 Display queues

Enter choice: [1] 4 <RET>

QUEUES FOR LASER PRINTERS

- 1 In users area
- 2 In operators area
- 3 In both areas
- 4 Exit

Enter choice: [1] <RET>

Queue for laser printer in users area:

CURR 4121 KEVORKIAN DECUSBAR6 PRI = 4 12-SEP-85 SIZE = 3

AVAILABLE LASER PRINTERS

- 1 In users area
- 2 In operators area
- 3 In 15-125 (for permission call x 3986)
- 4 Display queues

Enter choice: [1] <RET>

Bargraph plotted. Any more bargraphs to plot? [YES] N <RET>

DATA OBJECTS TO PRINT/PLOT

- 1 TABLE
- 2 MODEL
- 3 PROCEDURE
- 4 BARGRAPH
- 5 PIECHART
- 6 GRAPH
- 7 THREED
- 8 EXIT

Enter choice: [1] 8 <RET>

Figure 3c

PRO: A Multiple Priority, Multitasking Process Control System and Language
as Implemented in an Inhalation Exposure Facility

Edwin R. Lappi and Leon C. Walsh, III
Inhalation Exposure Group
Northrop Services, Inc. – Environmental Sciences
Research Triangle Park, NC

ABSTRACT

A number of vendors offer process control systems and software for the manufacturing environment. PRO is one such system and language that runs on an LSI 11/23 CPU. This paper discusses PRO's implementation in a small animal inhalation exposure laboratory as the control system for pollutant concentration profiles in environmental exposure chambers. In particular, this paper addresses PRO's ability to control the generation of smoke obscurants for a study evaluating the potential human health hazards of these obscurants. Also, the acquisition of chamber environment data during the exposure is described.

INTRODUCTION

Northrop Services, Inc. – Environmental Sciences (NSI-ES) conducts animal inhalation exposure testing under contract with the U.S. Environmental Protection Agency. One current project is the whole-body inhalation exposure of small rodents to an aerosol generated from 10-weight motor oil. The oil aerosol can be field generated at sufficient concentrations to be an effective smoke obscurant for troop movements or other military activities. Evaluation of the health effects of repeated acute and subchronic exposure to this aerosol, which might occur during troop training exercises, is the overall goal of the project.

Operation of a facility for scientific testing involving animals is, at best, an extremely complicated undertaking. Critical considerations include the environmental control systems, animal housing and care, specific research protocols, and the actual testing procedures to be implemented. This paper addresses the operation of the exposure system, specifically the monitoring and control of pollutant levels during an animal inhalation test.

The aerosol particles are generated from the bulk oil using a vaporization/condensation technique that produces particles approximately 1.0 to 1.3 μm in diameter. The concentration of the aerosol in the exposure chamber is monitored with a real-time aerosol monitor (RAM-1, GCA Corp., Bedford, MA) equipped with a diluter to allow high concentrations to be effectively sampled. Since a complete regimen of toxicological tests must be conducted on a population of animals large enough to yield statistically significant results, the exposure facility consists of multiple chambers, each equipped to generate and monitor the oil aerosol at individually controllable levels.

Operation of this exposure system requires that all critical parameters be continuously monitored and controlled within limits established by the study protocol. Items to be monitored include oil aerosol concentration within the chamber; aerosol particle size; chamber air flow, temperature, and negative pressure with respect to the laboratory; temperature of the two heaters contained in the generator; three separate temperature limit sensors; the inlet air conditioning unit; and the chamber exhaust system. As shown here, this is an extremely complex system – hence the rationale for utilizing automated control wherever possible to ensure the concise, repetitious exposures required for long-term research projects.

EXPOSURE FACILITY

A special laboratory facility was constructed for the project. The exposure facility consists of an exposure control laboratory, exposure room, necropsy room, shower room, and three total-exhaust animal rooms [1,2]. Only the first two rooms, the exposure control laboratory and the exposure room, are of concern in this paper. The room layouts are shown in Figures 1, 2, and 3. A barrier separates the exposure control laboratory from

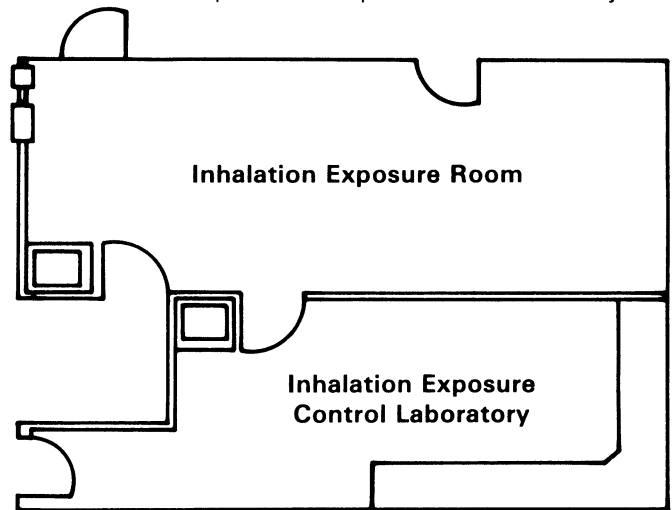


Figure 1. Exposure Facility Floor Plan.

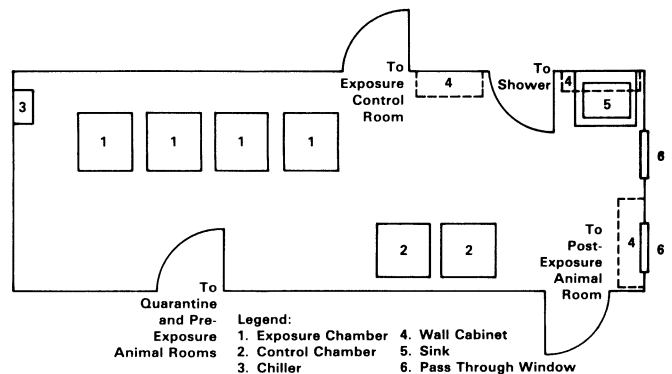
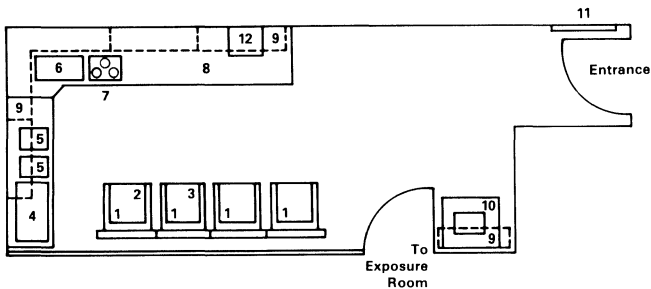


Figure 2. Exposure Laboratory.



- Legend:
- | | |
|---------------------------------------|--|
| 1. Exposure Control Panel | 7. High Pressure Liquid Chromatograph Terminal |
| 2. Chiller Control Panel | 8. Casework |
| 3. Recorder | 9. Wall Cabinet |
| 4. Gas Chromatograph | 10. Sink |
| 5. Gas Chromatograph Terminal | 11. Pressure Monitoring System |
| 6. High Pressure Liquid Chromatograph | 12. Instrument Parts Storage |

Figure 3. Exposure Control Laboratory.

the exposure room. All air lines and electrical connections for the exposure monitors and control devices pass through sealed bulkheads attached to the barrier wall. The top half of the barrier is constructed of plate glass windows to allow visual inspection of the exposure room from the exposure control laboratory and vice versa.

The exposure room contains six stainless-steel Rochester chambers for whole-body exposure of small animals. Four chambers are used for pollutant exposures, and two chambers are used for control animals. The system's design, even prior to automation, allows the use of any one or any combination of the exposure chambers simultaneously; this provides a high degree of flexibility in chamber operations. The exposure control laboratory contains the exposure control panel, scientific instrumentation, and the control computer.

SYSTEM OVERVIEW

The basic requirements for the control and monitoring system follow:

- on-line interactive, multiple priority, and multitasking operation;
- ease of use by the scientist involved with the exposures; and
- modular construction.

For this application, availability for quick implementation was also necessary, as time constraints were rather severe. Writing the tasks by using machine language subroutines for data acquisition and control tasks and by using FORTRAN for the other tasks operating under RSX or even RT-11 would have been a possibility, except for these time constraints. These constraints did not allow for adequate design and testing of user-written software, even though the necessary expertise was available.

Since most of the control and monitoring requirements fall under the domain of process control-type operation, a search was conducted for a suitable process control operating system and language available for use with readily obtainable, reliable hardware.

SYSTEM DESCRIPTION

The process control unit, named PROSYS I, is fabricated and assembled by the Adaptive Data Acquisition and Control Corporation (ADAC, Woburn, MA) using a DEC LSI 11/23 CPU with

256 K bytes of memory, which is custom designed for industrial process control. Peripheral devices include a printer (DEC LA-120), a terminal monitor (Lear Seagler ADM-5A), a dual floppy disk system (DEC RX02), and a ceiling-mounted large video monitor (19-in. Sony Trinitron Model CVM1900) (see Figures 4 and 5). The computer is interfaced to monitor the following system functions: chamber temperature, Vycor heater temperature, heat tape temperature, chamber negative pressure, generator nitrogen flow, and RAM-1 concentration sensing for each chamber. The computer system is interfaced to control exhaust blower speed, temperature on/off relays, and oil flow to the generator (see Figure 4).

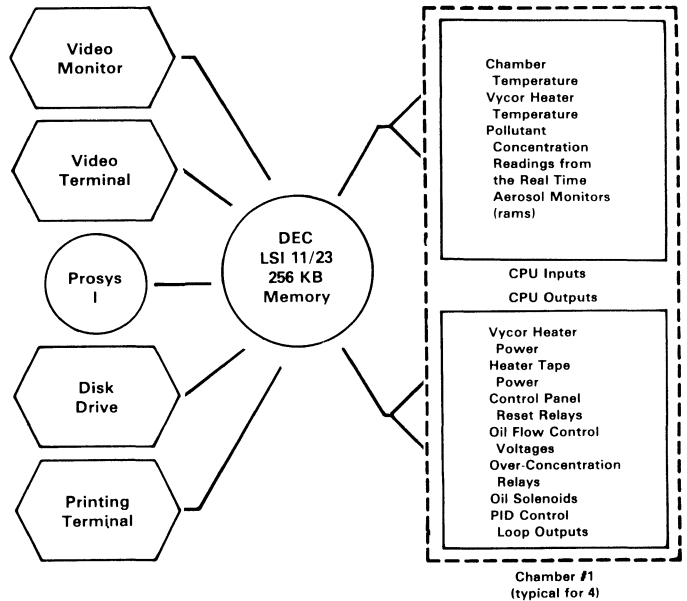


Figure 4. Automated Control System Configuration.

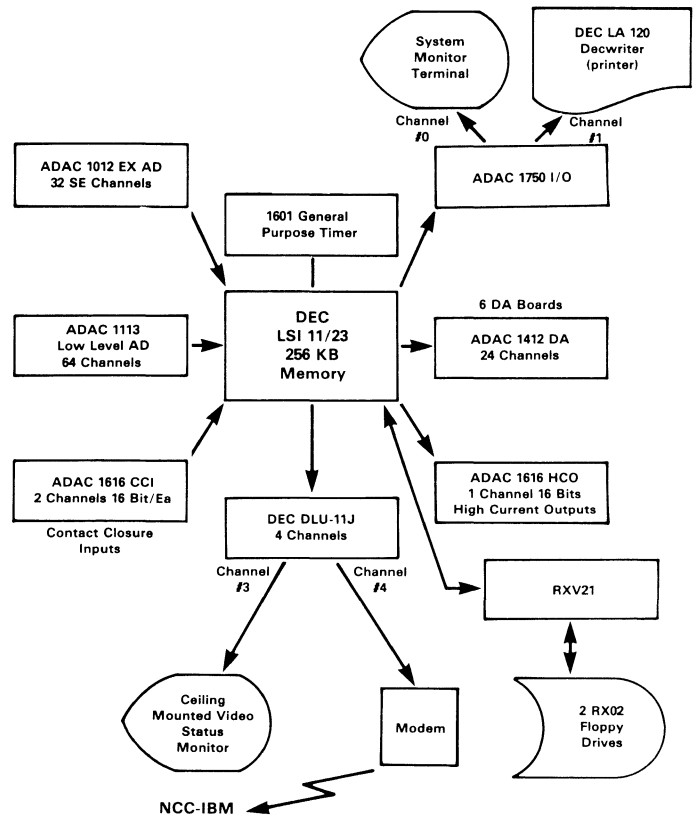


Figure 5. Hardware Configuration.

PRO (developed by Staff Computer Technology Corp., San Diego, CA) is a memory resident, multiple priority, multitasking process control and monitoring software package. PRO can handle up to 256 tasks concurrently. User statements may be entered from the system terminal or from off-line storage media. Statements may be added, listed, deleted, and changed while other activated tasks (programs) are executing. Four main elements: compiler, operating system, run time program, and I/O drivers, provide everything that is necessary to accept user statements written in PRO.

Compiler

The compiler is an interactive incremental compiler that examines each program statement as it is entered for syntax and sequence errors. It then either accepts and compiles the statement or immediately notifies the programmer of an error by a diagnostic error message that requests correction of the statement. This interactive attribute makes PRO easy to learn, program, and debug because the programmer is immediately made aware of an error. Additional checks are made after all statements have been incrementally compiled to further reduce the possibility of program errors. A full discussion of the advantages of PRO's compiler and a comparison of it with the batch compiler and interactive interpreter can be found in an article by Benton [3]. (Also, see Figures 6, 7, and 8.)

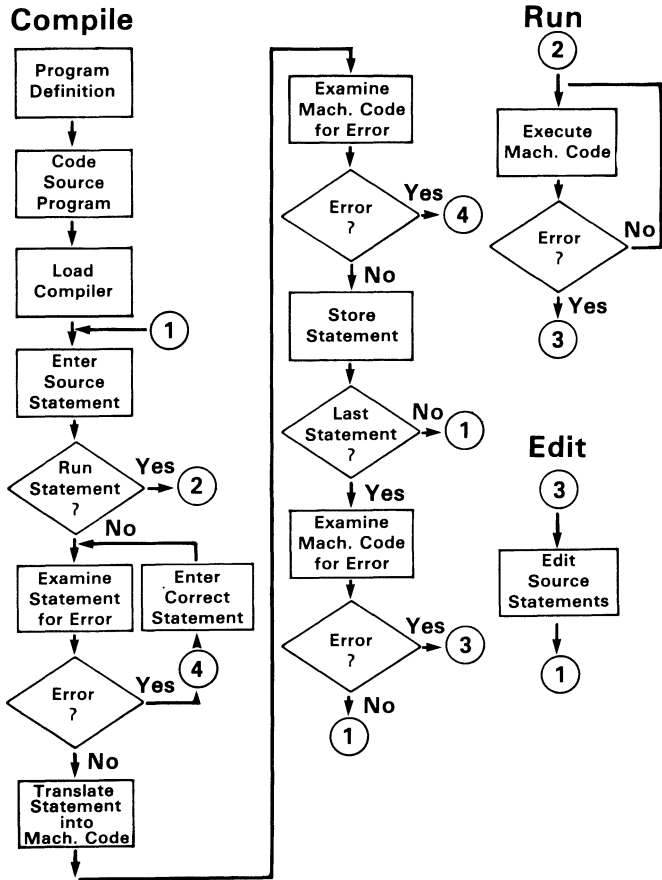


Figure 6. Interactive Incremental Compiler.

Compile

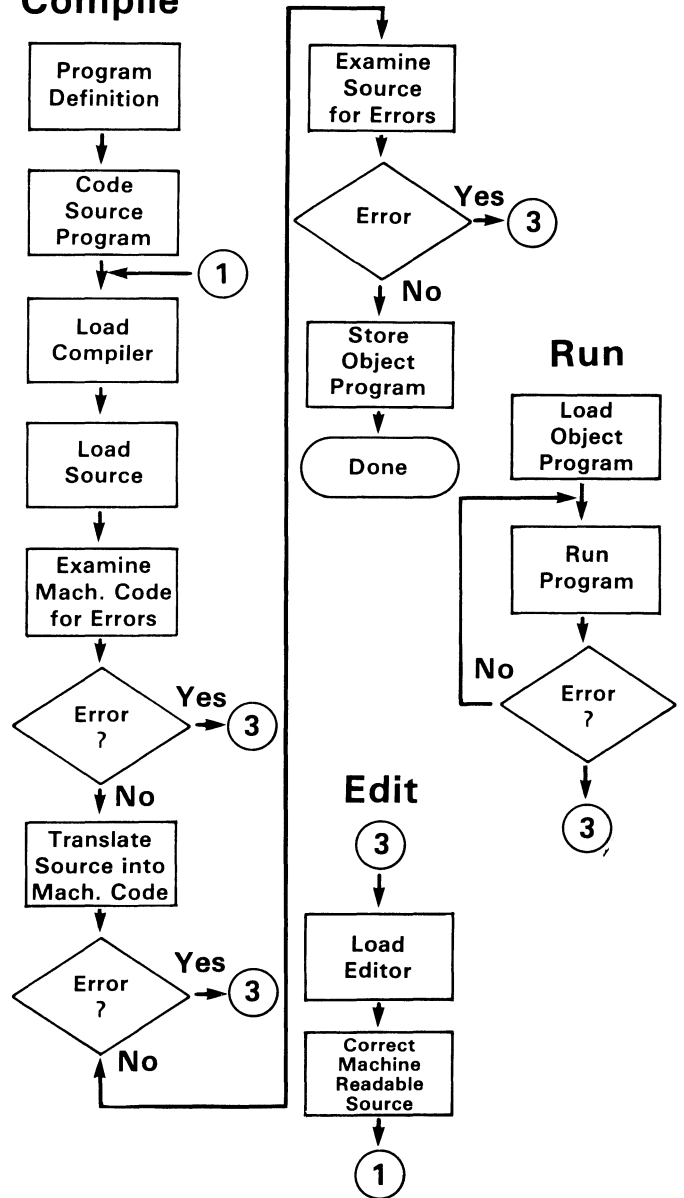


Figure 7. Batch Compiler.

Operating System

PRO provides a complete stand-alone operating system for scheduling, setting priorities, allocating system resources, implementing general "housekeeping" routines, and monitoring the operation of the system.

PRO operates as a multitasking system, i.e., it is capable of handling numerous tasks concurrently. User-assigned priorities control the scheduling of these tasks. The operating system controls execution by suspending low priority tasks until high priority tasks have been executed or suspended.

Run Time System

PRO's run time system is the collection of programs required to execute both the system and user tasks. Run time includes system initialization, processing of alarms, and detection and interpretation of errors in the executing task.

Compile

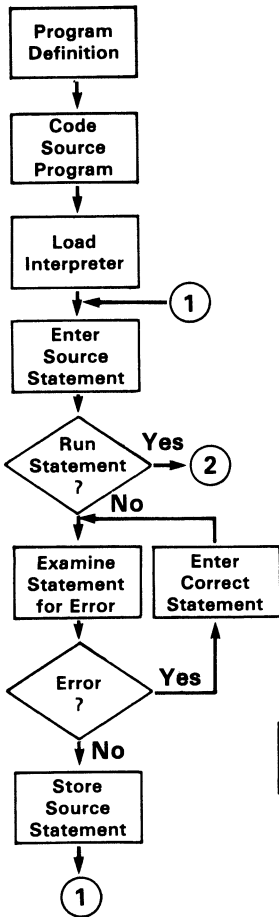
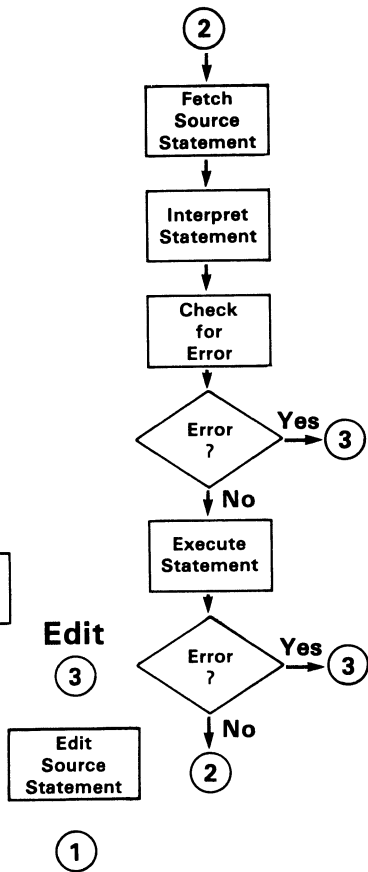


Figure 8. Interactive Interpreter.

Run



Once PRO is booted up, the user begins by defining the digital outputs, analog outputs, digital inputs, analog inputs, and automatic control loops (i.e., PID-Proportional, Integral, and Derivative Control). Figures 9 through 13 illustrate each of these definitions.

```

DEF:          DOUT:BITS;
CHAN:66;
1*           BITOUT:VIC(0);
2*           BITOUT:HTR(1);
3*           BITOUT:RST(2);
4*           BITOUT:VIC2(3);
5*           BITOUT:HTR2(4);
6*           BITOUT:RST2(5);
7*           BITOUT:VIC3(6);
8*           BITOUT:HTR3(7);
9*           BITOUT:RST3(8);
10*          BITOUT:VIC4(9);
11*          BITOUT:HTR4(10);
12*          BITOUT:RST4(11);
/;
    
```

Figure 9. Digital Outputs.

PRO provides the ability to modify or fine tune tasks on line while all other unrelated tasks are still being executed on the same system. The operator may examine any input, change any output variable, or completely activate or deactivate any task or I/O in real time.

The PRO editor enables the computer to list programs in their alphanumeric source formats, even though these source statements are not stored. The editor allows additions and corrections to be made by using standard PRO statements.

I/O Drivers

PRO provides all necessary software drivers to control the operation of both communications and process I/O devices. These include

- 2048 discrete inputs or outputs,
- 256 analog outputs,
- 1024 analog inputs, and
- 8 serial I/O ports.

Figures 4 and 5 show the automated control system and hardware configurations.

```

DEF:          AOUT:OFLW1;
CHAN:133;
/;
SEL:OPMP1;
LIST(1);
DEF:          AOUT:OPMP1;
CHAN:130;
/;
SEL:OSOL1;
LIST(1);
DEF:          AOUT:OSOL1;
CHAN:129;
/;
SEL:OCNC1;
LIST(1);
DEF:          AOUT:OCNC1;
CHAN:132;
/;
    
```

Figure 10. Analog Outputs.

```

DEF:      AIN:VC1;
PER:5;
CHAN:8;
CONV:LIN;
IRG:0,41,25;
ERG:20.000,1000.0:DEGC;
/;

```

Figure 11. Digital Inputs.

```

DEF:      AIN:RAM1;
PER:5;
CHAN:512;
CONV:LIN;
IRG:0,9997;
ERG:0 E3,177.60:MGPERM3;
1*      FILT;
FILTV:1,0000;
2*      ALM;
HIAL:210.0;
LOAL:0 E3;
DB:0 E3;
3*      PID:RAM1PID;
HSPLM:201.00;
LSPLM:0 E3;
DB:0 E3;
DEVL:201.00;
PGAIN:RAM1GAIN;
SETPT:DESCON1;
OUTMX:105;
OUTMN:0;
OUTPT:OFLW1;
IGAIN:.03808;
DGAIN:0 E3;
/;

```

Figure 12. Analog Inputs and Control Loop.

The programmer can then begin writing applications tasks – the sequencing and management logic created by the user to direct the management and control of the process. Figure 13 illustrates a portion of the main control task as used at NSI-ES.

```

DEF:      TASK:CONTROLEXP(1);
1*      STRING:R(1);
2*      ARRAY:CN1(48);
3*      ARRAY:CN2(48);
4*      ARRAY:CN3(48);
5*      ARRAY:CN4(48);
6*      ARRAY:CT1(48);
7*      ARRAY:CT2(48);
8*      ARRAY:CT3(48);
9*      ARRAY:CT4(48);
10*     ARRAY:OV1(48);
11*     ARRAY:OV2(48);
12*     ARRAY:OV3(48);
13*     ARRAY:OV4(48);
:
43*     READ:(1)R;
44*     IF(CHAR(1,R) = 89)GOTO:10;
45*     LET:EX = EX + 1;
46*     GOTO:15;
47*     10: LET:N1A = 1;
48*     15: PRINT:(1)"WILL CHAMBER TWO(2) BE USED TODAY (Y/N)";
:
65*     40: LET:N4A = 1;
66*     GOTO:50;
67*     45: IF(EX = 4)GOTO:70;
68*     50: ACT:INIT;
69*     WAITUNTIL(IN = 99);
70*     DEACT:INIT;
71*     55: IF(SEC < SSC)GOTO:60;
72*     IF(MIN < SMN)GOTO:60;
73*     IF(HOUR < SHR)GOTO:60;
74*     GOTO:65;
75*     60: WAIT:10;
76*     GOTO:55;
77*     65: ACT:BITS;
78*     WAIT:1;
79*     ACT:BLWRS;
80*     WAITUNTIL(BL = 99);
81*     DEACT:BLWRS;
82*     WAIT:1;
83*     ACT:HTRS;
84*     WAITUNTIL(HT = 99);
85*     DEACT:HTRS;
86*     WAIT:1;
87*     ACT:OILFLW;
88*     WAITUNTIL(OL = 99);
89*     DEACT:OILFLW;

```

Figure 13. Portion of Main Control Task.

PRO allows modular construction of the minor and major tasks; this enables the user to break overall tasks into manageable units for ease of debugging and comprehension.

At NSI-ES, the tasks are written so that one main task activates one subtask at a time during the exposure. The tasks are as follows:

- CONTROLEXP – main task that activates and deactivates all subtasks during the exposure. After activating each subtask, CONTROLEXP waits for the subtask flag to be set, which signals CONTROLEXP to deactivate the subtask. CONTROLEXP initializes all storage arrays to zero before any data are stored and requests input from the operator as to which chamber(s) are to be used (see Figure 14).
- INIT – subtask that requests that the operator input the exposure start and stop times. It activates subtask ECHO, which lists the active and inactive chambers as a check for the operator.
- ECHO – subtask that prints a summary of chambers activated and the set points being used during the current exposure (see Figure 15).
- BITS – subtask that activates the high current outputs (HCOs) so that the heaters for all generators can be turned on and off as required.
- BLWRS – subtask that checks for active chambers and whether or not chambers are to be controlled, activates

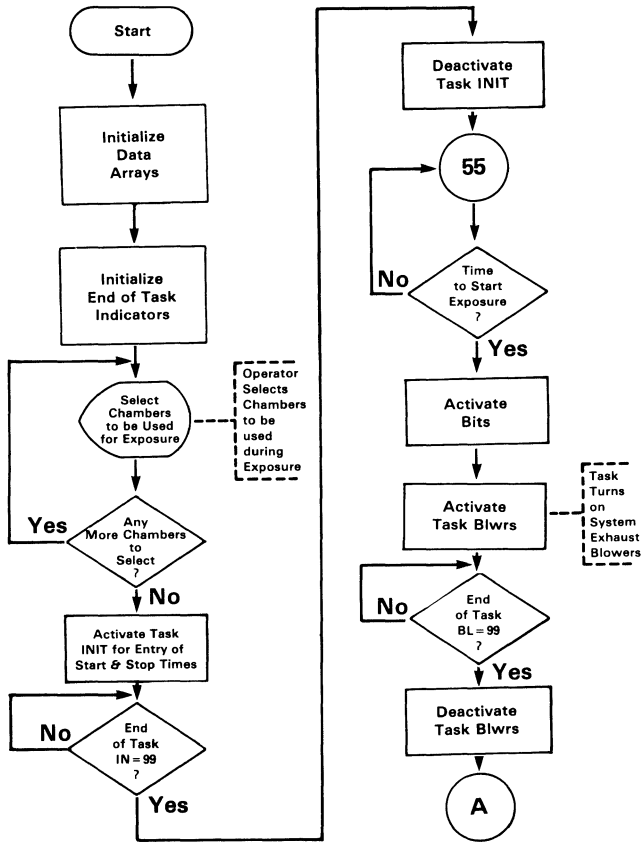


Figure 14. Control Task.

CURRENT TIME IS 8:50:11
 EXPOSURE START TIME IS 8:45:0
 EXPOSURE STOP TIME IS 9:45:0

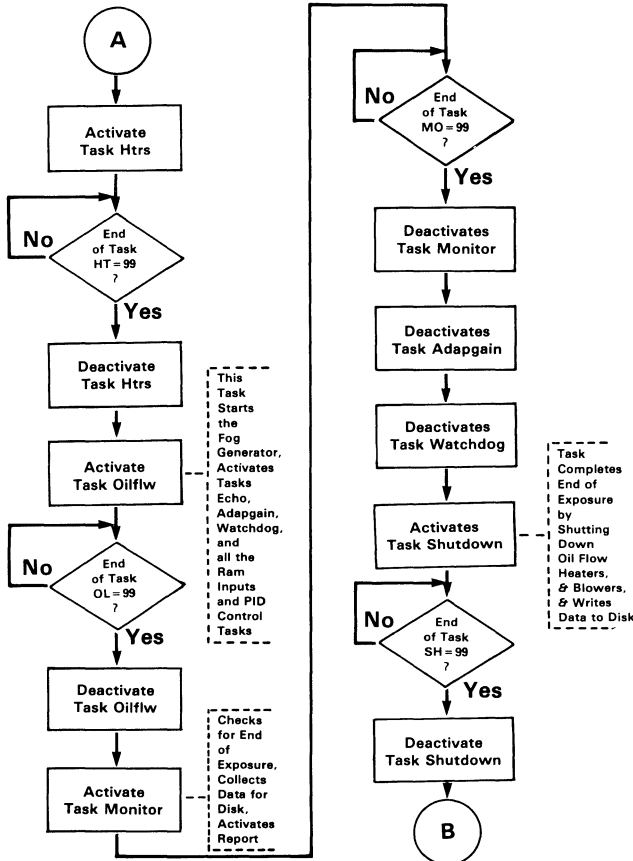
CHAMBERS ON CONTROL FOLLOW; 0 = NO CONTROL, 1 = CONTROLLED
 CHAMBER 1 1
 CHAMBER 2 1
 CHAMBER 3 1
 CHAMBER 4 1

RAM SETPOINTS FOR CHAMBERS FOLLOW
 CHAMBER 1 70.00
 CHAMBER 2 61.00
 CHAMBER 3 148.00
 CHAMBER 4 52.00

Figure 15. Control Variables ECHO.

the associated analog outputs and resets the overconcentration relays, makes sure heaters are off, and activates analog inputs to read chamber temperatures.

- HTRS – subtask that checks for active chambers, turns on associated heaters, waits for Vycor heater temperatures to reach 500°C, turns off Vycor heaters for 30 s and then turns them back on for the duration of the exposure, and turns off all resets and deactivates the analog inputs to read Vycor heater temperatures.
- OILFLW – subtask that initializes oil solenoids to zero, activates the analog output to the oil solenoid for an active chamber, activates the analog output to the oil pump, activates analog inputs to read the RAMs, activates the PID control loops, and requests entry of the desired set points for activated chambers. It activates ECHO to list active chambers and set points being used and activates subtask ADAPGAIN.
- ADAPGAIN – abbreviation for ADAPTIVE GAIN – subtask that is used primarily to slow the PID’s response during chamber start-up, when the error is large, and to speed the PID’s response when errors occur once the set point has been reached. It is a commonly used method in process control for varying the proportional gain, K_C , in the PID equation to vary the speed of the response for different degrees of error, E .
- WATCHDOG – subtask that contains oil flow checking and limiting. Every five seconds the subtask checks each chamber’s oil flow and compares it to the high oil flow limit for the chamber; it does not allow the oil flow to



go above the limiting value. No alarm bells or messages are given when the oil flow limit is reached.

- MONITOR – subtask that activates all chamber temperature analog inputs, activates subtasks REPORT and VDISPLAY, and checks every 10s to see if the end time for the exposure has been reached yet. If the end time for the exposure has been reached, MONITOR deactivates REPORT and VDISPLAY; saves all final readings of the RAMs, chamber temperatures, and oil flow control voltages in their respective arrays; and prints a final report of all active chambers on the line printer (see Figure 16).

TIME = 9:0:7

THE OUTPUT OF RAM1 = 47.88
THE TEMP IN CHAMBER ONE = - 99.00 DEG F
THE OIL FLOW VOLTAGE FOR CHAMBER 1 = 67 MVOLTS

ALL CHAMBERS SHUTDOWN AT: 9:0:45

Figure 16. Final Report for all Active Chambers.

- REPORT – subtask that makes sure VDISPLAY is activated, saves data into arrays every 5 min, and prints a 5-min report of data on the line printer.
- VDISPLAY – subtask that displays 5-min status data on all active chambers on a ceiling-mounted video display.
- SHUTDOWN – subtask that deactivates oil flow, the PID control loop, and analog inputs and displays a message on the console that the chamber oil flows have been shut down. SHUTDOWN activates STOREARRAYS and Vycor heater analog inputs, turns Vycor heaters and heat tape heaters off, waits until Vycor heater temperatures have cooled to below 200°C, turns off overconcentration relays, prints a message on the console indicating that the chambers have been shut down, and deactivates Vycor heater analog and temperature analog inputs.
- STOREARRAYS – subtask that sets up block addresses on disk for each array and writes data from an active chamber onto disk.

DATA

Chamber environmental data collected during each exposure include the following: the output readings of the RAMs, chamber temperature, and oil flow control voltage to the oil pumps (see Figure 17).

CONCLUSIONS

One of the limitations of PRO is that it uses block addresses to write data onto disk. This means that it writes an array of data, even a small one, onto one block of the disk. The next array must then be written onto the next contiguous block on the disk. In addition, PRO's format is not readable by either RT-11 or RSX; therefore, these data disks cannot be interchanged with one of the standard DEC systems. However, this lack of compatibility with one of the standard DEC operating systems was easily overcome and proved to be only a minor nuisance to NSI-ES during this project.

AT TIME = 8:55:35

THE OUTPUT OF RAM1 = 12.406
THE TEMP IN CHAMBER ONE = - 99.00 DEG F
THE OIL FLOW VOLTAGE FOR CHAMBER 1 = 48 MVOLTS
THE OUTPUT OF RAM2 = 7.383
THE TEMP IN CHAMBER TWO = - 99.00 DEG F
THE OIL FLOW VOLTAGE FOR CHAMBER 2 = 53 MVOLTS
THE OUTPUT OF RAM3 = 2.609
THE TEMP IN CHAMBER THREE = - 99.00 DEG F
THE OIL FLOW VOLTAGE FOR CHAMBER 3 = 150 MVOLTS
THE OUTPUT OF RAM4 = E3
THE TEMP IN CHAMBER FOUR = 69.04 DEG F
THE OIL FLOW VOLTAGE FOR CHAMBER 4 = 53 MVOLTS

Figure 17. Five-Minute Report for all Active Chambers.

Overall, NSI-ES was extremely pleased with the ease with which PRO was implemented and learned by the staff. PRO has been proven to be a very powerful and flexible system that runs on the reliable DEC LSI 11/23 computer.

REFERENCES

1. Northrop Services, Inc. – Environmental Sciences. 1983. *Facility Description: Oil Aerosol Inhalation Exposure Facility*. Research Triangle Park, NC: Northrop Services, Inc. – Environmental Sciences.
2. Davies, D.W. 1984. *Inhalation Toxicology of Fog Oil Obscurant, Phase I: Inhalation Exposure Facility*. Research Triangle Park, NC: Toxicology Branch, Inhalation Toxicology Division, Health Effects Research Laboratory, U.S. Environmental Protection Agency.
3. Benton, L.A. 1978. *Interactive Compiler Translates Step-by-Step and Stores Translation*. Reprint from Control Engineering.



Expert System Usage in the Laboratory

Thomas A. Turano
Digital Equipment Corporation
Marlboro, Massachusetts

ABSTRACT

Although expert systems are becoming more prevalent, their application as an aid in the laboratory has been minimal. LDP is investigating the use of expert systems to aid the scientist both in configuring real-time data collection systems and in statistically analyzing the data collected. These expert systems would use sets of rules enumerated by experts in various fields to provide consultation to the experimenter.

The purpose of a real-time system configurer is to help the experimenter select the system components necessary to collect the required data. This interactive expert system, similar to XCON the system currently used within DIGITAL to configure VAXen, would ask questions concerning the nature of the experiment and the type of computer system being used. From this data, the expert system would determine what combination of components is available for such a data collection system.

Similarly, the statistics expert system would suggest to the experimenter which statistical methods are the most suitable for the analysis of the data as determined by the nature of the data and the hypothesis presented.

This talk will describe the current results of the ongoing investigation into the feasibility of these systems.

1 INTRODUCTION

This paper concerns a use of a branch of artificial intelligence (AI) in the laboratory. It is not concerned with the research side of AI which attempts to determine what human intelligence is and emulate its functions. Instead, this paper will address one application of AI technology, the Expert System.

2 EXPERT SYSTEMS

2.1 The Approach

In recent years, a subspeciality of AI called expert systems has developed. An expert system is defined as a program which goes about solving a problem in a manner that an expert would use. An expert does several things in approaching a problem (11).

First, the expert quickly discards avenues of inquiry that he believes will be unsuccessful. For example, if you go to the doctor complaining of pain in your arm, the doctor will not simply recall all she knows about anatomy. She immediately discards all the knowledge she has that is unconnected to what might cause a pain in the arm. In this case, there is no reason to think about toes, so any toe knowledge is not examined. Since a heart attack may cause a radiating pain in the arm, the knowledge base she has

concerning the heart is examined until she is confident that the heart is not a cause of the pain. Once heart problems have been ruled out, she ceases to use the knowledge she has about the heart. By approaching a problem in this manner, the expert moves through a huge amount of data very quickly. Had each piece of information known by the expert been examined, the total time required for a search would be prohibitively long.

Another thing an expert does not immediately do is attack the problem from fundamental principles. Going back to the example: the doctor looking for the cause of pain in the arm does not immediately begin by thinking about the conduction of nerve impulses in terms of membrane potentials. That approach would be too fundamental and, even if the cause and effect relationship between the membrane potential and the perception of pain were completely understood, getting to a solution of the problem at hand from this high level of detail would again take too long. In reality, many problems are incompletely understood, so that it would not be possible to arrive at any solutions from fundamental principles.

Instead of basic principles, the expert uses rules of thumb or heuristics to isolate the problem. The use of rules of thumb is termed shallow knowledge. You do not have

to know about Maxwell's equations of electricity to use a light switch. Experts generally have both shallow and deep knowledge about the problem area (domain). Expert systems have shallow knowledge only, and still satisfactorily solve problems within their domain.

The question is then: why bother with deep knowledge if shallow knowledge will do? The answer is that the performance of something or someone with deep knowledge will degrade more gracefully than someone or something with shallow knowledge as the limits of the domain knowledge are reached. Going back to the example again; assume you, having a pain in the arm, go to a knee expert instead of a general practitioner. The knee expert might not know which muscles or nerves are in the arm; but, having deep knowledge he can analogize using the commonality of properties he knows exists with the knee. A knee expert system having shallow knowledge would simply not function when this arm domain boundary is reached.

2.2 The Justification

The justification of the effort in building an expert system comes from several considerations. First, there is so little natural intelligence to go around. This is not meant to be derogatory. Just consider what an expert is. Here is a person who has

spent considerable study and acquired considerable experience. Generally, true experts are scarce and therefore expensive.

Second, the use of an expert may be inconvenient. For example, an expert oil drilling geologist's idea of a great time might not be to spend six months in the arctic tundra advising the drillers, yet it might be necessary to have such a person on station.

Third, the expertise within an organization is ephemeral. Companies spend a great deal of money and effort generating experience. So, after someone is trained and is an expert, what can the company expect? The person eventually retires, dies, or quits. This fundamental truth is the reason behind the apprentice system. It would be far more efficient if the knowledge base kept increasing instead of being rebuilt each time an expert leaves the organization.

Finally, the expert system is immortal, and for the egotists among us, so is the expert knowledge incorporated into it. By analogy, so then is the expert whose knowledge has been incorporated.

2.3 The Criteria

The question then arises, why are there

relatively few expert systems? The answer is that not all problems are amenable to solution by expert systems. There is a small set of criteria by which the success of an expert system approach can be estimated. The first criterion is that the problem must be of the correct size. What is the correct size? The correct size is one which is small enough to have a reasonable amount of code and large enough to be worth doing. The proper size has been described as the type of problem you could attempt to have an expert solve by describing the problem to him over the phone. This limited bandwidth problem description is perfect for expert systems.

The second criterion is that there be a consensus of opinion by experts. If the experts can not agree on the solution, then one expert's view will have to be chosen, and the results will be biased toward this view. This is generally not satisfactory.

On the other hand, there has been some discussion of using expert systems where there are NO experts. The idea follows the scientific principle that if no one knows the solution, anyone's guess is as good as anyone else's. One then programs the expert system with the best guesses, and from there determines the result. If the result is satisfactory, the system is left alone. If the result is unsatisfactory, a portion of the system is changed, and the result of the

change on the outcome is examined. This can be considered to be equivalent to building a prototype expert. The problem with this approach is that it is time consuming and not guaranteed to converge on a correct answer. Each rule has to be varied separately and in combination, and it is not guaranteed even then that there is not simply a rule missing.

Finally, the problem must require specialized rather than generalized knowledge. This frequently poses a problem since some general knowledge is always assumed by an expert. Consider an expert system that does some form of medical diagnosis. Assume the expert approaches the problem that if the patient is pregnant and has high blood pressure, then there may be toxemia. The expert would automatically rule out toxemia of pregnancy as a possibility if the patient is male. The fact that males do not become pregnant is general knowledge and would be the type of knowledge that might be missing from a medical diagnostician program. As a result the program would be forced to ask "Is the patient pregnant?" even if the patient is a male.

2.4 Some History

The earliest expert systems developed were for use by the scientific community.

The first such system was DENDRAL, which interpreted mass spectrographic data to determine molecular structure. Construction of this system was begun at Stanford in 1965 by Buchanan, Mitchell, and Feigenbaum (6). Many others have contributed to it over its 20 years of existence. In this time period DENDRAL has been constantly learning and it is a good example of retaining knowledge which would otherwise be lost. In addition, because DENDRAL has been examining spectra for so long a time, it has become very good at molecular analysis. Some claim that it now better than any analytical chemist.

Medical expert systems came into being with the beginning construction in 1976 of MYCIN, an expert system which diagnoses infectious blood diseases (14). Dr. Shortliffe of Stanford, realizing that the heuristics of medical diagnosis could be placed into an expert system, created MYCIN. MYCIN's success in turn was the inspiration for many other medical diagnostic programs.

The first commercially sold system was developed in 1979 at the Stanford Research Institute by Duda, Gaschnig and Hart. This system examined geological data to locate mineral deposits suitable for mining. The system showed its worth by discovering a previously unknown molybdenum deposit (14).

Another commercially successful tool is XCON. This system started as R1 at Carnegie

Mellon in 1980. McDermott produced this program to incorporate the expertise of technical editors who configure VAX computers for the DIGITAL Equipment Corporation. A complex computer system previously was assigned to a technical editor who verifies all the components ordered are present and correct. This was a tedious task that has been greatly improved by the use of an expert system. DIGITAL has saved millions of dollars with this program.

Several laboratories have experimented with expert systems to help people use statistics (1,3,4,8,9,12). These programs help a user select the correct statistical tests to analyse his data. One such system operating at the U.S. National Institutes of Health helped clinicians and researchers with t-statistics and regression analysis. In its first year of operation it was used by 100 different people approximately 12,000 times (12).

Finally, just for general information, expert systems have not been restricted to the scientific field. Systems have been developed that attempt to teach marketing strategy, understand tax law (2) and compose newspapers. It should be apparent that expert systems will be giving advice in the future to people in both technical and social areas.

2.5 The Implementation

Expert systems fall into two major categories, rule based and frame based. A rule-based system simply takes the rules used by an expert and puts them into a machine format (5,10). The general form of the rule is:

```
IF condition THEN action
```

So, if you were building an expert system to tell you when to use your umbrella, the first attempt at the rule might look like:

```
IF
[raining]
THEN
[use umbrella]
```

This first attempt at the rule is too general. If this rule were followed strictly, the system would tell you to use an umbrella if it were raining even if you had no intention of leaving your living room. To correct this oversight, you might try:

```
IF
[raining and you are going outside]
THEN
[use umbrella]
```

This is better, but, according to this rule, if you going for a ride in your car, you'd have to use the umbrella. The rule would have to be made more specific still by putting in as part of the conditions, something about walking outside:

```
IF
[raining and you are walking outside]
THEN
[use umbrella]
```

The right hand side of the rule is also used to modify the set of data against which the rule is compared. In the example then, assume that you could control the weather. The rule might then be modified:

```
IF
[raining and you are walking outside]
THEN
[use umbrella and make it stop raining]
```

With this additional action, once the rule was invoked (fired), the "MAKE IT STOP RAINING" clause of the rule would modify the dataset so that, on the next cycle, it would not be raining, and this rule would not be invoked again.

The second approach to an expert system is the frame-based approach(7). Some AI people favor this because it groups things together that seem to belong together. Using the same example, the first frame

would be a procedure frame explaining what is to be done under which circumstances.

FRAME NAME:

Precipitation Weather Procedure

CAUSE OF FRAME ACTIVATION:

Precipitation

Outside travel on foot

PROCEDURE CAUSES:

Use umbrella

The second type of frame could be described as an "is a" frame, which describes what something is. In this case, the frame explains that rain and snow both are precipitation.

FRAME NAME:

Precipitation

CONDITION:

Rain

Snow

IS:

Precipitation

Once the Precipitation Weather Procedure was invoked, the system would attempt to determine if the causes of frame activation were present. That is, is there precipitation? The system would then activate the precipitation frame and determine that rain or snow is precipitation. Then, the system would check to see if rain or snow existed in the

current database. If it were raining, the precipitation frame would be valid. Once this check is complete, the precipitation weather procedure would then be valid if the database indicates the person is going out on foot.

2.6 Gathering The Knowledge

It is fine to talk about how the system is implemented once the knowledge is available, but the more difficult part of building the system is extracting the knowledge from experts. People, in general, have difficulty articulating how they attack a problem. First, they are not conscious of some of the steps they take. If pressed for an explanation, people tend to justify what they did rather than explain it. This means they will rationalize what they did because they were not conscious of the real approach to the problem.

The process of knowledge extraction is called knowledge engineering. Knowledge engineers use a variety of approaches to get the information from the expert, including interviews and having the expert speak as he is solving a problem (15). The difficulty in generating rules has led to some interesting experiments in automatic rule generation.

However, once a rudimentary set of rules is in place the expert can then modify, add, or remove rules as necessary.

Remembering our IF-RAIN-THEN-USE-UMBRELLA expert system, a rudimentary rule would be the first one: IF [rain] THEN [use umbrella]. Now the expert uses the system and finds that this rule leads to the use of an umbrella if you are sitting in your living room and it happens to be raining. She'd then say that the result wasn't right and that the rule should include a reference to going outside. You'd then probably modify the rule so that it looks like the second example, with the result that, if you were riding in your car, you'd use an umbrella. Again, the expert would say this isn't right and this cycle of modify and test would continue until the rule the expert is looking for is achieved.

There have been some successful automatic expert system modifier programs which change the rules by interacting with the expert directly and without the use of the knowledge engineer. TEIRESIAS is a modifier program for the expert system MYCIN. When an expert uses MYCIN and finds that MYCIN is coming to the wrong conclusion, TEIRESIAS is invoked and it tells the expert which rules were used to come to the conclusion, asks whether a rule is missing, or if a present rule is wrong or incomplete. TEIRESIAS then takes the reply from the expert translates it into the format required by MYCIN, and determines the effect of the rule change on the result. If the result is still wrong, the cycle repeats.

2.7 Success Rate

Eventually the question arises: How successful are these systems? Without intending to hedge, the answer depends on what you mean by success. These systems get very good at what they do; so if by successful you mean accurate, these systems are very successful. If by successful you mean used by the people for whom they are intended, they are less successful.

Some of the systems, especially those physically located where there are few or no human experts, for example, the oil drilling expert systems, are used frequently. Those which are to be used by experts themselves, such as the medical expert systems to be used by doctors, are under utilized. The reason for this user resistance has to do more with the psychology of the user than with the accuracy of the system.

To understand this consider how two human experts interact. It isn't simply one expert telling another the answer. There is an exchange of opinions and ideas with both sides explaining their reasoning. The early expert systems didn't necessarily take this exchange into account, and instead would just tell the user what the answer was without telling him how this answer was arrived at. No reasonable doctor would simply take the diagnosis given by a computer without knowing how this was

arrived at. Some of this user resistance can be removed by including as part of the system an explanation facility to show the reasoning behind the result. Some of the more sophisticated expert systems under development actually have several levels of explanation, so that an expert familiar with the reasoning doesn't have to be bothered by detail, while someone who is less familiar can keep asking for more justification (16).

3 LABORATORY DATA PRODUCTS

3.1 Current Investigations

Currently, there are two expert systems in early research and development at Laboratory Data Products. The first is a Real-Time System Configurer, similar to XCON. The second is a Statistics Expert Advisor and Process Monitor.

3.2 Real-Time System Configurer

With the various real-time hardware products available and certified to work with DIGITAL computer systems, it is difficult for the sales representatives to know what is available for each system type. This has been further complicated by having some options not certified for different systems, although both those systems have the same bus. For example, not every option built for the Q-bus is certified to work on the microVAX, although the microVAX has a Q-bus.

XCON is used to configure complete systems, so if a sales representative wants to add an A/D converter to an existing system, he can not use XCON to select the A/D. It was decided to make a system similar to XCON, but capable of specifying real-time product configurations. The proposed system will encompass both DIGITAL and third-party vendor products. Additionally, the intent is to expand the prototype system to include software available through LDP's public domain library.

The prototype is being written in OPS5, a DIGITAL-supported language, from which rule-based expert systems are easily constructed. For an example of the type of rules that make up the system, assume that the user has specified that the computer used is a VAX 780 and that an A/D is required. Once the user has entered A/D, the system must inquire how many channels are needed. The rule to ask how many channels are required if the application is an A/D would look like (in OPS5):

```
(p number_chan_required
(appli_reply ^application <c> = A/D)
-->
(write (crlf)
How many single ended channels required?)
(make_reply ^channels (accept)))
```

This rule is read as follows. The p

indicates this is a production, another word for a rule. number_chan_required is the name of the rule. The next set of parentheses, anything before the arrow and after the name of the rule, is the conditional part of the rule, the IF part. The IF part of this rule says that, if the variable, Application, in the answer appli_reply is equal to the analog to digital converter A/D THEN (THEN is indicated by the arrow) write the sentence shown, and accept the answer into a variable Channels and name this answer reply.

After a number of such questions are answered, the following rule would be used to select the proper device.

```
(p admux_U_selection
(AD_module)
(unibus)
(mux_required)
-->
(write (crlf)
The modules required are
AD11-K, LP11-K and AM11-K.))
```

Again translating this rule, the p indicates a procedure which has for a name admux_U_selection, an abbreviation for A/D multiplexer selection for the unibus. The requirements for the rule to fire are: the need be for an A/D, AD_module, that it be for a unibus, unibus, and that more than 16 channels be needed, mux_required. When

these requirements are met, the rule specifies the modules listed after the arrow.

When this system is completed, it should make the selection of real-time hardware and software less complicated. Eventually a user will be able to call into the system and see what options are available for various configurations.

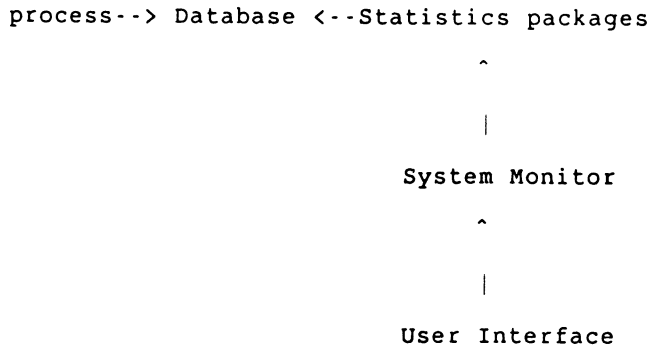
3.3 Statistics Expert Advisor

A second system in LDP research is a statistics expert system. The statistics expert system is a joint effort involving within Digital; LDP, the Central Quality Group, and the AI Applications Group. The system as now envisioned consists of two portions, an automated data-monitoring system and a statistics advisor.

The assumption in creating a data monitoring system is that a higher degree of monitoring would lead to better process control. An automated system is suggested because humans have a difficult time keeping up with the large amount of data generated by manufacturing processes.

The implementation consists of a database into which the process is placing data, a statistics subsystem to calculate the relevant statistics about the process, a system monitor to evaluate the statistics

and determine how the process is behaving, and a user interface to provide the information in a form the user can assimilate.

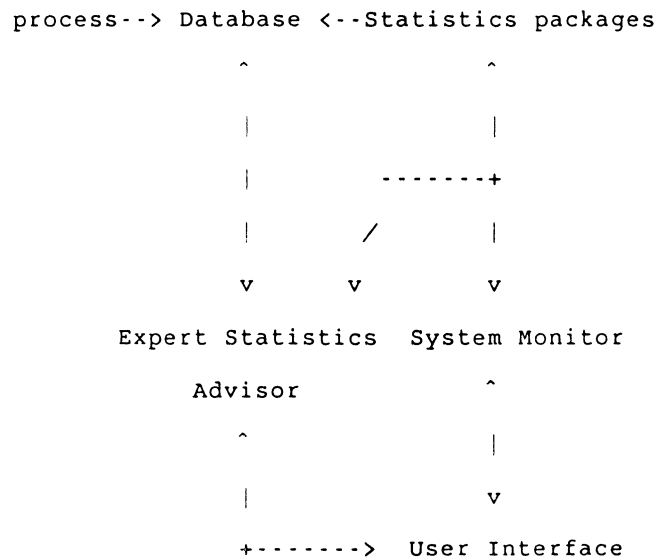


The system monitor may be either an expert system or algorithmic in nature. An example of the heuristics which could be used as part of a system monitor built from an expert system is:

```

IF
  [ 2 out of 3 consecutive data points
    fall outside 2 standard deviations on
    the same side of the mean ]
THEN
  [ the process is shifting toward that
    side of the mean ]
IF
  [ 7 consecutive data points fall outside
    1 standard deviation on the same side
    of the mean ]
THEN
  [ the process is shifting toward that
    side of the mean ]
  
```

The second part of this system is an expert statistics advisor. This advisor will help the investigator choose the most appropriate statistical techniques for analyzing the data under investigation. The assumption here is that most researchers misuse statistics either by using the wrong statistical techniques or by using statistical techniques which are not the most applicable to the data. The system will not be able to help the researcher determine whether the results are meaningful, only whether they are significant statistically. This system is coupled with the automated process monitor, so that a user will be able to go into the database, and examine the data using various techniques and then inform the monitor which parameters should be monitored. Together the complete system would be configured:



The expert statistics advisor notifies the user if a data point is too far from the value expected, and gives the user the ability to modify, ignore, or use the data point as is. The expert system would make calls to the statistics packages when appropriate and give the user the results through a multiwindow user interface.

To do this a set of heuristics which determine applicable statistical tests are used. For example, two rules which might be used by the system to help the user with t-statistics are:

```
IF
  [ a t-test has been requested to compare
    2 sets of data and there are an equal
    number of data points in each set ]
THEN
  [ suggest a paired t-test to
    the user as a stronger test ]

IF
  [ a paired t-test has been requested
    to compare 2 sets of data and there
    are an unequal number of data points
    in each set ]
THEN
  [ inform the uer that this test
    is invalid ]
```

In all, many statistical tests have subtleties which are generally lost on the casual user. A statistician examining the

data would prevent a user from using an inappropriate test. This expert system will attempt to do the same.

The four-window user interface contains:

1. The interactive window

The user interacts with the system through this window, answering questions and getting replies.

2. The inference engine monitor

The user views, through this window, how the system came to the conclusions it did.

3. The database window

The user can examine and modify the database through this window.

4. The graphics window

This window is used to display control charts, scatter diagrams, and other statistical graphs that the user may require.

The expert system prototype will be built in LISP and will run under VMS. If possible, the system will be built as a

system of interfaces which will allow the user to specify any database management package and any callable statistics package. Whether we will have the ability to make the interface standard enough to allow this is not currently known. The graphics portion of the system will require GKS.

4 CONCLUSIONS

LDP is beginning to investigate areas in which expert systems might be useful in the laboratory. Currently, there are two systems under investigation. The first is a real-time system configurer which will aid the user in determining what hardware and software meets his needs. The second is a statistics system which will monitor processes to increase process control and give expert advice as to which statistical tests are most appropriate in analyzing data. It is hoped that these systems will bring the power of statistical analysis to users in a form that will help them extract a maximum of information from their data.

This paper presents ongoing research within the DIGITAL EQUIPMENT CORPORATION and as such does not represent a commitment by DIGITAL to continue to develop or produce as products the investigations and prototypes mentioned within this paper.

5 ACKNOWLEDGEMENTS

The author would like to thank Dr. M. Joshi, Mark Turner, Meyer Billmers Dan Theriault, and Will Anderson for their helpful discussions. In addition, I would like to especially thank Susan Brown for her editing of this and other papers and Jane Whitney for preparing the paper for publication.

6 BIBLIOGRAPHY

1. Some Thoughts on Expert Software, J. Chambers, Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, W.F. Eddy (Ed.) Springer-Verlag, N.Y. 1981, pg. 36
2. The Applications of Artificial Intelligence to Law: A Survey of Six Current Projects, Cook et al., Proceedings of the National Computer Conference, 1981, pg. 689
3. Expert Systems and Statistics, Darius, SEAS SM84 Session Report p.529 1984
4. A Logic-Based Expert System for Model-Building in Regression Analysis, F. Darvas, K. Bein, and Z. Gabanyi, Logic Programming Workshop, 1983, pg. 229

5. An Overview of Production Systems, Davis and King, Machine Intelligence, 8, 1977, pg. 300
6. The Fifth Generation, E. Feigenbaum and P. McCorduck, Addison-Wesley, Reading, Massachusetts, 1983
7. The Role of Frame-Based Representation in Reasoning, R. Fikes and T. Kehler, Communications of the ACM, Vol. 28, No. 9, Sept. 1985, pg. 904
8. Artificial Intelligence Research in Statistics, W.A. Gale and D. Pregibon, The AI Magazine, Winter, 1985 pg. 72
9. An Expert System for Regression Analysis, W.A. Gale and D. Pregibon, Computer Science and Statistics; Proceedings of the 14th Symposium on the Interface, Heiner (Ed.), Springer-Verlag, New York, 1982, pg. 110
10. Rule-Based Systems, Hayes-Roth, Communications of the ACM, Vol. 28, No. 9, Sept 1985, p.921
11. The Technology of Expert Systems, Michaelsen, Michie and Boulanger, BYTE, April 1985, pg. 303
12. An Expert System For Statistics, R.O'Keefe, presented at Theory and Practice of Knowledge Based Systems, 14 Sept 1982, Brunel University, Surrey, England, pg. 23
13. Development Of A Friendly, Self-Teaching, Interactive Statistical Package For The Analysis of Clinical Research Data: The BRIGHT STAT-PACK, Rodbard, Cole and Munson, 1983 IEEE Computer Applications In Medical Care, p.701
14. Artificial Intelligence and the Future of Medical Computing, Shortliffe, Computers in Medicine, Feb 12, 1984
15. Inferring an Expert's Reasoning by Watching, Wilkins, Buchanan, and Clancey, Stanford Heuristic Programming Project, HPP-84-29, June 1984, (appearing also in 1984 Conference on Intelligent Systems and Machines)
16. Human Interface Aspects of Expert Systems, R.M. Young, in Expert Systems, State of the Art Report 12:7, Ed. J. Fox, Pergamon Infotech Ltd. (1984)

Copies of the slides from the presentation are available from the the author.

DATA MANAGEMENT SIG



B. Z. Lederman

2572 E. 22nd St.
Brooklyn, N.Y. 11235-2504

The purpose of this paper is to make people aware of what data encryption is, how it is used, who needs it, and why it is needed. It is intended as an introduction to the subject, so it will not go deeply into the mathematical internals of ciphers.

As is true for many subjects, what something is and how it is used is often interlinked, so that one needs to understand one before the other can be explained; so to begin with, some very simple definitions will be given, and later they will be expanded.

What is it?

Cryptography covers the general field of transmission of information which is protected from unauthorized access, and includes secret writing (concealing a message by various means), codes, ciphers, and their use and defeat. Lately, encryption and decryption have come to be used in place of encipher and decipher to refer specifically to the use of ciphers to protect data, and will generally be used as such here.

Stated more simply, data encryption is a method of protecting data so that it can be accessed only by the people who are supposed to be able to get to it. This definition, while correct, is rather vague (it could apply equally well to the physical protection of data such as locking it up in a safe, or translating it into an obscure language): it does, however, explain the purpose of encryption, which is to limit the accessibility of selected items of information. This will be explained first, as it is desirable to understand why access should be limited to understand how it is to be done

Why is it used?

If you are working on a computer system which can be accessed by one or a very limited number of users, and which has no outside lines (no modems or dial-in lines), and which stores all information on easily removable media (floppy disks or tape cartridges), and you always remove this media and lock it in a safe when you are not using it, then you may not need encryption. If you can eliminate all access to your data other than by having the key or combination to the safe, and if no-one can look over your shoulder or otherwise tap into your computer or terminal lines while you are examining your data, then access to your information has been made about as secure as possible through physical means, and encryption is probably not necessary. Unfortunately, this ideal state of affairs does not often exist. Sometimes your storage media cannot be kept in a safe, or you must store your information on a fixed disk which cannot be removed, or you must share the system with many other users at the same time, or you must have dial-in lines so that people outside your physical location can access the same

machine, or you must send information to other locations: in any of these cases, you may need to limit access to your information, and encryption is one method of doing this.

The immediate reaction many people have to this is: "Our computer is used only by people within our company. We don't have dial-in lines, [or our dial-in lines are secured by other methods, such as passwords or dialback], and all of our terminals are within our company area. Why do I have to protect my data?" Even in this situation, there may still be good reasons for using encryption.

Confidentiality.

First, you may have information which you are obliged to keep confidential. If you use your system to administer company medical benefits, for example, you may be obliged to keep personnel medical records confidential. Without some sort of encryption or other protection scheme, it may be possible for many people in your company to peruse the medical records of other employees at will. Even if you are certain no-one will do this, increasing demand for rights to privacy of personnel records may set a legal requirement that you protect information from indiscriminate access. (Note that encryption will not protect against the persons who must still have access to the data: other checks are needed to insure that persons who must have the data will not misuse it.)

Next, there may be information you want to keep confidential. If you use your system to keep track of employee performance records, or calculate salaries as part of your budget planning, you might not want the employees involved to read or modify that data. It is all well and good to say you trust your employees, and probably most people can be trusted: but locks were invented to keep out the small percentage of society which cannot be trusted. I rather imagine that most people reading this paper have locked their houses and cars before leaving them, even if they trust most of their neighbors: if you would do that, then you probably have information which should also be "locked up". Similarly, you might be preparing information for contracts, order placements, payroll records, competitive bids, and similar information which could represent a significant portion of your company's assets, and might be several times the annual salary of many of the people who have access to it (and they are not always only the people whom you think have access to it). The more important an item of information is, the more likely it is that someone could benefit by getting it, and therefore

the need to protect it increases directly with its importance.

Unauthorized access.

The case where a "hacker" or other unauthorized person calls into a computer system and proceeds to cause various type of mischief and/or damage is one that probably most people fear. You may have a system where it is necessary to have dial-in access for your own personnel, and it then becomes necessary to guard the system as much as possible. There are various methods of limiting access to a system through passwords, or through hardware, which are outside the scope of this paper. Data encryption can act as a second line of defense, however, and should also be considered. In many cases, "hackers" are simply looking for files they can read, or programs they can run: encryption can make data unreadable and programs unrunnable, and thus defeat two of the hackers main goals. Encryption will not prevent the random modification of data (where the modifier doesn't care what the change actually does) or deletion of files: other methods of protection are required to guard against that type of damage.

Protection on "outside" systems.

The situation may also be reversed, as many computer users do not own their own systems and have to use time-sharing or other outside computer processing to store data and provide other computer services. In this case, you may have little control over who in the world has access to your data. An encryption scheme that can be implemented on your own data on the outside machine would be one way of protecting your information. Similarly, many companies store copies of their records in outside warehouses or other storage facilities to protect against fire or earthquake damage at their main location, and while such facilities usually provide guarantees against unauthorized access, some extra protection might be desirable.

Protection during transmission.

One last situation which probably occurs to most people is when data has to be transmitted from one location to another, usually over some public facility (telephone, Teletype/telex, leased communication line, air freight, or mail). It is actually more likely that the data will be accessed from within your company than from without (intercepting telephone channels from microwave links is possible, but rather difficult), but the more important the information is, the more likely it is that someone will try, and it wouldn't hurt to take some reasonable precautions. If you are engaged in any type of electronic funds transfer, such as depositing your employees payroll directly to their bank accounts, or transfer of company assets to your bank or to other companies, the sums of money involved may be so great that not encrypting the data in some way is courting disaster. Consider what would happen if someone were to change the records just once: if that would seriously hamper your business, or cost you a significant amount of money (either by direct loss or the effort to replace the missing information, or loss of goodwill of the person at the other end), then you should consider encrypting your data. Remember that the true cost of data might not be just what it cost you to obtain it, but also what it will cost if you lose it.

Other protection methods.

It can be seen, therefore, that many users will have some use for a data protection scheme of some kind, as nearly everyone has some type of information which is not to be accessed by everyone else. This leads to the methods which can be used to protect information. Various computer operating systems are in use today, some of which include access protection through requiring users to log into accounts, or various methods of verifying that persons accessing dial-in lines are properly authorized, or through protection codes within the storage system (such as the file protection codes used in RSX-11, RSTS, and VMS). These are outside the range of this paper, but it will be mentioned that they don't always provide the limit of protection needed, either because there has to be at least one privileged user of the system who can bypass the checks, or because backup copies of the data must be stored off of the machine, or from other limitations of the system. Even when such schemes work well, they may not be enough, and they don't work at all if the information has to be sent outside (by wire or mail, etc.). This leads us back to data encryption, which will allow the information to be protected by a method which is independent of any protection which may be provided by the operating system. This does not mean that other protection schemes should not be used, or that encryption is the answer to everything, either: different protection schemes cover different areas, and usually complement rather than substitute for each other.

Once the need for some type of data protection is recognized, a protection scheme must be selected. As previously mentioned, cryptography covers, in general, secret writings, codes, and ciphers.

Secret Writing.

Secret writing covers such things as invisible inks, and concealing messages within other messages. This is a highly specialized field, and one which is not likely to have much general application: it is usually too cumbersome for easy use, and is not applicable to storage of large amounts of information on computer media. Just to show what it is like, consider the message:

"Inspect details for Trigleth, acknowledge the bonds from Fewell."

which doesn't seem to mean anything. If you take the third letter of each word, however, you get the message "Strike Now". This is an example of secret writing, (a method which follows a fixed formula like this may also be called a concealment cipher), and it can be easily seen that it would not be easy to use: if it had no other faults, the concealed message has become over 6 times its original length, and if you have to pay for disk storage space or transmission costs, you can see a big disadvantage to this type of protection. Invisible inks can be used on paper messages, but obviously won't work at all on data stored on disk or magnetic tape. (There was one fictional story where a message was written on a reel of tape with a grease pencil, but this tends to gum up the tape drive, and isn't very practical.) They can be useful to authenticate documents, as they cannot be duplicated

by photocopying machines, but again, this is a field where expert assistance from a printing company or ink manufacturer is required. The one and only advantage to secret writing is that many countries are implementing restrictions on trans-border data transmission: even though they encrypt their data, they won't let you encrypt your data, so they can monitor your transmissions: a good method of secret writing might evade this restriction, but most methods are too cumbersome to be practical. We will not give any more attention to this subject.

Codes.

A code is the arbitrary mapping of symbols to other symbols. It is usually one to one, but can be one to many or many to one. One example of a code which is in very common use every day is ASCII, the American Standard Code for Information Interchange, used by most computer terminals to map binary signals to numbers, letters, and other characters, a portion of which is shown here.

040 SPA	060 0	100 @	120 P	140 `	160 p
041 !	061 1	101 A	121 Q	141 a	161 q
042 "	062 2	102 B	122 R	142 b	162 r
043 #	063 3	103 C	123 S	143 c	163 s
044 \$	064 4	104 D	124 T	144 d	164 t
045 %	065 5	105 E	125 U	145 e	165 u
046 &	066 6	106 F	126 V	146 f	166 v
047 '	067 7	107 G	127 W	147 g	167 w
050 (070 8	110 H	130 X	150 h	170 x
051)	071 9	111 I	131 Y	151 i	171 y
052 *	072 :	112 J	132 Z	152 j	172 z
053 +	073 ;	113 K	133 [153 k	173 {
054 ,	074 <	114 L	134 \	154 l	174
055 -	075 =	115 M	135]	155 m	175 }
056 .	076 >	116 N	136 ^	156 n	176 ~
057 /	077 ?	117 O	137 _	157 o	177 DEL

text was much shorter than the original message: two groups of five letters could be pushed together to make one 10 letter group, which was counted as only one word in the cost of sending the telegram. Since the mapping is arbitrary, codes can be very secure. Generally, you have to have the arbitrary mapping in order to defeat (or "break") the code, though if the code is re-used often enough, the mapping can sometimes be deduced. They are also vulnerable if one can obtain a copy of the plain text and the coded text which goes with it, and of course are defeated if the wrong person obtains a copy of the code book. Some authorities consider book codes like this that are used once only to be completely unbreakable, and it would be easy to use a computer to generate lists of arbitrary code words to use.

Codes do have many disadvantages in the computer environment, however. A computer program to automatically code a message with a scheme like the example would be very complex, as the context of the entire message is needed to search through the list of phrases on the right and find the appropriate code word: decoding the message by looking up the letter group would be a easier. Encoding large strings of numbers is tedious and likely to increase the size of the message, and there is always the problem of what to do if you need a phrase which is not pre-defined in the code book. Binary data cannot be coded at all which this particular scheme, and would be difficult to encode with most coding schemes. Since we would like a method which would work on a computer, and accommodate a wide variety of data with a minimum of human intervention, we will not consider codes further.

Ciphers.

A cipher is a method of transforming data from one form to another through a logical process, usually with a geometric or mathematical basis. Since a cipher is a method or system rather than a group of pre-defined mappings, it should be possible to transform any "plain" or "clear" text, regardless of length or content, into a single enciphered message. This is more easily understood with an example, such as a simple geometrical cipher. I will take the familiar phrase,

"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOGS BACK"

and write it out in a square in the usual fashion, left to right, top to bottom.

```

T H E   Q U I
C K   B R O W
N   F O X   J
U M P S   O V
E R   T H E
L A Z Y   D O
G S   B A C K

```

To encipher this message, I can take the letters out by some sequence other than the way they went in: for example, top to bottom, right to left. (This is an example of a transposition cipher, as it works by transposing or changing the order of the letters in the message, but not the letters themselves.) This will give me:

It isn't usually thought of as a code, and it certainly isn't a secret, but it is a code: it transforms one type of data into another through an arbitrary mapping. Note that the mapping is indeed arbitrary, even though the letters do follow the alphabet for convenience: there is no reason why they would have to do so for the code to work.

Another code which better fits the general public's perception of a code is the type of code which has been used for telegrams, a portion of which is reproduced here:

```

MUWUB Improving rapidly
MUXAW Improving slowly
MUXEX Is not improving as I/we could wish
MUXIZ Is there any change
MUXNO Is there any improvement
MUXPU Progressing satisfactorily
MUXRY Sorry to year you are (..... is) ill
---
MYGEL How would
MYGIM HURRY (See Haste)
MYGON HYPOTHECATE-D
MYHAL IF
MYHCI And if
NYHDO And if not

```

and so on. It can be seen that the mapping between the original phrase (the "clear" or "plain" text) on the right and the code word on the left is completely arbitrary, and that the book is the only way to go from one to the other. This particular code had the advantage that in most cases the coded

"IWJV OKUO OEDCQRX H A BOSTYBE FP Z HK MRASTCNUELG"

which doesn't look anything like the original. The underlying principle here is that there is a definite method of transformation between the original text and the enciphered text without considering the actual content (even if it is not obvious on a cursory inspection), whereas in a code the transformation was completely arbitrary and very sensitive to content. Because ciphers work on a method of translating data from one form to another, they are generally much easier to implement on a computer, and they are generally much less data sensitive than codes would be. In this example, each character could easily be a byte or word of binary data, and the scheme would work just as well.

There are a great many types of ciphers, some more secure than others, and some easier to use than others. One which is very common, and even occurs in some daily newspapers, is a simple letter substitution, where one letter is replaced by another. For example,

ABCDEFGHIJKLMNOPQRSTUVWXYZ

can be replaced with

EFGHIJKLMNOPQRSTUVWXYZABCD

This is a substitution cipher, which changes the letters in the message, but not their order in the message. This would make the sample phrase "THE QUICK BROWN ..." come out to be:

"ZLI UYMG0 FVSAR JSB NYQTW SZIV ZLI PEDC HSKW FEG0"

Since this is a one to one mapping, I am going to leave it to the purists to determine if it is a code or a cipher, though it is content insensitive (there is obviously some overlap between some codes and ciphers). The drawback to a simple cipher like this is that it is too easy to break with just a pencil and paper, and with even the least expensive home computer it is literally child's play. (You can read "The Gold Bug" by Edgar Allan Poe or "The Adventure of the Dancing Men" by Sir Arthur Conan Doyle to find out how.) There have been many other, more sophisticated, transposition and substitution ciphers than the ones demonstrated here in use in the past few centuries, but since they were all implemented by hand, they are all too easy to break by modern methods. You can simply go out and buy a number of books that will tell you exactly how to do it with just pencil and paper, and the proliferation of home computers makes most of them very simple to break indeed. They may still be adequate for some purposes however, but considering how good a cipher needs to be will be discussed later.

Modern Ciphers.

If existing ciphers are too easy to defeat with computers, then what is left? The answer is that most modern encryption schemes are based on the same principles as older ciphers, but use the power of the computer to expand the magnitude of the scheme. For example, in the transposition cipher shown, the

box was 7 letters on a side: it could be made larger, but when encryption is done by hand, a box much larger than 15 or so on a side becomes too cumbersome to use. With a computer, however, there is no limit to the size of the box: simply increasing the box to 100 per side makes it too large to "break" the cipher by hand. This scheme of using the computer to expand on a good encryption method can be used to create ciphers that are difficult to defeat, even with another computer (the box cipher would still be too easy to break by computer and is given only to illustrate the idea). One which I have used is a variation on the periodic number substitution (also known as an addition or Vigenere) cipher. In this scheme, a number sequence is added to the text: a simple example would be to add the sequence

1357135713571357135713571357135713571357135713571

to the numeric value of the ASCII characters in the message

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOGS BACK

to get this:

UKJ'RXNJL#GYPZS'GR]'KXRWT#T]FU%[IH%SB]`'ERLZ!EFJL

With a number sequence this short, the cipher would not be too secure (you can see even in this short message that a SPACE becomes a ' four times, and the sequence "SPACE-something-U" has twice been changed to "'-something-X", for example) though it is more secure than the simple substitution cipher shown before. Various methods of obtaining a less repetitive sequence have been tried in the past, but usually produce no real increase in security. Using the computer, however, a number sequence can be generated that appears to be random, and is thousands of digits long. Most computer languages have a random number generator (or more accurately, a pseudo-random number generator, as the sequence can be repeated exactly when desired), such as:

LET A = RND(B) in BASIC, and
A = RAN(B) in FORTRAN,

and similarly for other languages. There are theoretically an infinite number of such pseudo-random sequences, and even for a specific generator there are a very large number of specific sequences: in DEC's Fortran-77, the number that starts the sequence (the variable B) can have at least two billion possible values. This particular cipher is sometimes called the Fast "Infinite-Key" method, and has been widely used with good results. We could then repeat the above procedure by generating a pseudo-random number sequence such as:

1986833925153857265815341697347183799587665798742

and adding it to

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOGS BACK

to obtain

UQMYXLLM%CWR_S'HU](KZPTT&X]HV'UPH'UJ_a'JULZ)JHGM

At first glance, this doesn't appear significantly different from the first example, but if someone were to attempt to defeat the cipher by the usual method of looking for repetitive patterns and common adjacent letters, they wouldn't find any, and would not be able to defeat the cipher. This cipher has the additional advantage over the "box" cipher in that the characters can be processed in the order they are read: in the box cipher, a large portion of the message has to be read in and stored before any of it can be processed. In most computer ciphers, it is an advantage to be able to process the message serially, and to not have the length of the message have any effect on the encryption scheme itself, especially then the messages being processed are being transmitted from one place to another (over a communications line, or to a disk or tape drive are two examples).

It can be seen, therefore, that even though the computer has made it easier to defeat some encryption schemes, the power of the computer can also be used to raise the complexity of a cipher to the point where it is very difficult to defeat, even with another computer. This is the basic principle behind most good modern computer ciphers: the use of the computer to raise the complexity of the cipher until it is (hopefully) beyond the ability to defeat by any practical means.

Data Compression?

It was mentioned that the telegraph code example shown earlier also compressed the information into a more compact form. There are a number of data compression schemes in use on computer systems to minimize the amount of space data occupies when stored, or to reduce the amount of time needed to transmit information from one location to another (and hence reduce the cost of transmission). Some of these compression schemes could also be thought of as ciphers, as they transform data from one form to another. While they have the obvious advantage of compressing the data, generally the compression algorithms are too well known for this to act as a really secure cipher.

With some understanding of what encryption is, we can perhaps present a better definition. One such definition could be:

"Encryption is a method of transforming data into a state where it is not easily available to persons other than those for whom it is intended (using ciphers)."

This is a very general definition, and it does

appear to be somewhat cumbersome, but it is worded in this way deliberately. Note especially the emphasis of the phrase, "not easily available". Generally, no encryption scheme is absolutely secure from ever being defeated, and a decision has to be made as to how good a scheme is needed. From a practical standpoint, the real purpose of encryption can be defined as this:

"To make obtaining the data more expensive than the data itself is worth."

(Where expense is counted in time, effort expended, cost of labor, cost of computer services, etc.)

While this definition may not precisely define a cipher, it does clearly define the goal encryption should achieve.

To evaluate a potential encryption scheme, one must consider from whom the data is being protected. Some possibilities are:

1. Curious employees
2. "Hackers"
3. Outside visitors
4. Service personnel and/or vendors
5. Competitors
6. The Criminal Element (internal or external)
7. The IRS
8. The "spooks" (CIA, NSA, KGB, MI5, etc.)

among others. The first four can probably be discouraged with even a very simple cipher: as mentioned before, most "hackers" and other idle curious are simply looking for files that can be read or run. If they were to see a file such as this:

```
RTP $%& &.2H8I]).4HHQPPJ8IKNUIOQPP
RUP $%& &.3H8I],%.H342DH).4H8III
RVP 2%-
SQP 02).4 B#/-054!4)/. /& -/2'!'% 0!9-%.43B
SRP 02).4
SUP 02).4 B0,%!3% ).054 4(&% 02).#)0!, H7)4(
SVP ).054 0
SWP 02).4 B).054 4(% !.5!), ).4%2%34 2!4% H). EIB[
SXP ).054 )
SYP 02).4 B).054 4(% 4%2- H). 9%!23IB[
TPP ).054 4
TQP 02).4
TSP 4]4JQR
TUP 1])
TVP ))OORPP
UPP -]&.2HOJ)OHQM0QHOK)I>4II
UTP 02).4 B02).#)0!,B[4!"HSUI[BDB[0
UUP 02).4 B).4%2%34 2!4%B[4!"HSTI[1[BEB
UVP 02).4 B4%2-B[4!"HSTI[4[4!"HTPI[B-/.4(3B
UWP 02).4 B-/.4(,9 0!9-%.4B[4!"HSUI[BDB[4!"HSXM&.3
```

they might well pass it by, or maybe make a few simple attempts to read the file as if it was binary data. But if anyone should happen to figure out or guess that it is really a BASIC program, then it would not take long to decipher it, as it happens to be encrypted with a simple letter substitution cipher. Since a computer is going to do the work, it would be just as easy to use a more secure cipher, and one which will transform the data into something which will not look like obviously encrypted data when examined. For example, the "Infinite-Key" method takes no more computer time or disk space than simple substitution, is very much

more secure, and the resulting data doesn't look at all like text, so there is no reason to use the simple substitution when such superior methods are easily available.

If interception of data by a competitor, or by a dishonest employee (which is really the greatest threat) is a serious consideration, then you will probably want the most secure cipher that can be reasonably implemented (one which protects the data well, but will not use up so great an amount of computer resource that it becomes more expensive than the data it is protecting).

If you intend to protect your data from categories 2, 3 and 4, then other protection schemes should be your first choice, such as not allowing outside visitors to wander un-escorted about your plant, removing your data from the system before allowing it to be serviced by outside personnel, and using various protection schemes to prevent unauthorized dial-in access. Encryption of data can act as a second line of defense in these cases, however, and should still be considered: it must be stated again, however, that encryption is not necessarily the best solution to every situation, and that all methods of protecting data need to be evaluated to determine what best suits a given need.

Against the last two categories: you have to be realistic, and understand that any government agency that can put the gross national product of a world power into it's efforts is going to be able to break any cipher you could use. That doesn't mean you have to make things easy for them, and there are ciphers available now which are very difficult for anyone to defeat, but you must remember that no cipher is absolutely unbreakable.

How Good is Good Enough?

It was stated that a good encryption scheme costs more to defeat than the information is worth. This means that the cost of the labor expended, and computer resource dedicated to the task are more than the ultimate value received from the information which may be obtained. For example, the only ways known to break the Infinite-Key and DES ciphers is by brute force: trying every possible key, and looking at the result to see if it makes sense. Even if someone is willing to dedicate a computer to the task, it could take months or even years of effort to break one message, by which time the information may be useless. In addition, the time a computer spends on breaking the code cannot be used for anything else, like doing payroll, or inventory, or other normal business functions. If you are preparing bids on a contract which will yield, say, \$10,000 and a competitor tries to steal your information and under-bid you, then your encryption scheme is successful if it either takes so long to break cipher that the competitor can't meet the deadline for submitting bids, or if it costs the competitor more in computer resources than the \$10,000 or so that the contract would yield: even though the cipher is broken, the person who broke it comes out with a net loss. Few "hackers" are going to have the patience to let their home computer run for several months or years to decrypt one message and not use the computer for anything else, and not much information is so valuable that it would be worth while renting a Cyber or Cray super-computer for several months to break the message relatively quickly (unless you are a government agency, and can do whatever you like).

It is possible that someone within a company might use the company computer to try to break a cipher by brute force, reasoning that the computer time doesn't cost them anything. Since defeating a good encryption scheme would use up relatively large amounts of computer time over an extended period, it should be possible to detect if anyone within a company is using the computer system in this manner, and deal with the problem directly.

Other Necessary Precautions.

A consideration which is equally important as the selection of an encryption scheme is keeping the keys themselves secure. Just as it would do no good to buy the most expensive lock and lock your house and if you then put the key under the door mat, it does little good to encrypt your data if anyone can get the key. In terms of internal security, this often means correct selection of a key to use: since most modern ciphers use a number as the key, there is a great temptation to use an easily remembered number such as your telephone number, birth date, social security number, wedding anniversary, or some such number as a key. Unfortunately, any number that you can remember easily will also be easy for anyone who knows you to guess. If you are trying to protect data internally in your company, using such a number would defeat the best cipher: rather than having to try several billion possible keys, the number of attempts are reduced to a few dozen or so. This leads to the paradox that you must chose a number you can remember (or you may never get your data back if you forget the key), but one which no one else is likely to guess; or else you have to write the number down, but in a place where no one is likely to get it. The latter scheme is probably better than trusting to memory, but you should not keep important numbers laying about: keep them in your wallet (and keep your wallet with you), or in some other secure place. Similarly, don't put them in the telephone directory or card file that sits on top of your desk, or in other easily accessed places. It is also a good idea to change the keys periodically, especially if it is being used for data transmitted externally. (Internally, the threat is greater that someone will figure out your key, or may see you type in the key, or be able to compare the encrypted data with the "clear" data, and deduce the key that way). Basically, you must use at least as much caution in dealing with cipher keys as you would use in handing out door keys to your plant, or electronic lock keys to your personnel: they all protect your assets, and have to be treated with the same respect. While you can hire guards for physical security in a plant, you cannot do the same for information in a file or transmitted over a wire, and information is easier to move than equipment; so if anything, the cipher keys must be kept even more secure than other kinds of keys.

Public Keys.

When data has to be transferred from one location to another, then the risk is doubled, as the key has to be kept in two places. One absolute rule is that you never, ever, transmit the key with the data it protects (you might just as well not bother encrypting at all). It is usually a good idea to use an encrypted transmission to send the next key to be used at one time, and the data at some other time, and that both parties must exercise the same caution in protecting the keys. Otherwise,

you must use some secure method of transmitting the keys to the locations where they will be used (such as sending someone you can trust to carry them by hand), and storing them in a safe or other secure location. One partial solution to the problem is the Public Key method of selecting keys. This is not an encryption scheme, but is a method where two people can create a large numeric key by each selecting a number which forms half of the key, and where each party knows only half of each key. The advantage of this method is that one half of the key can be made public, and anyone can use it to encipher a message intended for you, but only you can decipher the message using the other half of the key which was kept secret. This can also be used for source verification if both halves are kept secret: for you to be able to decipher the message, it will have to have been enciphered using the matching half of the key. The method is based on the fact that it is difficult to factor a very large number which is the product of two very large prime numbers (each party picks one of the large primes): lately, there have been some announcements that it might not be as difficult to break as was formally thought, but it may still be useful to many people. If you are transferring data within an organization, and can keep the key secret at both ends, then Public Key isn't necessary: it's primary use is where the security of the key at one end isn't known, or must be made public.

Hardware Protection and DES.

So far, we have considered encrypting data while it is in the computer system, and before it is stored or transmitted. This is not the only way it can be done: it is also possible to attach a device to a communications line so that information passing through it is encrypted in one direction and decrypted in the other direction. For example, the device could be attached between your computer and a modem, so that "clear" information being transmitted from your computer will be encrypted before it goes into the modem and out into the world. Most of the special hardware currently offered for sale for this purpose use the Data Encryption Standard (DES), also called the Data Encryption Algorithm (DEA). This method of encryption was developed by the National Bureau of Standards to provide a standard, secure encryption method, and it involves many stages of transposition and substitution. Furthermore, there are several modes for data to pass through the encryption scheme: the method any individual will use depends upon the application. According to the developers, the DEA is intended for use only with hardware encryption schemes for several reasons, two of which are security of operation and verification of correctness.

The first reason includes protecting the key and the encryption method: if it is in special hardware, you have to enter the key into that piece of hardware, and it won't be "floating around" your computer system as it might be if a software program was used. Similarly, only the manager in charge of the special hardware knows what the key is: you don't have individual users losing their keys (or giving them away). In addition, there are often ways for one user to monitor another user's program on the same computer (for example, to watch someone type in their key), and it was felt that it would be more difficult to tap into a separate piece of hardware. With the protection in hardware there is the additional advantage that no-one can forget to encrypt data before sending it out: anything which

is transmitted on that line is automatically encrypted. It was stated before that encryption might not prevent "hackers" or other unauthorized persons from accessing a system, but the one exception is if there is a hardware encryption device placed between the system and the modem which always encrypts the data on that line. Encryption would then prevent unauthorized access, as anyone who wishes to dial in on that line must have an encryption device which uses the same cipher and key. In a similar manner, a hardware device can be placed between a computer and a peripheral device: for example, a disk. If this is done, then all data on the disk is automatically encrypted, and you don't have to worry about users forgetting to encrypt sensitive data, or service personnel reading it during maintenance.

The second reason, that it would be easier to test if the hardware is working correctly than to test if a program is working correctly, is a reason with which I do not entirely agree. It also means that the use of DES would be limited to those applications that can send the data through a line to the special hardware, and that you would have to buy the special hardware for every location which wanted to encrypt data: this meant that locations with personal or small business computers had to buy an encryption device that was as large and as expensive as the computer itself. This is changing rapidly as more large scale integrated circuits which implement the DES are being placed on the market, so that the cost of a peripheral device that does encryption in hardware is decreasing, but it still has many drawbacks for some users. As a result, software houses are offering data encryption programs that use the DES method to encrypt data on the system itself with no special hardware.

DES in the future.

Use of the DES was expected to increase over the next several years, especially where information has to be exchanged between different companies, because it is a standard and it is possible to obtain different pieces of hardware or software which implement it and will still be compatible, as they have to meet the standard to be able to say they use DES; but recently, a snag has developed. Like most modern ciphers, DES uses a numeric key, and there were some arguments about how secure DES really is, based on the length of the key, which is 56 bits (the scheme adds bits to make it 64 bits long). Some of the developers suggested that the key should be 128 bits long, but the National Security Agency required the NBS shorten the key: some critics suggested that a key of this length is such as to be virtually unbreakable by anyone except the NSA itself. Even so, it was expected that the DES would probably be secure enough for most commercial users for the foreseeable future, or at least through 1987, but recently the NSA has been privately telling hardware companies not to put the DES into any new equipment, and to stop using it now. They apparently want to use a new algorithm which will not be made public, ostensibly for better security, but possibly for other motives. In spite of this, the DES will continue to be very difficult for commercial and home users to break, so it will probably continue in use for some time (remember what was said earlier about determining from whom you wish to protect your data).

Additional Precautions.

If you expect a real effort will be made to defeat your encryption scheme, there are a few extra precautions that can be taken to reduce the risk. The easiest way to break a code is if you have a copy of the enciphered message and the clear text together, and can compare the two to work back to the cipher. This indicates that access to important information should be carefully restricted: for example, if encryption is used to protect data during transmission, then when the data is deciphered and safe, the enciphered copy should be erased or destroyed. If it is carelessly discarded, it might give someone a chance to work on it at leisure, especially if the threat is within the company, where the clear text might also be available. Some newspaper codes were broken because the text of an article was transmitted in cipher (by radio, where it could be heard) and then printed word for word the next day in the paper: sending the contents of the article but re-wording it before releasing it to the public solved that problem. Similar precautions could be taken if such things as financial reports are to be transmitted: if possible, don't transmit the data in exactly the same form in which it will be published. In the case of business letters and memos, most start with a date and the person to whom it is addressed, and someone could know (or guess) how the message starts, and use that to cut down the number of attempts needed to find the key to the cipher: one way to stop that is to arbitrarily cut the memo in the middle somewhere, and put the last part before the first. The recipient, after deciphering, can easily see where the real beginning is, and move it back where it belongs. In short:

Don't be predictable.

There are also a few other precautions one can take if you feel that someone is really trying to defeat your encryption scheme. If you think someone is trying to get your key by brute force, you can put random garbage at the beginning and end of your data: anyone who is trying a key and checking only the beginning of the file to see if the data makes sense will not realize it if they do find the right key, as the decrypted data still won't make sense. Of course, anyone can simply check the entire contents of the message for every key tried, but this is much slower, and anything that slows the process of defeating an encryption scheme means the scheme is that much more secure. If there is some reason to believe that whole messages are being intercepted and stored (with some ciphers, the more data you have, the easier it is to find the key, though it might not help much with Infinite-Key, DES and some other modern ciphers), then you should change the key more often than you might otherwise do. In any event, you should not use a given key for too great a period of time, just in case someone is collecting your messages. You can also occasionally send out messages which are the same length and otherwise look like your real messages, but which contain enciphered garbage. The contents (before enciphering) should look as much like real data as possible, without actually meaning anything. This will add to the difficulty of defeating the encryption scheme, but is only worth while if there is a real possibility that someone is making a concerted effort to break the cipher.

Bibliography

There are a number of good descriptions of cryptography in popular literature. In addition to the two examples of the simple substitution cipher given before ("The Gold Bug" by Edgar Allan Poe and "The Adventure of the Dancing Men" by Sir Arthur Conan Doyle), two books by Dorothy L. Sayers (in addition to being entertaining in themselves) are of interest. "Have His Carcass" contains a good description of the Playfair cipher (a good combination transposition and substitution cipher which is easily worked with only a pencil and paper), and a good description on one way to attempt to break it which also clearly shows the hazard of sending messages in a form which allows the content to be deduced. "The Nine Tailors" contains an extremely ingenious example of secret writing. Both are currently published in paperback.

On a more formal basis, the following will be useful:

"Cryptanalysis, a Study of Ciphers and their Solutions" by Helen Fouche Gaines (Dover Publications, Inc.) though written before computers were developed, contains thorough descriptions of many ciphers, and specifically the methods used to defeat them, with worked examples and reference tables. Dover has a mail order department.

"Security and Privacy in Computer Systems" by Lance J. Hoffman (Melville Publishing Co.) treats a wide variety of computer security subjects, one of which is the use of data encryption. It includes a good description of the "Infinite Key" cipher, with a mathematical test of its effectiveness. It also covers operating security, physical plant security, and other subjects.

"Cryptanalysis for Microcomputers" by Caxton C. Foster (Hayden Book Co. Inc., Rochelle Park, New Jersey) Contains explanations of many ciphers, with programs in BASIC to implement them or act as aids in defeating them. The programs may require some work to implement (you have to search through the book to find the subroutines, and sometimes the names of variables change), but some good material is included. The programs are in a simple version of BASIC which most computers should handle as is or with only minor changes.

"Securing Data Inexpensively via Public Keys" by Brian Schanning (Computer Design, April 5 1983, Vol. 22 #4) is an article which describes the mathematics used to generate the two halves of a Public Key.

"The Data Encryption Standard, Recent Controversies" by John E. Hersey, (Telecommunications, Sept. 1983, Vol. 17 #9) gives an encapsulated history of the development of the DES, with some of the arguments for and against its method of implementation and use.

"The Codebreakers" by David Kahn (Macmillan) gives a good history of ciphers (and other data protection schemes such as voice scrambling) and their use, and a description of how some good modern ciphers were broken. The paperback version may be abridged. Considered one of the classic works on the subject.

I have not been able to review the following sources myself, but they may be useful.

"RSA: A Public Key Cryptograph System" by C. E. Burton, (Dr. Dobb's Journal, Mar 1984, 16-21)

"Mathematical Games" by M. Gardner, (Scientific American, 237(2), August 1977, 120-124

The following government publications may also be useful:

"Data Encryption Standard"
Federal Information Processing Standards
Publication 46

"DES Modes of Operation"
Federal Information Processing Standards
Publication 81

Standards Information Office
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

The Smithsonian Institution has a section devoted to cipher machines, and give the following address for inquiries for more information on the subject:

Division of Mathematics
The National Museum of American History
Smithsonian Institution
Washington, D.C. 20560

Eric Newcomer
 Digital Equipment Corporation
 Nashua, New Hampshire

This paper presents an introduction to both types of databases offered by the VAX Information Architecture. The paper includes benefits/features of each as well as a discussion of questions to ask and guidelines to follow when choosing between them.

Introduction

The following is the text of the VAX Information Architecture Databases presentation. This presentation provides introductory information about VAX Information Architecture relational and network model databases -- VAX Rdb/VMS and VAX DBMS, respectively.

This presentation:

- o Defines the terms database and database management system
- o Lists the advantages of using a database management system
- o Compares and contrasts the two VAX Information Architecture database products
- o Provides some guidelines for choosing the database product that is best for your application

Definition of Terms:

o Database

A database is an organized collection of data, usually described independently of the applications that use it.

o Database Management System

A database management system implements a data model that establishes how data can be described and accessed and how relationships among different types of data are represented.

In short, a database management system is software that manages a database.

A database management system transparently handles such programming activities as disk I/O, concurrent file access, and data type conversion.

Why Use a Database Management System?

- o In general, most applications can benefit from using any database management system. A database management system provides for your data:
 - Integrity and security controls
 - Concurrency and consistency controls
 - Centralized definition and storage
 - Easy access

These features are discussed in the following sections.

- o A database management system improves the consistency and integrity of your data. A database management system provides data recovery and data validation routines.
- o A database management system controls the amount of data redundancy, thus eliminating the problem of updating data that is stored in multiple places. A database provides a central location to store and manage your data.
- o A database management system can control which users can access the database, providing additional application security.
- o Another benefit of database management systems is that they increase programmer productivity. Because the database management system performs file access functions, programmers are free to concentrate on application development rather than on data management issues such as concurrent access.
- o A database management system usually includes a straightforward user interface for retrieving, storing, and modifying data and performing simple

data manipulation operations. User interfaces usually fall into two classes -- an interactive query language for examining the contents of the database and testing program logic, and a programming language interface for including data manipulation statements directly in a high-level language program.

- o Most database management systems have integrated procedures for database backup and recovery so that you can save copies of your database at regular intervals and use them to rebuild the database in the event of system failure or database corruption.

Types of Database Management Systems

- o A database management system implements a data model that establishes how data can be described and accessed and how relationships among different types of data are represented. Like RMS files, database files contain data and record definitions, but database files also contain information that describes relationships among the data items and records.

- o The three most common data models upon which database management systems are based are the:

- Hierarchical model
- Network model
- Relational model

- o Hierarchical

A hierarchical database establishes relationships between records as hierarchical tree structures, in a parent-child relationship. Relationships in a hierarchical database can be established only between records logically above or below each other, like in an organizational chart or a VMS directory structure. A hierarchical structure is good for implementing one-to-many relationships, but is not suited for representing many-to-many relationships.

- o Network

A network database establishes relationships between records using sets, where one record is the owner and one or more records are members. Relationships in a network database can be established between any two records in the database, not just those logically above and below each other. A network structure can

implement one-to-many or many-to-many relationships.

A network database contains records organized into set relationships in which one type of record is the owner and another type is the member. These relationships are defined when the database is created and are maintained by the database management system using internal pointers. The pointers are stored along with data for each record that participates in a set. Application programs can then use these pointers to navigate through the database and access the data.

- o Relational

A relational database establishes relationships between records by matching the values of key fields common to both records. Relationships in a relational database can be established dynamically between any two records in the database. A relational database provides more flexibility than either the hierarchical or network model because relationships do not exist as predefined structures. Pointers that link records together are not embedded into the records themselves in a relational database, as they are in a hierarchical or network database.

A relational database organizes data into tables, or relations. Each row in a relation corresponds to a single record, which is made up of fields. Unlike a network database, a relational database does not explicitly define the relationships between different types of records; instead, relationships are established dynamically at run time by selecting and sorting records based on the criteria specified by the user. Therefore there is no need to embed pointers in data.

- o VAX Information Architecture database products implement two of these data models -- the network model and the relational model. Each has its advantages and disadvantages, although one usually is more appropriate for a particular application than the other.

The network model is suitable for a database in which relationships are stable and can be defined in advance, while the relational model is suitable for a database in which relationships change frequently and data access patterns are not always predictable. We will talk more about this later. Right now, let us examine some of the common features of these two VAX Information Architecture database products.

Common Features of VAX Information Architecture Databases

VAX Information Architecture database products are full-function database products. They afford you data integrity by reducing redundancy, controlling user access, validating input, and protecting the database from system failures. They allow users to share data and support simultaneous database access of multiple databases by multiple users from multiple nodes. Finally, they provide utilities to maintain the database and to recover it from hardware and software failure.

Most of these features are what you would expect any good database management system to have. In fact both VAX Information Architecture database products share a common software foundation that implements these basic data management features. VAX DBMS and VAX Rdb/VMS also share other features specific to DIGITAL styles of computing. These common features include:

- Transactions
- Before-image journaling
- After-image journaling
- Concurrency control
- Snapshots
- Multiple databases
- Interactive query tool
- VAX language preprocessors
- Common query language (DATATRIEVE)
- Security controls
- Data validation controls
- Common Data Dictionary interface
- Run on entire range of VAX processors
- Support for run-time environments
- VAXcluster compatibility
- Remote database access
- Part of the VAX Information Architecture

We will discuss these common features in more detail in the following sections.

o Transaction

A series of data manipulation operations is called a transaction. In a transaction, the operations must execute as a unit or not at all. If an error occurs before all the operations in the transaction are completed, the operations already

performed are rolled back to ensure the integrity of the data. The use of transactions guarantees that operations on the database are never partially completed.

o Before-Image Journaling

When a transaction must be rolled back, VAX Information Architecture databases use before-image journaling to undo any updates that have already been made. A before-image journal is a file in which the database management system keeps a record of each transaction before it is committed to the database. Before-image journaling is done automatically.

o After-Image Journaling

An after-image journal is a file that contains images of records that have already been updated. You can use this journal to rebuild a database that has been corrupted by a hardware or software failure. In case of failure, you can recover the database up to the last successfully completed transaction by applying the journal file to a copy of the database that has been backed up on disk or tape. The process of using an after-image journal to replace lost updates is called recovery or roll-forward.

o Concurrency

VAX Information Architecture databases support database concurrency and consistency by allowing multiple users to read, write, and modify data in the database simultaneously. To prevent the introduction of inconsistent data into the database, the database management system allows an application program to protect the records from access by other programs during a transaction. For example, during an update operation, you prevent other users from modifying the same record until your transaction has completed by locking the record.

o Snapshots

Read-only transactions against a database can use a stable, consistent version of the database known as a snapshot. Using a snapshot, you can retrieve data without locking the record from access by other users. If updates are made to the database by other users after you have started a transaction, you do not see the updates. Instead, you see only data that had been committed when your transaction began. For simple retrieval operations where up-to-the-minute information is not crucial, snapshots allow fast performance and a minimum of

contention for records in the database.

- o **Multiple Databases**

VAX Information Architecture databases support multiple databases on a single node. You can also use both database products together in a single application, or in a single program.

- o **Interactive Query Tool**

Both VAX Information Architecture database products supply an interactive database query tool. You can use these tools for ad-hoc database querying and for testing the logic of data manipulation language statements that you later include in an application program.

- o **VAX Language Compatibility**

You can access VAX Information Architecture databases using any VAX language that adheres to the VAX/VMS calling standard.

VAX Information Architecture databases feature a data manipulation language (DML) designed to work with each particular type of database. Precompilers allow you to embed DML statements directly into source code for most VAX languages.

- o **Common Query Language**

You can access both VAX Information Architecture databases using VAX DATATRIEVE. You can use VAX DATATRIEVE to combine data from both types of databases into a single report. Or you can use VAX DATATRIEVE to move data between the two types of databases.

- o **Security Controls**

Access control lists identify which users are allowed access to databases and to records within a database, and what kinds of access they are allowed. You can use access control lists to protect records from unauthorized access and to control the commands and statements that various users can execute.

- o **Data Validation Controls**

The use of validity checks further promotes data integrity by restricting the values that can be stored in a record. If you attempt to enter invalid data, the database management system generates an error message and does not permit the entry.

- o **Common Data Dictionary**

VAX Information Architecture databases

support the VAX Common Data Dictionary, a central repository for data definitions. By storing database definitions in the CDD, you make them available to applications that use other VAX Information Architecture products and to high-level language compilers and precompilers.

- o **Run on Entire Range of VAX Processors**

VAX Information Architecture database products run on the entire range of VAX processors, with the exception that VAX DBMS does not run on the MicroVAX I or the VAX-11/725. In addition, these products run on all hardware configurations, ranging from independent stand-alone systems to the powerful VAXcluster and all DECnet configurations. This range of configurations means that you can configure your system based on today's needs and know that your applications will run on any configuration you may need tomorrow.

- o **Support for Run-Time Environments**

VAX Information Architecture database products feature run-time only kits that allow you to support additional systems on which you wish to run applications but not develop them.

- o **VAXcluster Compatibility**

Operation in a VAXcluster environment means you can access the same database at the same time from all nodes in the cluster, using shared disk files. Automatic database recovery procedures allow users on other nodes to continue using a database when one node fails. Properly configured, VAX Information Architecture databases in a VAXcluster environment can provide virtually uninterrupted database availability.

- o **Remote Database Access**

Remote database capability means you can access databases on remote nodes using DECnet and a server process on the remote node. Automatic recovery procedures roll back the remote database users in the event of network failure. And you can access remote and local databases from the same program.

- o **Part of VAX Information Architecture**

Finally, VAX Rdb/VMS and VAX DBMS are part of a larger family of information management tools known as the VAX Information Architecture. These products improve productivity and enhance the usability of databases by reducing the programming effort required to implement and maintain an application. The VAX Information Architecture products provide you with

all the capabilities you need to build complete applications. These products are:

- A query and report writer that end users can use to read and modify data stored in a database (VAX DATATRIEVE)
- A forms package with a simple record-level interface that reduces programming time through the use of nonprocedural definitions (VAX TDMS)
- An application control and management system that can be used to implement complex transaction processing applications quickly and efficiently (VAX ACMS)
- Graphics products that can produce bar charts, histograms, and pie charts from data stored in databases (VAX DECgraph and DECslide)
- A data dictionary in which to store and retrieve data definitions shared by all products (VAX CDD)

VAX Information Architecture products can be used from most VAX languages, including Ada, BASIC, C, COBOL, DIBOL, FORTRAN, PASCAL, and PL/I.

The integration of these products with the database products improves your ability to design and maintain information management applications.

Unique Features of Each VAX Information Architecture Database

Each VAX Information Architecture database product has some unique features that distinguish one from the other. The following sections describe the unique features of each product.

The Relational Model -- VAX Rdb/VMS

- o To talk about relational databases, one must first be familiar with the terminology. Each "table" in the database is called a relation.

Each row in a table is called a record, or a tuple, and the data items that make up each record are called fields, or attributes. In order to relate the data in two tables, the relations must have one or more common fields or fields of like purpose. For example, if a DEPARTMENT NUMBER field is common to both the EMPLOYEE and DEPARTMENTS relations, you can relate EMPLOYEE data to DEPARTMENTS data based on a specific value in this field. Also, you have the option of defining indexes for relations,

creating an index from fields that are frequently used in data access criteria. The index defines an ordering of record field values that the database management system uses whenever possible to improve the speed of data access.

- o Among the capabilities of a relational database is the dynamic definition of data relationships. This allows users to access data in ways that may not have been thought of when the database was created. Likewise, a relational database can easily be restructured if future applications need new record types or new fields in existing records.

Another capability of a relational database is a set of high-level relational operations to access data. These operations are select, project, and join. We will discuss these in more detail shortly. Another feature is the view, a "virtual" relation that combines fields that are actually stored in several relations in the database.

- o The select operation retrieves all the records from a given relation that satisfy conditions specified by the user. For example, you can select all the records in the EMPLOYEES relation whose department number is 46.
- o The project operation retrieves specific fields from a relation, optionally sorting the field values and reducing them to unique values. For example, you can "project" the department number column in the EMPLOYEES relation, suppressing duplicate values and sorting them in ascending order.
- o The join operation combines fields in two relations based on field values. Conceptually it appears to create a third relation by combining the records in the two relations. For example, the EMPLOYEES and DEPARTMENTS relations can be joined on the department number field to form a conceptual relation that contains all the fields in the two relations.
- o A view is a way of making a permanent definition of the result from some combination of relational operations. It simplifies database usage by eliminating the need for many users to perform the same combination of select, project, and join operations over and over. Moreover, it can be used to control database access if a user is allowed to access data only by means of a view. Thus confidential information can be secured if it is not included in a view definition. For example, a view can be created by joining the EMPLOYEES and DEPARTMENTS

relations on the department number field and selecting the EMPLOYEE NAME and DEPARTMENT NAME fields.

- o The specific components of VAX Rdb/VMS are:
 - A relational database operator utility, RDO, that provides a single interactive environment for database activities: creating and modifying the definitions of database elements, interactive storing and manipulating of small amounts of data during program testing, and maintaining the database
 - Precompilers that support VAX high-level languages, namely BASIC, FORTRAN, COBOL, and PASCAL. A precompiler allows you to include data manipulation statements directly into in your program in virtually the same form you use in interactive RDO.
 - A callable RDO interface for all other VAX languages that support the VAX/VMS calling standard. The callable interface allow you to execute data manipulation statements through imbedded calls to RDO from such VAX languages as Ada, C, DIBOL, and PL/I.
 - A sample database that you can create on your system when you install VAX Rdb/VMS, allowing you to experiment with RDO and study the database design
 - An interactive help facility available from within RDO
- o VAX Rdb/VMS performance enhancements include the automatic optimization of database queries to produce the fastest, most efficient data operations possible. As mentioned earlier, database indexes improve performance by allowing the database management system to sort records based on an indexed field and thus minimize physical disk activity. Another performance gain is the use of sort/merge techniques to speed a join operation that uses non-indexed fields.
- o The database can easily be restructured if changing application needs warrant it, and the user's interface to the product has a simple, easy-to-learn syntax.
- o Constraints placed on a relation restrict the values that can be stored in the relation. You can use the VALID IF clause to test the values being stored against constants, but you can also use database-wide

constraints to check for existence, uniqueness, and nonexistence of values in other fields in the same or other relations.

- o DSRI

Characteristic of all DIGITAL's relational database products is their implementation of a common database architecture known as the DIGITAL Standard Relational Interface. DSRI ensures interface compatibility between applications and all DIGITAL relational product family members, present and future. For the user, DSRI means that applications built on one DIGITAL relational product family member, such as VAX Rdb/ELN, are compatible with other family members, such as VAX Rdb/VMS.

- o DSRI supports large unformatted data types called segmented strings. The database management system stores, maintains, and manages such data without interpreting the structure, leaving the user to analyze the data.

The Network Model -- VAX DBMS:

- o VAX DBMS is an implementation of the network model that complies with the CODASYL model specified in the March 1981 working document of the ANSI Data Definition Language Committee.
- o The network database model is based on establishing relationships among records in a database by the way in which the records participate in sets.
- o A set is defined as a combination of two or more records, one of which is the owner, and one or more of which are members. VAX DBMS provides three types of sets:
 - Indexed
 - Chain (sequential)
 - Calc (hashed key)

Records are stored and retrieved according to the type of set in which they participate as owner or member -- sequentially, according to the value of an index key, or according to the value of a hashed key.

- o Records can participate in more than one set, and can, for example, be a member of one set and an owner of another set. Records are placed into set membership according to their relationships with each other. For example, a DEPARTMENT may consist of a certain number of employees. EMPLOYEE records are identified as belonging to a department by their membership in the set owned that DEPARTMENT record.

Therefore the set of employees belonging to a certain department consists of all EMPLOYEE records owned by a DEPARTMENT record.

- o Each record in a network database includes data items and set pointers. Set pointers contain the database address (or database key) of the next, previous, and owner record in the set. Whenever a record is retrieved, VAX DBMS can use the pointers on that record to directly retrieve other members of the set using the database key.
- o Records in a user's buffers are called current and are used to navigate the database set structure. Entry to database structures is accomplished through SYSTEM records.
- o A record's set membership is determined when the database is created, and in most cases cannot be changed without unloading and reloading the data. When you use a network database, you should be sure of the relationships among your records and data items. If you are, you can take advantage of network database imbedded pointers to speed record retrieval, and to organize your records into complex relationships.
- o In a network database, you can organize your records so that whenever you retrieve an owner record you also retrieve into your buffers all member records for that owner. This can speed retrieval of related records.
- o The specific components of VAX DBMS are:
 - A 4-schema model (logical schema, storage schema, subschema, and security schema)
 - Database Operator (DBO) utility for creating, deleting, backing up, restoring, and recovering databases
 - Data Description Language (DDL) for writing schemas
 - Database Query (DBQ) utility for interactive database querying, including a callable interface
- o VAX DBMS allows you to split your record storage into multiple area files, which you can distribute among your disk devices
- o VAX DBMS provides Load/Unload facilities to help you load and unload your database

Evaluating VAX DBMS and Rdb/VMS

There is no standard answer to the question of whether you should use VAX Rdb/VMS or VAX DBMS for a particular application. Nor are there any hard and fast statistics that clearly delineate trade-off points between the network and relational models. Everything depends on the nature of the application you will be building. You must judge which database product to use within the context of what you are trying to accomplish with your application, and according to your application's particular needs.

However, there are some general guidelines that you can use in making your decision. The following sections present some of these guidelines.

o Training

In general, you need more training to use VAX DBMS than to use VAX Rdb/VMS. Typically, applications implemented using VAX DBMS are long-term, highly-structured applications. The investment in additional training and technical expertise pays off in performance benefits over the long term of the application. VAX DBMS, because of its predefined structure, needs a good initial database design. VAX Rdb/VMS is easier to learn and use, and a poor database design usually can be corrected dynamically.

o Size

VAX Rdb/VMS databases are created and maintained in a single file, while VAX DBMS databases are created and maintained in separate files under control of the user. VAX Rdb/VMS databases cannot as easily be as large as VAX DBMS databases.

A practical size limit on VAX Rdb/VMS databases is 500 megabytes -- or about the size of an entire disk volume. Once a VAX Rdb/VMS database grows larger than a single disk you will have to create your database using a bound volume set.

VAX DBMS databases can be as large as several gigabytes. Separate area and root files allow VAX DBMS storage capacity to be easily spread among several disk devices.

If you need to create and maintain a very large database, VAX DBMS is probably more suitable.

o Complexity

VAX DBMS databases are generally better for more complex data models.

Complex refers to the number of records (or relations) in the database, and the number of relationships among the records.

If you have more than 30 records or more than 30 relationships among your records, consider using VAX DBMS instead of VAX Rdb/VMS.

o **Stability**

If you know that your application or database will not change very much, use VAX DBMS.

In general, it is not worth incurring the performance penalty of using the relational model if you are not going to take advantage of its capabilities to dynamically restructure database relationships.

o **Prototyping**

VAX Rdb/VMS is much better suited to application prototyping than VAX DBMS because a VAX Rdb/VMS database design can be changed so easily. In fact, you may want to prototype your database design using VAX Rdb/VMS and implement it using VAX DBMS once the design is stable.

o **Ad-Hoc Access**

Because the VAX Rdb/VMS syntax is similar to DATATRIEVE syntax, VAX Rdb/VMS databases are easier to work with using DATATRIEVE than VAX DBMS databases. If ad-hoc and end-user access is important for your application, VAX Rdb/VMS may be a better choice.

You may want to maintain a VAX Rdb/VMS database that is a subset of the data you keep in a VAX DBMS database for end-user queries and reports. You could refresh such a database daily or weekly.

o **Unstructured Data**

VAX Rdb/VMS's "segmented string" data type allows you to store and manipulate unstructured data. VAX DBMS does not support the segmented string datatype. The segmented string allows you to mix data types and lengths in a single data item. If unstructured data manipulation is important for your application, VAX Rdb/VMS may be a better choice.

o **Performance**

Database performance depends on what you are using a database for. This includes such factors as:

- The number of database users

- The response time required by application users
- The percentage of update transactions compared to read-only transactions

In general, the more database users you have, the more you will need the performance of the network model. This is especially true when your users are performing many update transactions online, and quick response time is very important.

VAX DBMS provides several performance tuning features and performance evaluation tools to help you optimize database performance.

If you are planning to implement an application that requires high transaction loads per hour, you should probably use VAX DBMS and invest some time and training in learning to properly design and tune a network model database.

There are other ways to increase the performance of your database application. You can use ACMS to use operating system and hardware resources more efficiently. You can also expand your hardware resources using DECNET or VAXcluster configurations.

o **Precompiler**

If you are programming in Ada, C, DIBOL, or PL/I, VAX DBMS may be a better choice because it provides a precompiler for each of these languages. VAX Rdb/VMS provides a precompiler for BASIC, COBOL, FORTRAN, and PASCAL only. You can use VAX Rdb/VMS and VAX DBMS with any VAX language that adheres to the VAX language calling standard, but using a precompiler offers a performance advantage over using a callable interface.

o **Standards**

VAX DBMS adheres to the CODASYL standard for network model databases. If standards are important to your application or your application development shop, you may want to choose VAX DBMS. There are no standards for relational model databases.

Also, VAX DBMS data manipulation language statements are included in VAX/VMS extensions to the COBOL language.

Evaluating Your Application

This section provides some overall guidelines for evaluating your application in terms of the type of file storage method or methods appropriate for it, with emphasis on relational and network databases.

- o At one time, sequential files were the only available method of storing data. Now, however, you have the following methods to choose from:
 - Sequential files
 - Relative record files
 - Indexed sequential files
 - Hierarchical databases
 - Network databases
 - Relational databases
- o In general, the advances in file storage technology over time provide more and more functionality at a cost. The cost is the performance of the file storage system, which takes over more and more of the tasks that previously were performed in application programs.
- o In designing and developing an application you must evaluate various methods of data storage and organization and choose the right one or ones to meet your application's requirements. In making the decision, you have to weigh the benefits of each method against its overhead costs.
- o In deciding between network and relational databases, for example, you should consider the following:
 - Relational technology currently does not perform as well overall as network technology
 - There are applications for which a network database is more appropriate, regardless of performance, than a relational database, and vice versa

We will discuss this last point in the following sections.

- o In general, applications that do not change very often are better suited to VAX DBMS databases. Applications whose data definitions and data relationships change frequently should use VAX Rdb/VMS databases.
- o These different types of applications can be characterized as dynamic and static applications.

- o An example of a static application is an order-entry and inventory-management application. A company does not frequently change the way it processes orders and organizes its warehouses -- it is too expensive. An application that automates this type of business procedure can be characterized as a static application. Such static applications would change only when the company changes the way it does business, which should be infrequent. Static applications are good candidates for VAX DBMS databases.
- o An example of a dynamic application is a sales analysis application. A company might want to spot sales trends to plan for manufacturing capacity, or to evaluate the impact on sales of various forms of advertising. This raw sales data might be available by product, region, sales representative, or customer. Company executives may want to change the method of analysis many times in order to evaluate various factors. Such an application can be characterized as dynamic -- changing it is part of the way the application is used. Dynamic applications are good candidates for VAX Rdb/VMS databases.
- o There are some limits and restrictions that apply to these general rules:
 - If your database is very large -- more than 500 megabytes -- or must satisfy heavy throughput demands of online transactions, you should consider using VAX DBMS. VAX DBMS allows you to partition the database by placing the separate root, area, and snapshot files on different disk devices. This will spread the I/O among disks, increasing performance.
 - If your application development staffing requirements are not completely filled, you should consider using VAX Rdb/VMS. VAX Rdb/VMS is easier to learn and use than VAX DBMS and does not usually require a full-time database administrator.

The Implementation of Academic Faculty and Student
Database Management System

Capt. David A. Gaitros
Robert L. Ewing
Dept. of Computer and Electrical Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

Gary B. Lamont
Dept. of Computer and Electrical Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433
Visiting Professor
Wright State University
Dayton, Ohio

Abstract

The database presented in this paper describes a detailed design effort incorporating a data base system as a management information and decision support tool. It was designed to handle student, faculty and management information in the academic environment. Modular design, prototype development, data type abstraction techniques and emphasis on user-friendliness of the system has resulted in a complete and operational data base. The specific network data base used in this development was the TOTAL Data Base Management System marketed by CINCOM Systems Inc.

The Air Force Institute of Technology, Engineering and Computer Science Department Faculty and Student Database Management System (AFIT/ENG DBMS) has the potential for being a system that contains tens of thousands of lines of code and considerably more data. With a project of this magnitude, an ordinary approach to software development would invite disaster. With the ever increasing cost of software development (6:1) a systematic method was needed to build layers of software that could be tested separately, validated, and implemented over a period of time. This paper will discuss the requirements definition, design and implementation phases of the development in addition to the unique software engineering techniques used to implement the AFIT/ENG DBMS on a VAX 11/780, VMS system and a commercial DBMS (TOTAL).

Requirements Definition

The requirements definition phase is the most critical and time consuming portion of the Software Development Life Cycle (1:13). The critical element of this specific database implementation involves student; and faculty and their associated interrelationships including class enrollment, grades, professor, schedule, educational plans, class advisement, thesis/dissertation advisement, and personal information. Despite the amount of time and effort spent on obtaining system requirements (2,3) another extensive requirements definition phase was conducted to update the associated document. This document consisted of an

on-line list and tables. To aid in defining the requirements, a fast prototype of a major sub component, the Education Plan program, was developed. The program was presented to prospective users of the system and their likes and dislikes were noted and analyzed for completeness, consistency and validity. Appropriate items were translated into specifications and used to augment the requirements document. The requirements were then divided into the functional requirements and user-computer interface requirements.

The functional requirements specify what functions the system is required to perform. These included database transactions, management information displays, required reports, space specifications, and program language specification.

The user-computer interface requirements are those traits of a computer system that make it "user-friendly" to a typical user of the organization. The "typical user" was defined and the interface characteristics were designed for a person with the following traits:

1. Has run a word processor, computer or typewriter.
2. Will use the system on a casual basis.
3. Will have limited access to manuals.
4. Will not know many of the abbreviations used in the database.
5. Will not have access to social security numbers or record keys in general.

The definition of the typical user led to the design of a menu driven system with on-line help capability. With a menu driven system, the user would not have to memorize a complex command language or have need of a users manual to operate the system. The casual user could operate the system with little guidance and a short leaning period. The on-line help feature was provided by a commercial forms management system (FMS) which acts as the input/output media to the user. (5)

It was apparent early in the requirements phase that the system would have a long life and would require a long maintenance period to implement all of the design. To limit the amount of code necessary to program the system after the design, a set of standard routines were required that would act as abstracted routines to the database system TOTAL (4). Examples of some functions include routines that format records, write records, provide error checking, check database status, and sign on and off of the DBMS. Additional routines were needed to maintain lists of students and faculty members that could be searched and sorted in a short period of time for access to detailed information.

Design Phase

The design phase initially reviewed the characteristics of the requirements as they pertained to development of the system. The system requirements were decomposed into eight distinct components. The modularity specification required a structured development of the database software and its interaction with these components. The high level design was defined using hierarchical structure charts and module narratives (7). A layered software organization reflected this approach to the problem (figure 1). Layer 1 of the system would be the very top

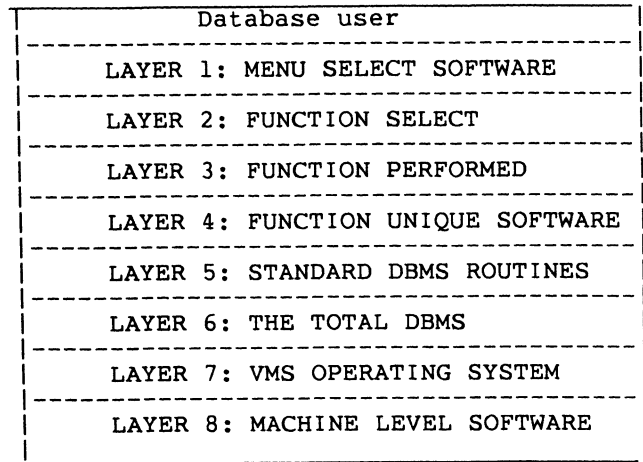


FIGURE 1: SOFTWARE LAYER

level controller and would control the entire database, calling all of the five main modules (figure 2). The second layer of software would provide access to different areas of the database that would correspond to student related file, faculty related file, thesis related files, etc. The third layer of software would contain the main function calling routines that control the procedures necessary to perform the function. Layer 4 are software procedures called by the layer 3 software and are unique to the related function. Layer 5 is the standard set of routines developed in the requirements definition phase and can be called by any of the above layers. Layer 6 is the TOTAL DBMS and runs in the batch mode on machine and should only be called by layer 5 modules. Layers 7 and 8 are the machine dependent code and hardware. Note that layer 6 is the abstracted DBMS and thus other DBMSs could be interfaced at this level without impacting the other layers.

Implementation

The coding and testing of the application software for the AFIT/ENG DBMS encompassed the implementation portion of the software development. In order to demonstrate and provide prototype user interaction, the lower level routines (layer 5) were coded and tested during the last phase of the requirements definition and the beginning of the design phase. This code accounted for 40% of the actual Pascal code needed to develop the Education Plan program. Note that Pascal was chosen as the implementation language because of its ability to implement some of the principles of software engineering such as data abstraction, type declarations, modularity and information hiding as well as the previous effort (2)

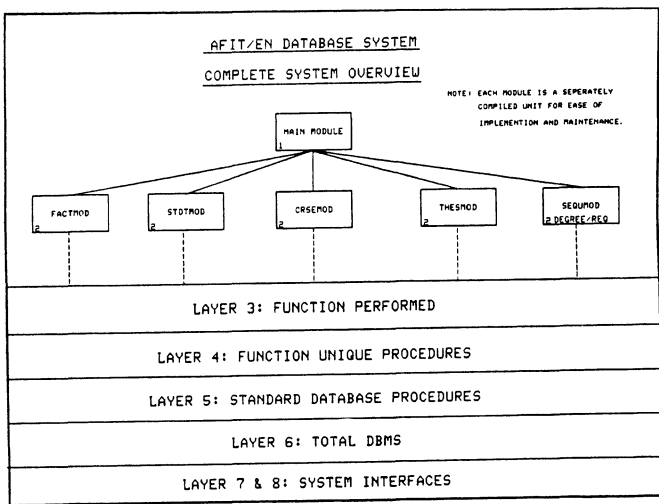


FIGURE 2: FIVE MAIN MODULES

The standards employed at this phase were associated with documentation (8), declaration naming, coupling and cohesion, modularity and data abstraction (9:328). Resulting from this effort was a consistent structure embodying the following major routines: EDPLAN, FACTMOD, CRSEMOD, SEQUMOD, BOOKMOD and THESMOD. The EDPLAN module maintained the students education plan for the faculty and students. The FACTMOD module maintained the faculty members personal and professional data. The CRSEMOD module maintained the course data used by the EDPLAN module. The SEQUMOD maintained the valid course sequences students were allowed to take. The BOOKMOD module maintained the data on required text books for each of the courses. Finally, THESMOD module maintains the data on past and current theses efforts.

To illustrate the complexity in this phase of the development, the actual coding and integration of the EDPLAN module is discussed. The standard routines and type declarations from the design phase were coded and defined as EDPLAN.PAS. The main routine was coded first. Stubs were insert for the layer 3 procedures to permit testing. The main program was tested to insure all routines interfaced properly and according to design specifications. One by one the layer 3 and 4 procedures were added and tested. Once this was completed, the complete EDPLAN program was demonstrated to various individuals and used on a limited basis. The program was modified to accommodate minor enhancements and a number of minor errors were corrected. It should be realized that this process of meeting user objectives is an on going activity, thus it is very important that extensive documentation be defined for each phase to facilitate maintenance.

Conclusion

The software engineering methods of data abstraction, data hiding, modular structure, fast prototyping, and software layering contributed a great deal to the development of the AFIT/ENG DBMS . In this paper, the EDPLAN component was presented in detail to illustrate the above concepts. It is only a small example of the vast amount of software needed to fully implement the design, but it acts as a template to future programmers on the method of implementation. The software developed by this project was meant to be modified and was written with the maintenance programmer in mind. In this regard, a major part of this approach is the configuration and control of the documentation associated with the requirements phase (list and tables), design phase (structure charts and module narratives), implementation phase (commented code and users manual) and the integrated and comprehensive data dictionary for all phases. The use of various tools such as FMS, TOTAL, CDD, GKS, SDW (ref), and Pascal can help provide this type of documentation and make large database development easier. If the systems analyst and programmers assigned to continue this effort adhere to the principles and practices set down by this example, the AFIT/ENG DBMS will develop into a highly flexible and responsive system for the AFIT School of Engineering or for any academic database environment.

Bibliography

1. Peters, Lawrence J. Software Design: Methods and Techniques, Yourdon Press, New York, 1981
2. Pangman, Myron E. Complete Development and Implement AFIT/EN Database Management System, Masters Thesis, School of Engineering, Air Force Institute of Technology, Air University (AU), Wright-Patterson AFB, Oh, 1983
3. Allred, Dean S. Consolidated AFIT Database, Masters Thesis, School of Engineering, Air Force Institute of Technology, Air University (AU), Wright-Patterson AFB, Oh, 1980
4. Cincom Systems, Inc. TOTAL User's Guide. Digital Equipment Corporation, Canada, 1980
5. Digital Equipment Corporation, VAX-11 FMS Software Reference Manual Order No. AA-J260A-TE. Digital Equipment Corporation, Maynard, Ma, September, 1980
6. Myers, Ware. The Need For Software Engineering, Computer, February 1978
7. Gaitros, David A. Implementation of the AFIT/ENG Student and Faculty Database Management System, Masters Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, Oh, 1985
8. AFIT/ENG Development Documentation Guidelines and Standards, Draft #2, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, Sept 26, 1984
9. Stevens, W.P., Myers, G.L., Constantine, L.L. "Structured Design", IBM Systems Journal, Volume 13, Number 2, 1974

Lisa M. Rotunni
 Edward C. Hohmann
 James A. Rounds
 School of Engineering
 California State Polytechnic University
 Pomona, California 91768

The School of Engineering at California State Polytechnic University, Pomona employs 120 regular and 185 temporary faculty members, and 50 clerical and technical staff persons. A data management application, SAPS (School Academic Personnel System) has been written in the database management program RDM to handle all the administrative, personal, and workload information for these faculty and staff members. The use of database technology insures consistency of many dissimilar reports and provides the ability to tabulate information in unique ways. This paper will describe the planning, programming and operation of SAPS, along with the administrative implications of on-line access to critical information.

OVERVIEW

California State Polytechnic University, Pomona has the largest School of Engineering in California with some 4,300 full and part time students. The administration of the School has to keep track of 120 regular and 185 temporary faculty members, and 50 clerical and technical staff persons. In order to handle this task quickly and efficiently, the School Administration decided in the Summer of 1984 to begin implementation of an office automation system. This system would include word processing, spreadsheet facilities, and database technology capable of handling employee administration information, course and facility scheduling and student information.

As the first step toward achieving this automated office, the School obtained a computer system consisting of a PDP 11/23 processor with 1 megabyte of memory and a 76 megabyte winchester disk, 7 DEC VT102 terminals, a dot matrix printer and two laser printers. The total value of this system is under \$25,000. Use of the RSX multi-user operating system with this collection of hardware allows us to keep two secretaries busy using the database system, entering and reporting information, and one programmer developing and enhancing data management applications, with spare terminals available for word processing and other intermittent tasks.

The word processing and spreadsheet capabilities desired were effective almost immediately with the purchase of packaged software. A database management system was also purchased, but unlike the word processor and spreadsheet, it required custom programming in order to fill our needs. Therefore, we chose to start on our most critical need, an employee administration application.

Why use a database system? First, it allows us to have all of the information needed in one place at one time. We can go to the same place to find an employee's home address and the year he received his bachelor's degree. Second, the information we give out is consistent. The information only has to be correct in one place, and if diligence is used in keeping it up to date we can be certain of its accuracy at any given time. Then, whether we're printing the payroll or a phone list, everyone's name is spelled the same, correct, way in both places. Third, information can be arranged in unique ways. Until the need arises, it is hard to imagine why anyone would want mailing labels sorted by office phone extension. But the need arose when we had to distribute campus phone books, and we were ready.

SAPS (School Academic Personnel System), the database system developed for the School of Engineering at Cal Poly University, Pomona comprises the administrative, personal, and workload information for the faculty and staff of the school. It allows us to produce reports like the on-campus location directory, the payroll list, home mailing labels, and a list of what companies the part-time faculty members are affiliated with.

SAPS is an applications package written within the commercial data base management system RDM (Responsive Data Manager), produced by Interactive Technologies, Incorporated. RDM provides the framework for the system through a series of PASCAL programs which handle the actual file manipulation and data input and output functions. The programmer working in RDM deals with a programming applications package to define data files, create data input screens, write reports, and set up menus for a specific application.

The SAPS package is designed to be a time saver. Through menus the user can add new data, edit old information, and print all or portions of the information in many different formats. SAPS doesn't really do anything that wasn't done before in some fashion or other. But SAPS makes it possible to do the work more easily, accurately and reproducibly.

Perhaps the most impressive feature of SAPS is the constant on-line availability of information. Any information or grouping of information that anyone wants can be produced quickly and efficiently, provided the general type of request was anticipated properly. And if it wasn't, as long as the information is in the system and the programmer is available, it just takes a little longer.

PLANNING THE SYSTEM

This is the tricky part. Ideally, the whole puzzle should be fit together before attempting to program the required system. However, the long lead time often required to accomplish this is not always available. It is sometimes necessary to use the system and modify it at the same time. This real time approach can be an advantage, since it makes it possible to try it, see if you like it, and then try something else if you don't.

The planning of SAPS was and is an ongoing function. The product SAPS described here is the final form of three distinct attempts at a computerized data management system for employee administrative information. The first attempt was made at the departmental level using programs in BASIC and sequential data files. This approach was severely limited in that these programs were written for the convenience of a single programmer and could not readily be used by anyone else. Data entry was performed with a text editor and was tedious and frequently inaccurate.

The next level of complexity in data management was reached when a systematic attempt was made to determine the administrative information and products required and to write a unified program package to meet these needs. These programs were still written in BASIC, but were designed for easy use and future programming. A menu system was also set up for program access. Data entry was still performed with a text editor, however, and this became a significant difficulty as the amount of information which needed to be stored became larger.

At this point we moved to the school level and invested in a commercial database system. In a package system the complex task of file interrelation is automatically performed. The RDM system is designed for ease of data entry and with the proper hard disk storage, large volumes of information can be stored and quickly accessed.

The most critical part of a database system is the data structure: the actual data elements which are going to be stored and relationships between them. If the data structure is well designed then the reports and user conveniences can be developed as the need arises. Effective data structure design requires an absolutely thorough understanding of where the information comes from, what people do with it, and where it goes to when they're finished. Gathering this information can be difficult, especially for the programmer who is not familiar with the particular administrative system which is being computerized. It is hard to find the right

questions and people to ask. The person who has been performing some task can't always describe the details of what they do and how they do it or what information they want when. However, even when planning time is in short supply, don't cut corners in designing the data structure.

In designing the data structure for SAPS we first looked at the more obvious reports that were being generated by the administrative and clerical staff of the School and determined what information they contained. For the Location Directory we needed to know employee's names, departments, offices and phone extensions. To print the payroll report we needed social security numbers, position numbers and teaching load. We needed home addresses for home mailing labels. After thinking of as many items as we could, and categorizing them according to type, a group of data files were defined to hold this information and a secretary was trained to enter it.

Once we had on-line access to the vital statistics of each faculty and staff member, we could produce the reports we needed. Then the fun began. In the further development of SAPS it was planned, programmed and used simultaneously in the environment in which it was needed. This real time evolution of the system permitted a constant realistic evaluation of its effectiveness and responsiveness.

In an effort to categorize the structure and function of SAPS, the information can be broken down into four types, each leading to a different variety of reports. The personal and location information within SAPS allows the production of such reports as a directory of faculty and staff offices and phone numbers, a social directory of employees' home addresses and phones, and home mailing labels. The workload information facilitates keeping track of the number and type of units being taught during the year and the amount of money spent on faculty salaries. The position information yields a roster of faculty positions and their funding, and the department information may be used at any time to identify the various names, numbers and other codes associated with the administrative and academic departments.

The basic functions which SAPS needed to perform may be categorized as well. Taking information from hiring forms, payroll charts, university administrative documents and other places and putting it into the system is considered an information management task. Also included in management is the purging of outdated or incorrect information.

Projection tasks involve the production of information which aids in intelligent decision making about some future event. The faculty workload report can be used as a projection tool because it shows how much will be taught in future quarters and aids in planning. The faculty roster provides salary information and allows the projection of spending for the academic year.

Because of the School's intermediate position between university administration and the academic departments, many of the tasks which must be performed can be described as auditing functions. For example, payroll sends a list of who is being paid out of what fund. The school office must certify that the employee is actually working, report time off and identify any discrepancies or errors. The approach taken in SAPS to auditing

outside information is to print the same list from the information within SAPS and then manually compare the two lists. The information within SAPS is known to be accurate, and therefore discrepancies between the two sources of information indicate an error which must be corrected.

These categories of information and tasks facilitate clear thinking about what the system does and what it should do in the future.

PROGRAMMING THE SYSTEM

This is the critical part. If the programmer knew exactly what was needed right from the beginning, it would be a breeze. If it were really possible, all planning should be complete before programming begins. But since having a system sooner is usually better than later, and something is usually better than nothing, programming may need to commence before planning is really finished. And in dealing with people who have done administrative tasks by hand, it is often efficient to show them what can be done and see how they like it.

The main programming tasks involved in the development of SAPS were defining the data file structures, describing the data input screens, creating the menus which facilitate system use, and writing the reports which print or display the data in a variety of formats. RDM provides convenient methods for accomplishing all of these tasks.

In RDM, data files are formed of a collection of fields. The programmer defines the fields, giving each a name and data type. The group of fields which is repeated within the file is called a record. For example, in the SAPS employee file eighty-three separate fields have been defined. Items like surname, teaching department, home address and job title each have their own field. The group of fields, all related to the same employee, is designated a record. If there are forty employees in the file, there will be forty records. Data types available in RDM include real numbers, integers, alphanumeric fields called strings, boolean (yes/no) fields, dates and times.

Input screens are described in RDM by entering into a form control table the name of the data field and its desired location on the screen. RDM then uses this file to display records on the screen and to accept data into the fields of a record. Additional headers, boxes and other special items may also be displayed using special commands. Menu screens are defined in a similar fashion. Command names are entered into a menu command table, along with instructions on what the results of entering each command should be. In effect, RDM is programmed using RDM. The command tables are accessed and edited by the programmer in the same way that data files are edited by the users.

The end result of most information manipulation tasks is a report. Some group of data items is printed, either to a terminal screen or on paper. In RDM the specifications for reports are defined in Report Control and Report Format Tables. Through an RDM internal programming language, the programmer can determine what data files will be used, what fields will be reported, the field or fields by which the output will be sorted, and the appearance of the final document. Reports are defined by entering values in fields which appear on report definition input screens.

Each report must have a primary data file, which is the main source of information for the report. The records in this file are usually processed in turn and therefore the file is usually sorted to obtain the final order for output. Other files can also be accessed in the report and information to be printed can be taken from them as well as the primary file. The records in these secondary files are located and processed due to their interrelationship with the primary file.

Almost any final report format is possible through the report definition in RDM. Any information in the primary file and up to three secondary files can be printed, plus additional supporting text, headings and other information. Several examples of SAPS reports are provided in Appendix I.

USING THE SYSTEM

This is the fun part. This is where the feedback comes from, the critique of what has been produced by all the planning and programming. It isn't until the system is actually being operated by the people who need it that anyone can really determine its advantages and shortcomings. And no matter how much time and effort was spent planning, once the system is running the users always want something extra, or something else.

In its current and more or less finished state, SAPS is made up of several data files which allow for distinct groupings of information. The employee file is by far the largest. All of the SAPS information relating specifically to each employee is listed in a record for that employee. This information is taken from hiring forms, personal information provided by the employee, payroll forms, information provided by the departments, and other administrative sources. Because of the large size of the total employee data, three separate files are used for data entry. Although identical in structure, they permit the grouping of employees into the three distinct categories which affect the nature of the data entered for each employee. Separate data files are used for regular faculty, temporary lecturers and staff. The combination of these three files makes up the employee file.

The faculty positions file is not organized according to people but according to the university's construct of positions. When a new faculty member is hired there must be a position open to hire him into; when a faculty member leaves a position becomes open, and that position may be filled by hiring another faculty member. The staff positions file is also organized according to positions. In addition to position information, it also keeps track of vacation and sick time used and accrued by the staff members in those positions.

The salary information file contains the dollar amounts associated with each salary rank and step. Thus, given the faculty member's salary schedule, rank, step and number of teaching units from the employee file, it is possible to calculate his salary for the quarter from the salary information. The departmental file contains the abbreviations, full names and other identifying codes associated with the various administrative and academic departments. It serves as a dictionary to which reports using the other data files may refer for more information.

In SAPS data entry is performed through input screens, which display the field names and blank lines of the correct length for the data item named. Arrows and other commands may be used to move from record to record within the file and from field to field on the input screen. Data is added or edited in the fields by positioning the cursor in the field and typing the new data. Each SAPS data file has its own input screen.

The primary file for SAPS reports is usually one of the standard data files, most frequently the employee or positions file. However, occasionally it is necessary to work with some data grouping which is not provided for in any of the existing files. In that case, the data must be processed before it can be reported. An example of this situation is the list of employee degrees and the institutions at which they were received. The employee data contains one record for each employee and this record lists all the employee's degrees. This data format made it impossible to produce a final report sorted by the institutions at which the degrees were received. Therefore, it was necessary to create an intermediate file which had one record for each degree for each employee. The ability to pre-process data is very useful. However, it does increase the time necessary to produce a report and should be used only when necessary.

Normally, RDM reports print information from all of the records within the primary file for the report. However, it is possible to limit the records which will be reported through either programmed or user selections. Programmed selections are defined into the report by the programmer. For example, no staff should appear on the faculty roster. Therefore, within the Report Control Table, the report has been instructed to skip over any staff records in the primary file when it prints that report. This programmed selection can not be changed by the user and applies every time the report is printed.

User selections may vary with each report printing. Through the selection mode, displayed before the report is actually printed, the user may limit the records to be considered for printing by defining a range of values for specific data fields. These selections then apply in addition to any programmed selections. For example, if the faculty roster was desired for only the Electrical & Computer Engineering Department, the user could specify that the 'Department' field of the employee file be equal to 'ECE'. The report would still skip all staff records and, in addition, it would now skip any record which did not have 'Department' equal to 'ECE'. Multiple selections may be made to further limit the information printed and selections may be based on any field in the primary data file. User selections greatly enhance the flexibility of SAPS by making it possible to print almost any grouping of desired information in any of the standard report formats.

One of the requirements for SAPS was that it be comfortable for use, not just by the programmer but by the faculty, staff and administrators who would be working with the system. This requirement is met through the use of menus, help files and user documentation.

The most prominent SAPS feature seen by users is the menu. The SAPS Menu consists of six screen pages full of commands which can be displayed in a continuous loop. The last command on each menu page

displays the next menu page. The commands on the menu pages are divided according to categories of information. The first page contains the various directory reports available in SAPS. The second page allows the printing of reports which are of interest in departmental administration. The third is devoted to school administrative reports. Employee information is added and edited on the fourth menu page and other files are edited on the fifth page. The last page lists the commands which perform various information checking and auditing functions.

The menu commands are easily accessed through the keyboard arrow keys, or by typing the command. The user does not need to know any file or report names in order to use the system.

Each command line on the menu lists a brief description of the function of the command, which is generally only enough to jog the user's memory. If the user is unfamiliar with the system or needs a more complete description, help information is available directly from the menu by pressing the help key on the numeric keypad. For reports, a complete description and example of the print format is displayed on the screen. For editing and checking functions, other appropriate information is provided. When the user is finished reviewing the help information he may return to the menu.

When editing or adding information to the data files of the system, help text is available for each record field. By positioning the cursor at the field for which help is desired and pressing the function help keys, the user can access a screen of text describing the field and listing the type and range of information which should be entered.

All of these menus and help screens are designed to allow convenient use of SAPS without the need for memorizing many commands or remembering exactly what each field may contain from one editing session to the next.

A complete user manual for SAPS is available, written so that someone who is not familiar with RDM and has never used SAPS before can enter the system and print reports or perform data management tasks.

This fall was the beginning of the second academic year that SAPS has been in use in the School of Engineering. Therefore, we were able to see how the data editing functions and reports developed and programmed as they were needed last year filled our ongoing needs. New information entry went smoothly, the system functioned effectively without modification, and no additional reports were needed. In academic terms, SAPS passed its final exams.

The success of SAPS has encouraged the development of other complete systems. The student Requests And Tracking System (RATS) is currently in use to keep track of change-of-major requests, general academic petitions and other student related items. The School Instructional Planning System (SIPS) is currently being developed to handle course and room scheduling.

APPENDIX I - SAMPLE REPORTS

CALIFORNIA STATE POLYTECHNIC UNIVERSITY, POMONA
SCHOOL OF ENGINEERING

WINTER 1985 LOCATION DIRECTORY

Aerospace Engineering

Sutherland, Rodney D. Professor and Chair	13-229	4301
Bowles, Stephanie I. Dept Secretary	13-227	4301
Howard, Rollen D. Equip Technician	13-119a	4306

Faculty

Davey, Robert F. (Dr.)	13-223	4304
Graves, George R.	13-223	4304
Hudson, Terrance J. *	13-101	0242
Lehr, Mark E. *	13-101	0242
Lord, Paul A.	13-225	4303
Mardis, Larry D. *	13-101	0242
Mortensen, William E.	13-101	0242
Newberry, Conrad F.	13-225	4303
O'Cain, Brian D. *	13-106	4348
Rickard, William W. *	13-106	4348
Schoneman, Scott R. *	13-106	4348
White, Terry *	13-101	0242

* P/T Lecturer # F/T Lecturer + Retired (FERP Qtr)

List of Employee Degrees and Institutions

24-APR-85 Name	Dept	Type	Institution	Degr	Discp	15:05:24 Year
Darweesh, Farouk	IME	N	Bolton Inst	GCE	Mechanical	64
Baher, Farrokh	ECE	P	Cal Poly Pomona	ME	Engr	78
Epperson, Jr., Edwin H.	ECE	P	CSU Los Angeles	BS	Math	62
Epperson, Jr., Edwin H.	ECE	P	CSU Los Angeles	MA	Math	66
Schoenwetter, Earl E.	ET	T	CSU Los Angeles	MS	Industrial	75
Janger, Frank J.	CE	T	Manhattan College	BCE	Civil	68
Janger, Frank J.	CE	T	Manhattan College	ME	Sanitation	69
Hohmann, Edward C.	EGR	T	Michigan State Univ	MS	Chemical	67
Rubinstein, Eli	ECE	P	Pomona College	BA	Physics	61
Eke, Fidelis O.	ME	P	Stanford Univ	PhD	ME	79
Williams, Edwin H.	ME	T	UC Berkeley	BS	Mechanical	49
Darweesh, Farouk	IME	N	Univ of Bermingham	BS	Mechanical	67
Darweesh, Farouk	IME	N	Univ of Bermingham	PhD	Mechanical	70
Galbraith, Edward D.	IME	T	Univ of Toledo	BSME	Mechanical	52
Galbraith, Edward D.	IME	T	Univ of Toledo	MIE	Industrial	60
Rossman, Edward A.	ET	P	Univ of Washington	MS	Aero	49
Schoenwetter, Earl E.	ET	T	Univ Of Wisconsin	BS	Electrical	57
Hohmann, Edward C.	EGR	T	USC	BS	Chemical	66
Williams, Edwin H.	ME	T	USC	MS	Mechanical	66

CAMPUS LOCATION LABELS

(A similar format is used for home address mailing labels.)

Mr. Robert R. Schneider ^
Civil Engineering
Rm 11-114 Ext 4321

Mr. Robert R. Schneider
Civil Engineering
Rm 11-114 Ext 4321

Mr. William M. Harris
Chemical & Materials Engineering
Rm 13-116 Ext 4314

Mr. William M. Harris
Chemical & Materials Engineering
Rm 13-116 Ext 4314

Dr. Thuan K. Nguyen
Chemical & Materials Engineering
Rm 13-226 Ext 4364

Dr. Thuan K. Nguyen
Chemical & Materials Engineering
Rm 13-226 Ext 4364

Dr. Robert L. Bernick
Electrical & Computer Engineering
Rm 09-415 Ext 4330

Dr. Robert L. Bernick
Electrical & Computer Engineering
Rm 09-415 Ext 4330

Dr. Mysore R. Lakshminarayana
Electrical & Computer Engineering
Rm 09-415 Ext 4330

Dr. Mysore R. Lakshminarayana
Electrical & Computer Engineering
Rm 09-415 Ext 4330

Dr. Edward C. Hohmann ^
School of Engineering
Rm 09-227 Ext 4311

Dr. Edward C. Hohmann
School of Engineering
Rm 09-227 Ext 4311

Mr. Earl E. Schoenwetter
Engineering Technology
Rm 09-246 Ext 4801

Mr. Earl E. Schoenwetter
Engineering Technology
Rm 09-246 Ext 4801

Mr. Leonhard M. Myers
Industrial & Manufacturing Engr
Rm 11-128 Ext 4369

Mr. Leonhard M. Myers
Industrial & Manufacturing Engr
Rm 11-128 Ext 4369

SQL/DSRI and QUEL/DSRI Implementation

JOHN D. MARKEL, Ph.D.
SIGNAL TECHNOLOGY, INC.
GOLETA, CA 93117

Abstract

This paper describes the first Digital Standard Relational Interface implementation of the two de-facto Standard Relational Query Languages. The Structured Query Language (SQL) is rapidly becoming the industry standard. It is the query language syntax used by commercial products such as IBM's DB2. The QUEL syntax as developed for the Berkeley Ingres project is also a widely used, commercially available syntax. In this paper, issues are presented which relate to what can and can't be efficiently implemented through the DSRI. Syntax comparisons among implementable SQL and QUEL commands and the Rdb/VMS interactive query utility (RDO) are presented along with actual performance results.

1 INTRODUCTION

1.1 Overview

In the VAX/VMS environment, there are three widely used relational languages. The language currently available for Digital VAX Information Architecture products is the Datatrieve language. With only minor variations, the Datatrieve syntax is what is used in Digital's relational database products Rdb/VMS and Rdb/ELN.

Two other widely used languages in the VAX/VMS environment are QUEL and SEQUEL (or SQL). The QUEL language was developed at the University of California at Berkeley as part of the Ingres project. Ingres has been converted into a commercial product by Relational Technology, Inc., using its own proprietary database file structures under VAX/VMS.

The SEQUEL (or SQL) language (which stands for Structured Query Language) was developed at the IBM Research Laboratory in San Jose, California. In 1977, the public domain specifications of SQL were implemented by Oracle Corporation as part of their commercial product, using its own proprietary database access methods under VAX/VMS. IBM's new relational product DB2 is also based upon the SQL relational query syntax. This syntax has been proposed as an ANSI standard and is rapidly becoming at least the de-facto standard relational query language in the computer industry.

Meanwhile, Digital Equipment Corporation has developed the Digital Standard Relational Interface (DSRI) which is the foundation of its current relational database offerings, and of its future offerings (such as RALLY and TEAMDATA, discussed by Digital at the Fall 1985 Decus Symposium). In effect, the VAX/VMS file management standard (RMS) is being joined by a new relational database management standard (DSRI).

Neither RMS nor DSRI have any inherent language syntax associated with them. They define only the "backend" file or database file structures.

1.2 Purpose of This Paper

Digital's first VAX/VMS database product introduction is Rdb/VMS which uses the Datatrieve-like syntax. Since QUEL and SQL are widely used relational languages not only with VAX/VMS but also with many other computer systems, we decided to implement QUEL and SQL "bridges" into the Digital standard RMS and DSRI backend structures as illustrated in Figure 1. The development for QUEL has been completed and is offered by Digital as part of the Digital Classified Software product SMARTSTAR. An SQL development is ongoing.

Signal Technology is the first company (outside of Digital) to successfully develop a commercially available interface to DSRI. In this paper, we would like to share some of our observations about using the DSRI for implementing QUEL and SQL language structures along

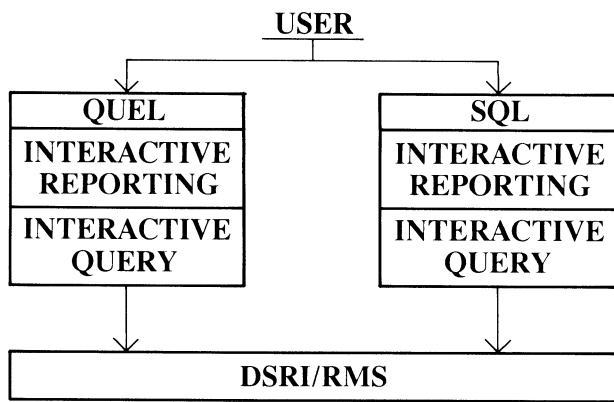


Figure 1 - Unified Bridge FOR QUEL/SQL With Both DSRI And RMS

with performance measurements. Although parallel language developments have been undertaken with the RMS file structures for consistency, only the DSRI implementations will be discussed in this paper.

The presentation is organized as follows:

- First, Rdb/VMS is briefly introduced and its relationship with DSRI is illustrated. Relative performance among Rdb/VMS components is shown.
- Next, the DSRI itself is briefly introduced and several observations about complexity versus benefit are presented.
- The QUEL/DSRI implementation is discussed along with performance measurements.
- The SQL/DSRI implementation is then discussed along with certain complexities that arose.
- Finally, a brief summary of our experiences is presented with recommendations.

It is assumed that the reader is already familiar with QUEL and/or SQL for this presentation.

2 RDB/VMS AND THE DSRI

2.1 Introduction

The Digital Standard Relational Interface (DSRI) was introduced by Digital at the Fall 1984 Decus Symposium. To date, a small number of organizations have been given access to the DSRI programming specifications for developing DSRI compatible third party

software products. Digital has stated that the DSRI specifications will be made available during 1986 (DSRI is already available to those who have Rdb/VMS, since Rdb/VMS is in effect a layered product on top of DSRI.) What is missing are the programming language specifications for directly accessing the DSRI instead of indirectly accessing DSRI through the Rdb/VMS facilities.

At the current time Digital has three products which can access the DSRI: Rdb/VMS, Rdb/ELN and Data-trieve (DTR), as illustrated in Figure 2. All local and remote data access is handled through the DSRI.

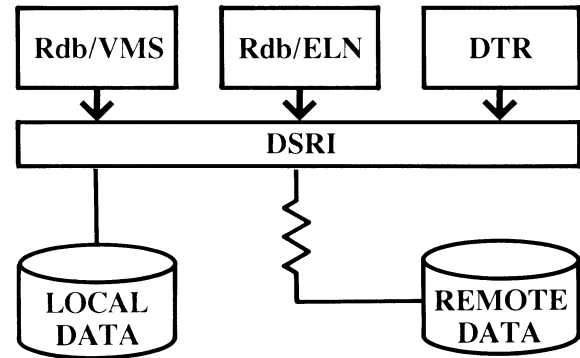


Figure 2 - Digital Products Which Are Layered On Top Of DSRI

Within the product Rdb/VMS for example, there are several components which individually access the DSRI as shown in Figure 3.

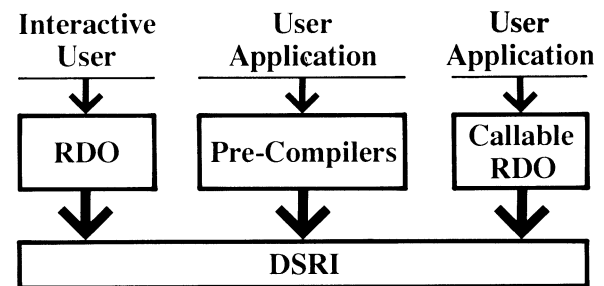


Figure 3 - Individual Rdb/VMS Components Which Access The DSRI

This diagram shows the three important methods for data access within a relational database system.

- Interactive access
- Compile time binding access (preprocessor, pre-compiler, embedded query access)
- Run-time binding access (callable or run-time access)

Each of these methods has associated advantages and disadvantages. Interactive access is very convenient for performing random ad-hoc query. In addition, all Data

Definition Language (DDL) activity must occur here (through RDO). Whenever a query is executed, it is interpreted as though it is a brand new query each time. For example, if you wish to find the count of all records where salary is greater than 20K, 30K and 40K in sequence, three distinct queries must be run, each totally independent of the other.

Compile time binding on the other hand "binds" the database variables to the VAX host variables, and also parses the query into a set of tokens which can be read by DSRI at run-time. Thus if the above query is placed into a loop with SALARY as a variable which increments over the desired range of values, no additional reparsing of the command is necessary. Thus the Rdb/VMS pre-compiler offers two primary performance advantages:

- At run-time, the user query has already been decomposed into a set of low-level DSRI calls.

and

- If the query is in a program loop, then it can be re-executed without additional time required for parsing.

Turning these statements around, it is important to note that if ad-hoc queries are entered which are independent of one another, then interactive query performance using the DSRI can be theoretically as good as through the pre-compiler except for the parsing time of each query (generally a small percentage of the query execute time). There are two major disadvantages of pre-compilers:

- A separate pre-compiler must be made available for every supported language.
- Only pre-defined queries with parameters can be allowed (you cannot run a query with one qualifier and then decide to run it again with two qualifiers, or a different qualifier).

The solution to this problem is run-time binding access. In this case, general purpose queries can be executed at run-time. For example, the program can prompt you to enter an ad-hoc query under program control. This is not possible with compile-time access. Furthermore, all VAX host languages can be supported with run-time binding access because only standard run-time library types of calls are required. For example, with callable RDO and FORTRAN, we might use

```
CALL RDB$INTERPRET ( '
/   FOR J IN JOB
/   WITH J.JOBCLS > 5
/   PRINT J.*
/   END_FOR ')
```

In summary, the benefits of run-time binding are:

- All VAX host languages can be supported
- Ad-hoc queries can be supported under program control

Currently with Rdb/VMS, the user sees several significant disadvantages:

- The run-time Data Manipulation Language (DML) is somewhat restrictive (no transaction management is allowed, for example).
- The measured performance (as opposed to theoretically achievable performance) is rather poor relative to the pre-compilers.

To obtain a frame of reference for what is currently implemented with Rdb/VMS, the following graph in Figure 4 is instructive:

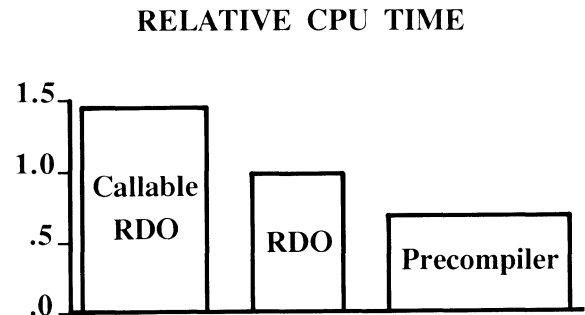


Figure 4 - Relative CPU Performance Measured From Rdb/VMS Components

With interactive RDO marked as 1.0, we have measured callable RDO to require as much as 1.4 in terms of CPU requirements for many examples, whereas the pre-compiler requires as little as 0.7. What is important to understand is that these significant differences are primarily implementation dependent and not inherent in using these three access methods with DSRI.

3 A BRIEF INTRODUCTION TO DSRI

Due to the fact that the DSRI specifications are not yet public, only basic, DSRI concepts will be presented here. The most important thing to understand about

DSRI is that it is a relational database management system standard in the same manner that RMS is the VAX/VMS Record Management Services file standard. It does not, however, define a high-level language structure. That task is left to the system implementer. The Datatrieve language implementation is the first language implemented with DSRI. The only other high-level relational language currently available which uses DSRI is QUEL as implemented in SMARTSTAR. Other languages such as SQL can also be implemented. Each language will have different complexities to deal with, as discussed in the following sections.

The DSRI consists of a low-level call interface from the programmers point of view. This call interface is fully programmable with access to control structures, recursion, aggregates, comparison operators, transaction management and data definition (create/destroy database, table, index, etc.).

The DSRI consists of two or three strictly layered protocols depending on whether or not remote access is desired (as illustrated in Figure 2). The essential protocol from the system programmers viewpoint is the BLR or Binary Language Representation. It is a low-level procedural language which includes commands for data definition and updating of these definitions and moving data between the host (system implementation) software and the database (DSRI) software. The BLR transmits instructions to the database software by way of requests. The essential items to understand about programming with the DSRI are:

- DSRI provides a complete program language as Digital's relational standard for the future. Thus, if system interfaces are developed with DSRI then the resultant information can be shared with any other program using DSRI.
- DSRI is low-level, complex programming, not for junior-level programmers or those faint-of-heart.
- DSRI is the only method for gaining access to the internal data definition language for layered product development such as we are addressing with higher level language development (high-level data manipulation language development can, however, be accomplished by "indirect" access to the DSRI via callable RDO. Unfortunately, the performance is unacceptable as will be shown).

4 THE QUEL/DSRI IMPLEMENTATION

4.1 Overview

We first started with the QUEL implementation of an Interactive Query Language (IQL), and with the Data

Manipulation Language only, since RDO can be used to perform the Data Definition Language requirements. We implemented two different systems. The first we call "Indirect DSRI Access" via the callable RDO facility, and the second "Direct DSRI Access" via the low-level DSRI calls without any reference to the Rdb/VMS components (such as the pre-compilers or the callable RDO).

4.2 The "Easy Way"

Our first attempt (the "easy way") is illustrated in Figure 5.

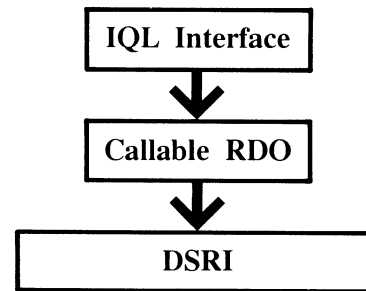


Figure 5 - First Attempt At Indirect Access To The DSRI - The "Easy Way"

This is the approach any current Rdb/VMS facility can use to indirectly access DSRI through callable RDO (the pre-compiler, as described earlier, cannot be used because the design objective is to allow totally flexible user query from the QUEL syntax). The system programmers task is limited to the box called "IQL Interface". The user's QUEL commands must be parsed or decomposed into basic elements and then recombined into the language syntax required by callable RDO (the Datatrieve syntax). The results of this approach are illustrated in Figure 6, for a data retrieval operation.

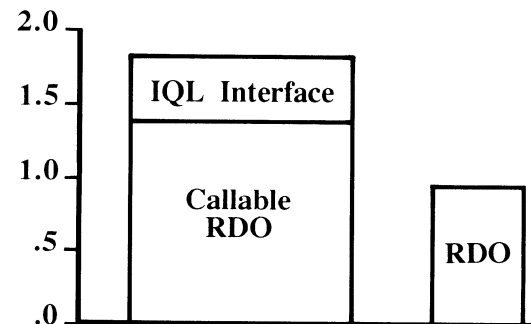


Figure 6 - First Attempt Performance Results

As shown earlier, not only is callable RDO a poor performer relative to RDO, but in addition the overhead in decomposing and recomposing the query is simply added to that of callable RDO. Even though the performance of this approach is clearly unacceptable, it is the only method available with the Rdb/VMS product, outside of direct DSRI programming, if one wishes to interface a different language syntax or system set of capabilities (such as spreadsheet access) to Rdb/VMS files.

4.3 The "Right Way"

Direct program access to the DSRI as illustrated in Figure 7 is the "right way".

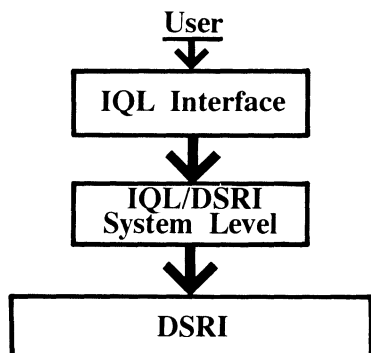


Figure 7 - Second Attempt At Direct Access To The DSRI - The "Right Way"

The first task "IQL Interface" requires development of a parser to decompose the user's arbitrary QUEL syntax into a set of keyword tokens with associated data values. The "IQL/DSRI system level" represents all of the DSRI interface coding which takes the parser output and formats it into the Binary Language Representation (BLR) for the DSRI.

4.4 Performance Results

The performance improvement is substantial. The following IQL command (using the QUEL syntax) on a demo database was executed:

```

range of e is emp
range of d is dept
range of s is sal
retrieve (d.name, e.name, s.salary)
ordered by d.name
where e.deptno = d.deptno
      and e.empno = s.empno
      and e.name = "A*"
\go
  
```

The results are shown as a function of CPU time and wall clock time for a single user stand-alone VAX-780 system in Figures 8 and 9, respectively. The performance improvement over using callable RDO is approximately a factor of three.

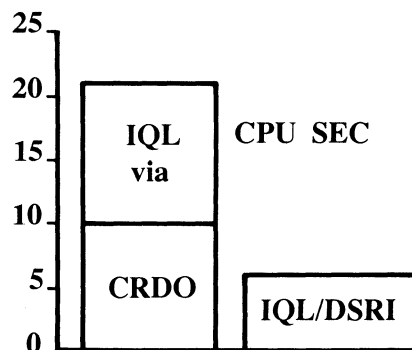


Figure 8 - QUEL/DSRI Performance Results For Complex Query (CDRO = Callable RDO, IQL = QUEL)

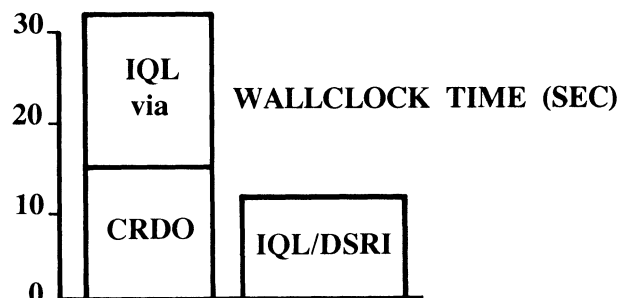


Figure 9 - QUEL/DSRI Performance Results For Complex Query (CDRO = Callable RDO, IQL = QUEL)

As one final performance measurement, the same query, only restructured as a report with formatting, headings, pages and control breaks without the "A*" qualification, was used to create a 55 page report from Datatrieve (which is the only method for obtaining reports using RDB databases) and from our IQL (even though not discussed here the QUEL/DSRI implementation has been extended to support complete report writing capabilities also). The results are shown in Figure 10 for both wall clock and CPU times.

A significant performance advantage over Datatrieve was measured. It is important to understand that in this case, we are comparing two different language implementations directly to DSRI. The performance differences relate to such things as the high-level language structure used, the parsing efficiency, and the number of messages used in communicating with the DSRI. In essence, it represents an "efficiency" rating of the system level interface coding.

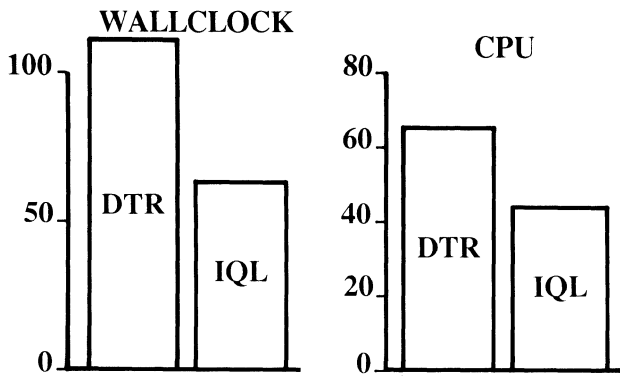


Figure 10 - A 3-Table Report
DTR/DSRI Versus IQL = QUEL/DSRI

5 THE SQL/DSRI IMPLEMENTATION

5.1 Overview

The QUEL/DSRI implementation has evolved into a production product. SQL/DSRI implementation is currently undergoing checkout. Here we will focus more on implementation concepts and observations rather than on performance details. However, a few preliminary results will be presented.

5.2 Language Differences

An illustration of the currently available relational language interfaces to DSRI (RDO/DTR and QUEL) and our proposed SQL interface is shown in Figure 11, along with who the typical interested user might be.

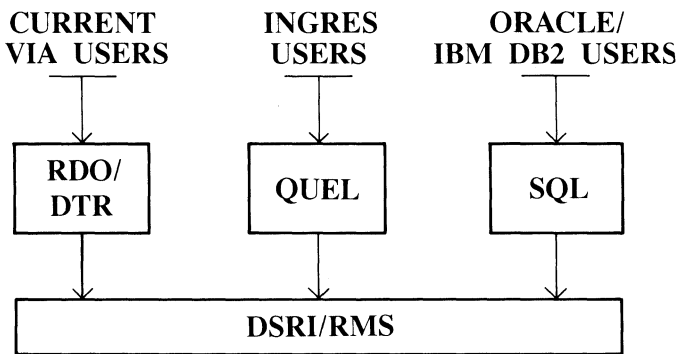


Figure 11 - Relational Languages Now Available Or Being Developed For The DSRI

At the surface level, for simple queries, the language choice would seem to be one of style over substance. For example, consider this simple one table example:

```

FOR J IN JOB           (RDO)
WITH J.JOBCLS > 5
PRINT J.*
END_FOR

RANGE OF J IS JOB     (QUEL)
RETRIEVE (J.ALL)
WHERE J.JOBCLS > 5
\GO

SELECT *              (SQL)
FROM JOB
WHERE JOBCLS > 5
;
  
```

Now, consider the following SQL example:

```

SELECT DEPT.NAME, COUNT(*)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
      AND EMP.EMPNO < 20500
GROUP BY DEPT.NAME
  
```

We are asking for a list of department names and the number of employees in each department, where the employee numbers are less than 20500. For an experienced query language user, this is a relatively simple query. In contrast, note the RDO syntax required to obtain the same result:

```

FOR E IN EMP CROSS D IN DEPT
WITH D.DEPTNO = E.DEPTNO
AND E.EMPNO < 20500
REDUCED TO D.NAME
PRINT D.NAME, COUNT OF -
E1 IN EMP CROSS -
D1 IN DEPT WITH -
D.NAME = D1.NAME AND -
D1.DEPTNO = E1.DEPTNO -
AND E1.EMPNO < 20500
END_FOR
  
```

The substantial complexity of this query is due to the fact that DSRI does not currently support the "GROUP BY" statement of SQL. Even though RDO supports aggregate functions, each aggregate function has its own tables and WHERE clause. In SQL, the aggregate functions are applied automatically to the same set of records as results from the FROM and WHERE clause. Thus, to support the GROUP BY statement, the system programmer has to implement the DSRI FOR loop and then for each unique department name generates the required result. We have implemented the SQL GROUP BY statement in this manner with very satisfactory results relative to RDO. For the above example, our SQL/DSRI implementation required 1.63 CPU seconds whereas RDO requires 2.47 CPU seconds (both implementations were running the same Version 2.0 of the underlying DSRI code).

5.3 Additional SQL/DSRI Issues

Consider the same SQL query with one additional statement added after the GROUP BY statement:

```
... GROUP BY DEPT.NAME  
HAVING COUNT(*) > 10;
```

In this case, only the departments having more than 10 employees will be listed. This query cannot be executed from RDO because the DSRI IF clause is required and it is not supported by RDO. We have also successfully implemented this structure using DSRI.

The final SELECT clause supported in the SQL language syntax is ORDER BY. Consider the statement extension to the above syntax:

```
... GROUP BY DEPT.NAME  
HAVING COUNT(*) > 10  
ORDER BY 2 DESC, DEPT.NAME
```

In this case, the final results are list ordered by the largest number of employees alphabetized by department name.

DSRI does not currently support this SQL structure since its ORDER statement is associated with the FOR loop and the final ordering desired is outside of the FOR loop. We have implemented the final sort externally from DSRI as part of the system development.

The most complex aspect of SQL/DSRI implementation, which fortunately is not generally needed, is GROUP BY and HAVING support within "subselect" statements. Until DSRI is extended to directly support GROUP BY and HAVING, SQL/DSRI subselect statements have to be limited to SELECT, FROM and WHERE. For example, an acceptable SQL/DSRI statement with a subselect statement is:

```
SELECT * FROM JOB  
WHERE JOBCLS >  
(SELECT MAX (JOBCLS)  
WHERE TITLE LIKE "S%")
```

5.4 SQL/DSRI Extensions

There are several areas of extension to the SQL language that can be directly or indirectly supported by DSRI. Extensions that we have been able to support include, for example:

- Report Writer integrated with query
- Outer Joins
- Control Constructs
- Import/Export (to/from RDB/RMS)

In addition, to fully support SQL in both interactive and programming environments, additional major development is required to support the SQL programming language syntax as specified in the proposal SQL ANSI standard and as implemented in IBM's relational database system DB2.

6 TECHNICAL SUMMARY

It appears that the design of DSRI has been strongly influenced by the QUEL syntax. Very few difficulties occurred in the QUEL/DSRI implementation. The complete set of QUEL DML operations have been implemented through DSRI including transaction management (BEGIN, END and ABORT), scalar aggregates (MIN, MAX, AVG, SUM, COUNT) and functions such as COPYIN and COPYOUT (file import/export).

Implementing SQL with DSRI has been much more complex for several reasons. As discussed earlier, SQL provides a much more concise syntax than QUEL or RDO for certain operations such as grouping. Simplicity to the user generally translated to complexity for the designer. The complexity is compounded by the fact that DSRI currently does not have request definitions in its binary language representation for GROUP BY and HAVING.

We have chosen to implement the outer join function as an SQL extension even though it is not part of the ANSI standard definition for SQL, due to its practical importance for "master detail" relationships. Since DSRI does not currently support request definitions for outer joins, it becomes a rather complex systems programming task to create the proper frontend (non-DSRI) and backend (DSRI) interaction.

The end results of our effort can be summarized as follows:

- DSRI programming is complex and time consuming.
- DSRI is at an early stage in its life and will undoubtedly have many new important features added to assist developers (such as, hopefully, GROUP BY, HAVING and outer join capabilities).
- We have been able to achieve comparable or better performance than other currently available Digital products layered onto DSRI with careful programming.
- The effort is well worth it in the final performance if you need to build a layered program on DSRI rather than being an applications user of DSRI.

"DBMS-20 Sorted Set Structures"

Jeffrey S. Finton and David W. Chilson
Bowling Green State University
Bowling Green, Ohio

ABSTRACT

The paper is a summary of a study conducted by the authors which examined the sorted set structure [owner record, index block(s), buoy record(s), and member record(s)] utilized by versions 5 and 6 of DBMS-20. Using a schema, a COBOL application program, and the DBINFO utility, the study investigated (1) the addition of a new member record to the beginning, the middle, and the end of an existing buoy chain which had already reached its maximum length, (2) the addition of a new member record between two existing buoy chains, both of which had already reached their maximum length, (3) the modification of the sort key of a member record in a buoy chain, (4) the removal of a member record from a buoy chain, and (5) the deletion of a member record from a buoy chain. A discussion of the results is provided, as are "before and after" diagrams of all sorted set structures.

INTRODUCTION

This paper is a summary of a study conducted by the authors which examined the sorted set structure [owner record, index block(s), buoy record(s), and member record(s)] utilized by versions 5 and 6 of DBMS-20. Using a schema, a COBOL application program, and the DBINFO utility, the study investigated:

(1) the addition of a new member record to the beginning, the middle, and the end of an existing buoy chain which had already reached its maximum length;

(2) the addition of a new member record between two existing buoy chains, both of which had already reached their maximum length;

(3) the modification of the sort key of a member record in a buoy chain;

(4) the removal of a member record from a buoy chain; and

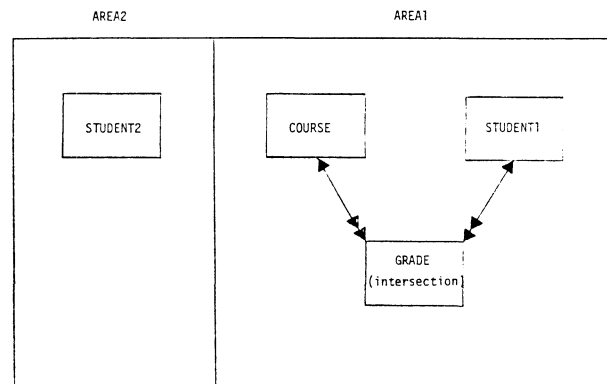
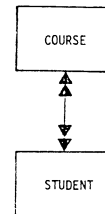
(5) the deletion of a member record from a buoy chain.

METHOD

The schema and application program used in the study were taken from a class project used in the graduate level database course at Bowling Green State University (Computer Science 562, "Techniques of Database Organization"). Basically, the project involved loading, modifying, and querying a student-course database. Because of the many-to-many relationship between students and courses, and because DBMS-20 as a CODASYL system cannot directly represent such a relationship, three record types were necessary for implementation:

Record types: COURSE
STUDENT
GRADE (an "intersection" record and a "member" of both the COURSE-GRADE and STUDENT-GRADE sets)

Set types: COURSE-GRADE
STUDENT-GRADE



(The schema and subschemas are shown in Figure 1.)

All three record types were assigned and loaded into AREA1, the first of two areas in the database. The second area, AREA2, was used in the class project for a second version of the STUDENT records, but is not relevant to this study. It should be noted that AREA1 had 9 pages (PAGE 1 to PAGE 9), a page size of 512 words, and allowed for 29 records per page. Also, since version 5 of DBMS-20 was used, the maximum length for a buoy chain was the default of 8.

The two set types, COURSE-GRADE and STUDENT-GRADE, were sorted and did not allow for duplicate member records. In the case of the COURSE-GRADE set, the ascending key was the R-SEQ data item in the GRADE record. At the time of initial loading of the database, this data item corresponded to the

alphabetical sequence of student last names. (This data item, however, was modified in parts of the study and at those points no longer yielded an alphabetical sequence.) In the case of the STUDENT-GRADE set, the ascending key was a combination of R-COURSE and R-SECTION, since there could be multiple sections of the same course.

The COURSE and STUDENT records each contained four pointers: a CALC chain pointer, an INDEX pointer, a NEXT pointer, and a PRIOR pointer. The GRADE records each contained six pointers: a NEXT pointer, a PRIOR pointer, and an OWNER pointer in each of the two sets in which they participated (i.e., in which they were members).

Primary attention was given in the study to one occurrence of the COURSE-GRADE set, specifically that for course CS 562 SECTION 0766.

Numerous program runs (application program and DBINFO utility) were made to produce the figures in this paper. Four of the figures show initial loadings of the database (i.e., AREA1). The remaining figures show the effect of various transactions against the database (i.e., additions, modifications, removals, and deletions).

ADDITION OF A MEMBER RECORD TO AN EXISTING BUOY CHAIN

In this part of the study, intersection/member records (hereafter referred to as just "member" records) were added to the beginning (head), the middle (interior), and the end (tail) of an existing buoy chain which had already reached its maximum length of eight records.

Figure 2 shows the makeup of the CS 562 COURSE-GRADE set after the initial loading of the database. There is one owner record, one buoy record, and eight member records. In all cases, the entries using parentheses [e.g., (7/??)] indicate the page number and line number (i.e., the database key) of the record occurrence. The remaining entries in the member records indicate the following for the STUDENT owner in the STUDENT-GRADE set: student last name, student ID number, and student sequence number.

The continuation of Figure 2 shows the makeup of four STUDENT-GRADE sets, where students are enrolled respectively in no courses, one course, two courses, and four courses.

Addition to the Beginning (Head) of a Complete Buoy Chain. Figure 3 shows the result of adding a member record to the beginning of a buoy chain which has reached its maximum length of eight member records. (Note the sequence number 030 coming before 040.) The original chain splits into two chains of length two and seven, and the newly added record becomes the first record in the first chain. A second buoy record is created, as is an index block (record), and both are stored on page 7, which is the page of the record immediately preceding the split.

Addition to the Interior of a Complete Buoy Chain. Figure 4 shows the result of adding a member record to the second position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 060 coming between 040 and 080.) The original chain splits into two chains of three and six, and the newly-added record becomes the second record in the first chain. The newly-created buoy and index records are stored on page 2 instead of page 1 (the page of the member record immediately preceding the split) only because page 1 has reached its maximum of 29 records.

Figure 5 shows the result of adding a member record to the third position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 085 coming between 080 and 090.) The original chain splits into two chains of four and five, and the newly-added record

becomes the third record in the first chain. The newly-created buoy and index records are stored on page 4, which is the page of the member record immediately preceding the split.

Figure 6 shows the result of adding a member record to the fourth position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 095 coming between 090 and 100.) The original chain splits into two chains of five and four, and the newly-added record becomes the fourth record in the first chain. The newly-created buoy and index records are again on page 4, which is the page of the member record immediately preceding the split.

Figure 7 shows the result of adding a member record to the fifth position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 105 coming between 100 and 110.) The original chain splits into two chains of four and five, and the newly-added record becomes the first record in the second chain. The newly-created buoy and index records are once again stored on page 4, which is the page of the member record immediately preceding the split.

Figure 8 shows the result of adding a member record to the sixth position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 115 coming between 110 and 120.) The original chain splits into two chains of five and four, and the newly-added record again becomes the first record in the second chain. The newly-created buoy and index records are still again stored on page 4 which is the page of the record immediately preceding the split.

Figure 9 shows the result of adding a member record to the seventh position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 130 coming between 120 and 140.) The original chain splits into two chains of six and three, and the newly-added record again becomes the first record in the second chain. The newly-created buoy and index records are stored on page 2 instead of page 1 (the page of the member record immediately preceding the split) only because page 1 has reached its maximum of 29 records.

Figure 10 shows the result of adding a member record to the eighth position in a buoy chain which has reached its maximum length of eight. (Note the sequence number 145 coming between 140 and 150.) The original chain splits into two chains of seven and two, and the newly-added record again becomes the first record in the second chain. The newly-created buoy and index records are again stored on page 2 instead of page 1 (the page of the member record immediately preceding the split) only because page 1 has reached its maximum of 29 records.

Addition to the End (Tail) of a Complete Buoy Chain. Figure 11 shows the result of adding a member record to the end of a buoy chain which has reached its maximum length of eight (i.e., to the ninth position in the chain). (Note the sequence number 160 coming after 150.) The original chain does not split in this case. Instead, the newly-added member record becomes the sole record in a buoy chain by itself. The newly-created buoy and index records are placed on page 8, which is the page of the last member record in the original chain.

ADDITION OF A MEMBER RECORD BETWEEN TWO EXISTING BUOY CHAINS

In this part of the study, member records were added between two existing buoy chains, both of which had reached their maximum length of eight records.

Figure 12 shows the makeup of the CS 562 COURSE-GRADE set after the second loading of the database. There is one owner record, one index record, three buoy records, two buoy chains of length eight, and one buoy chain of length three.

Figure 13 shows the result of adding a member record between two chains, both of which have reached their maximum length of eight. (Note the sequence number 160 coming between 150 and 170.) The second chain splits into two chains of two and seven, and the newly-added record becomes the first record in the chain of two. It should be noted that this result is the same as that observed in adding a member record to the beginning of a complete buoy chain. (Refer back to Figure 3.) The newly-created buoy record is stored on page 2, which is the page of the member record immediately preceding the split.

Figure 14 shows the makeup of the CS 562 COURSE-GRADE set after the third loading of the database. There is one owner record, one index record, three buoy records, two buoy chains of length eight, and one buoy chain of length one. Figure 14 also shows the makeup of one STUDENT-GRADE set occurrence (STUDENT 'TOWNSEND'), with one buoy record and three member/intersection records.

CHANGE (MODIFICATION) OF A SORT KEY IN A MEMBER RECORD

In this part of the study, the sort keys of member records in a buoy chain were changed (modified)

Change of a Sort Key within a Buoy Chain. Figure 15 shows the result of changing (modifying) a sort key so as to change the position of a member record with an existing buoy chain. The sort key for the member record corresponding to STUDENT 'FINTON' has been changed from 090 to 115, causing this member record to be moved from the fourth position in the first buoy chain to the sixth position in the buoy chain. (Note the sequence number 115 coming between 110 and 120.) The other two buoy chains remain unchanged, and the index and buoy records are unaffected.

Change of a Sort Key so as to Divide a Buoy Chain. Figure 16 shows the result of changing (modifying) a sort key so as to divide a buoy chain which has reached its maximum length of eight. The sort key for the member record corresponding to STUDENT 'FINTON' has been changed from 090 to 215, causing this member record to be moved from the fourth position in the first buoy chain to the fifth position in the second buoy chain. This in turn causes the second chain to split into two chains of four and five, and the modified record becomes the first record in the chain of five. (Note the sequence number of 215 coming between 210 and 250.) It should be noted that this result is the same as that observed in adding a member record to the fifth position in a complete buoy chain. (Refer back to Figure 7.) The newly-created buoy record is stored on page 4, which is the page of the member record immediately preceding the split.

Figure 17 also shows the result of changing (modifying) a sort key so as to divide a buoy chain which has reached its maximum length of eight. In this case, the sort key for the member record corresponding to STUDENT 'TOWNSEND' has been changed from 320 to 045, causing this member record to be moved from the first (and only) position in the third buoy chain to the third position in the first buoy chain. This in turn causes the first chain to split into two chains of four and five, and the modified record becomes the third record in the chain of four. It should be noted that this result is the same as that observed in adding a member record to the third position in a complete buoy chain. (Refer back to Figure 5.) The newly-created buoy record is stored on page 2, which is the page after that of the member record immediately preceding the split (page 1 having already been filled to its maximum of 29 records). Also to be noted in Figure 17 is the fact that the buoy record for the fourth buoy chain (7/022) remains in the database, even though it has no member records beneath it.

And finally to be noted in Figure 17 is the fact that the STUDENT GRADE set occurrence is unaffected by the modification of the COURSE-GRADE sort key.

Change of a Sort Key so as to Split Two Buoy Chains. Figure 18 shows the result of changing (modifying) a sort key so as to split two buoy chains, both of which have reached their maximum length of eight. Essentially, the modification is one which attempts to make the member record the ninth record in the first buoy chain. The sort key for the member record corresponding to STUDENT 'TOWNSEND' has been changed from 320 to 145, causing this member record to be moved from the first (and only) position in the third buoy chain to the last position in the first buoy chain. Since a ninth member is not possible, the record becomes the first record in the second buoy chain and causes the second buoy chain to split into two chains of two and seven. This result is the same as that observed in adding a member record to the beginning of a complete buoy chain. (Refer back to Figure 3.) The newly-created buoy record is stored on page 9 instead of page 8 (the page of the member record immediately preceding the split) only because page 8 has reached its maximum of 29 records. (Note that the record with database key 8/029 does not appear in Figure 18).

REMOVAL/DELETION OF A MEMBER RECORD FROM A BUOY CHAIN

In this part of the study, member records were removed or deleted from a buoy chain. Removal was from a particular set occurrence, and deletion was from the database as a whole.

Figure 19 shows the result of removing (or deleting) all member records from a buoy chain (two in this case, from the second buoy chain). The buoy record for this chain remains, with no member records beneath it, and points directly to the buoy record for the third buoy chain.

Figure 20 shows the result of removing (or deleting) a member record from a buoy chain. The effect of removing or deleting the member record corresponding to STUDENT 'MEARA' and COURSE 'CS 562' is the same in terms of the diagrams in Figure 20. That is, the removal or deletion of this record causes the logical deletion of the member record from both the COURSE-GRADE and STUDENT-GRADE sets. In the case of removal, deletion is only logical. The member record still exists physically in the database, but its NEXT pointers have been set to 0/000 (null). The NEXT pointers of the records logically preceding the removed record (one in each set) are modified so as to point around the removed record, and the PRIOR pointers of the records logically succeeding the removed record (again one in each set) are modified so as to likewise point around the record. In the case of deletion, the member record is truly (physically) deleted from the database and the space formerly occupied by the record is reclaimed: the member record with the database key 2/021 "moves up" on the page from an offset of +259 words to +249 words and is now adjacent to the member record with the database key 2/019.

Figure 20 also shows the result of removing (or deleting) a member record from a buoy chain. In this case, the purpose was to show that for a student having only a single member record, removal or deletion causes the logical (and perhaps physical) deletion of the member record, but leaves the buoy record present (for the STUDENT-GRADE set), even though there is no longer a member record beneath it.

DISCUSSION

The most important summary to be made in this paper concerns the location of newly-created buoy records and the type of buoy chain split (if any) that occurs.

Figure 11 showed the simplest case, namely the addition of a member record to the end of a complete buoy chain. The existing buoy chain remained intact, the new buoy record was

created and stored on the same page as the last member record in the existing buoy chain, and the new member record was the sole member in the newly-created chain.

Figures 3 through 10, on the other hand, demonstrated two methods for accommodating a new buoy record and a corresponding buoy chain split. In Figures 3 through 6 where the new member record was added just before the first, second, third, and fourth member records in the existing chain (respectively), the new member record was placed in the existing buoy chain and the new buoy record was in all cases placed two records after the new member. In Figures 7 through 10 where the new member was added just before the fifth, sixth, seventh, and eighth member records in the existing chain (respectively), the new member record was placed in the newly-created buoy chain and the new buoy record was in all cases placed on record before the new member.

```
*****
* DEVICE MDTAI CONTROL LANGUAGE ENTRIES
*****
```

```
IMAGES IN ORDER BY COMMAND.
INTERCEPT UPDATE EXCEPTIONS.
NOTE UNANTICIPATED EXCEPTIONS.
JOURNAL IS SCDB.
```

```
ASSIGN AREA1 TO SCDBA1
RPP IS 20
BUFFER COUNT IS 3
CALC AT MOST 3 RPP
FIRST PAGE IS 1
LAST PAGE IS 9
PAGE SIZE IS 512 WORDS.
```

```
ASSIGN AREA2 TO SCDBA2
RPP IS 18
BUFFER COUNT IS 3
CALC AT MOST 5 RPP
FIRST PAGE IS 10
LAST PAGE IS 14
PAGE SIZE IS 512 WORDS.
```

```
*****
* SCHEMA
*****
```

```
*****
* IDENTIFICATION DIVISION *
*****
```

```
SCHEMA NAME IS SCSCHEM.
```

```
AREA NAME IS AREA1
  PRIVACY LOCK FOR UPDATE IS PLU1
  PRIVACY LOCK FOR PROTECTED UPDATE IS PLPU1
  PRIVACY LOCK FOR EXCLUSIVE UPDATE IS PLEU1
  PRIVACY LOCK FOR RETRIEVAL IS PLR1
  PRIVACY LOCK FOR PROTECTED RETRIEVAL IS PLPR1
  PRIVACY LOCK FOR EXCLUSIVE RETRIEVAL IS PLER1.
```

```
AREA NAME IS AREA2
  PRIVACY LOCK FOR UPDATE IS PLU2
  PRIVACY LOCK FOR PROTECTED UPDATE IS PLPU2
  PRIVACY LOCK FOR EXCLUSIVE UPDATE IS PLEU2
  PRIVACY LOCK FOR RETRIEVAL IS PLR2
  PRIVACY LOCK FOR PROTECTED RETRIEVAL IS PLPR2
  PRIVACY LOCK FOR EXCLUSIVE RETRIEVAL IS PLER2.
```

Figure 1. Schema and Sub-Schema.

```

*****
* RECORD SECTION *
*****

RECORD NAME IS STUDENT
  LOCATION MODE IS CALC USING R-STUDENT-ID
  DUPLICATES ARE NOT ALLOWED
  WITHIN AREA1, AREA2 AREA-ID IS AREA-IN-USE.
02 R-STUDENT-ID      PIC X(005).
02 R-SECTION        PIC X(005).
02 R-NAME          PIC X(022).
02 R-FILLER        PIC X(004).
02 R-ADDRESS        SIZE IS 45  USAGE IS DISPLAY-B.
02 R-PHONE          PIC X(008) OCCURS 2 TIMES.
02 R-STANDING       PIC X(004).
02 R-EXP-GRAD-DATE  PIC X(024).
02 R-MAJORS         PIC X(035).
02 R-HOMETOWN       PIC X(020).
02 R-SEM-GPA        PIC 9V9999.
02 R-SEM-EARN-HRS   PIC 9(002).
02 R-SEM-POINT-HRS  PIC 9(002).
02 R-SEM-DUAL-PTS   PIC 9(003).
02 R-CUM-GPA        PIC 9V9999.
02 R-CUM-EARN-HRS   PIC 9(003).
02 R-CUM-POINT-HRS  PIC 9(003).
02 R-CUM-DUAL-PTS   PIC 9(003).
02 R-CUM-SEMESTER   PIC X(009).

RECORD NAME IS COURSE
  LOCATION MODE IS DIRECT C-DB-KEY
  WITHIN AREA1.
02 R-SECTION-NUMB   PIC X(004).
02 R-COURSE-NUMB    PIC X(007).
02 R-COURSE-TITLE   PIC X(020).
02 R-COURSE-HOURS   PIC 9(001).
02 R-MEETING-TIME  PIC X(011).
02 R-INSTRUCTOR     PIC X(013).
02 R-MEETING-PLACE  PIC X(008).

RECORD NAME IS GRADE
  LOCATION MODE IS VIA STUDENT-GRADE
  WITHIN AREA1.
02 R-COURSE         PIC X(007).
02 R-SECTION        PIC X(004).
02 R-STUDENT        PIC X(005).
02 R-SECTION        PIC X(005).
02 R-GRADE          PIC X(001).

```

Figure 1. Schema and Sub-Schema (continued).

```
*****  
* SET SECTION *  
*****
```

```
SET NAME IS STUDENT-GRADE  
MODE IS CHAIN LINKED TO PRIOR  
ORDER IS SORTED  
OWNER IS STUDENT  
MEMBER IS GRADE OPTIONAL AUTOMATIC  
LINKED TO OWNER  
ASCENDING KEY IS R-COURSE R-SECTION  
DUPLICATES ARE NOT ALLOWED  
SET OCCURRENCE SELECTION IS THRU  
LOCATION MODE OF OWNER.
```

```
SET NAME IS COURSE-GRADE  
MODE IS CHAIN LINKED TO PRIOR  
ORDER IS SORTED  
OWNER IS COURSE  
MEMBER IS GRADE OPTIONAL AUTOMATIC  
LINKED TO OWNER  
ASCENDING KEY IS R-SEQ  
DUPLICATES ARE NOT ALLOWED  
SET OCCURRENCE SELECTION IS THRU  
LOCATION MODE OF OWNER.
```

```
*****  
* SUB-SCHEMAS *  
*****
```

```
*****  
* SUB1 SUB-SCHEMA - NORMAL FULL ACCESS *  
*****
```

```
SUB-SCHEMA NAME IS SUB1  
PRIVACY LOCK IS SS1.
```

```
AREA SECTION.  
COPY ALL AREAS.
```

```
RECORD SECTION.  
01 STUDENT.  
02 R-ADDRESS.  
03 R-STREET PIC X(023).  
03 R-CITY PIC X(015).  
03 R-STATE PIC X(002).  
03 R-ZIP PIC X(005).  
COPY OTHERS.  
01 COURSEL.  
01 GRADE.
```

```
SET SECTION.  
COPY ALL SETS.
```

Figure 1. Schema and Sub-Schema (continued).


```

*****
* SUBIT SUB-SCHEMA - TESTING FULL ACCESS (TEMPORARY AREAS) *
*****

SUB-SCHEMA NAME IS SUBIT
  PRIVACY LOCK IS SSIT.

AREA SECTION.
  COPY TEMPORARY AREA1 AREA2.

RECORD SECTION.
  01 STUDENT.
    02 R-ADDRESS.
      03 R-STREET PIC X(023).
      03 R-CITY PIC X(015).
      03 R-STATE PIC X(002).
      03 R-ZIP PIC X(005).
    COPY OTHERS.
  01 COURSE.
  01 GRADE.

SET SECTION.
  COPY ALL SETS.

END-SCHEMA.

```

Figure 1. Schema and Sub-Schema (continued).

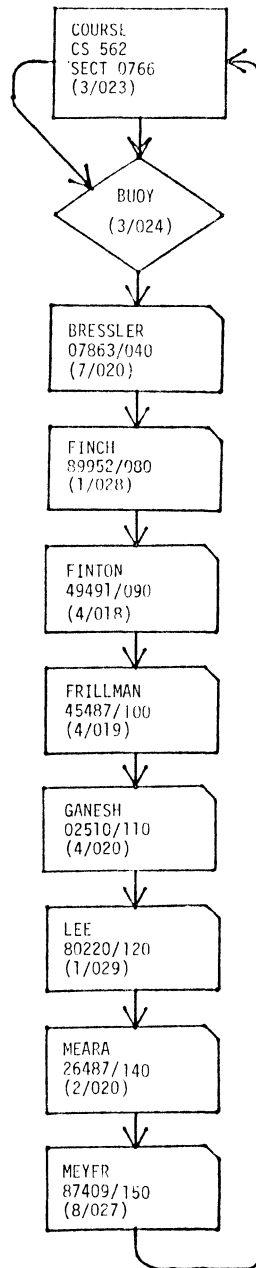


Figure 2. First Loading of the Database.

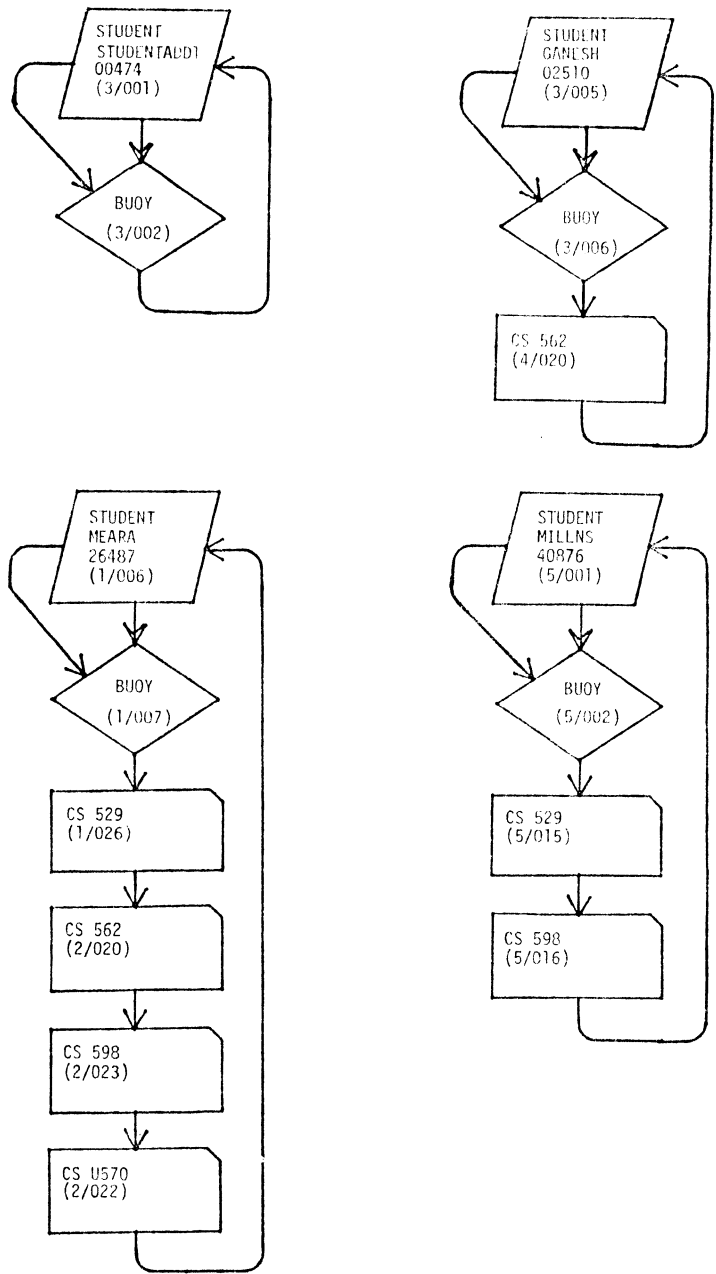


Figure 2. First Loading of the Database (continued).

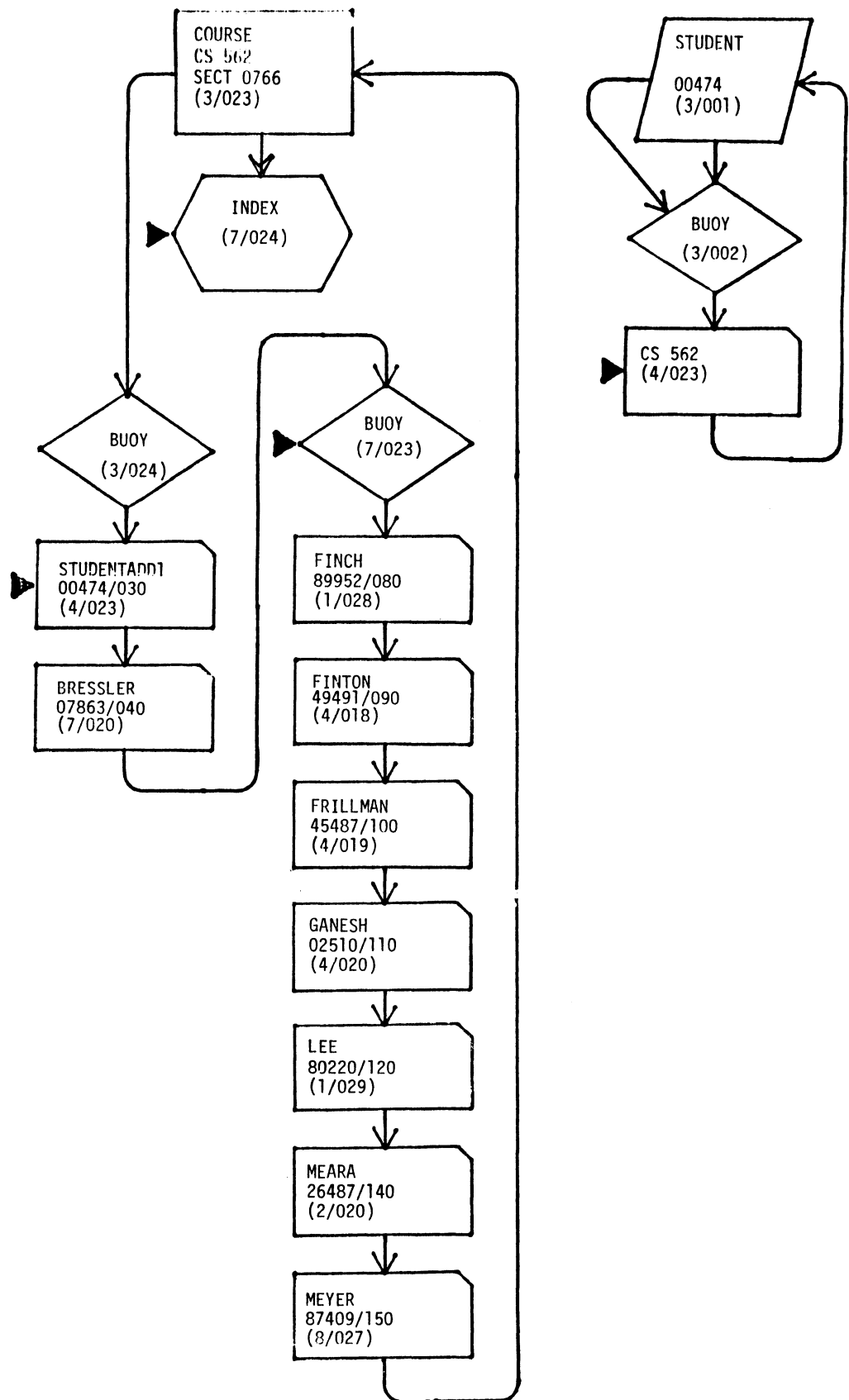


Figure 3. Addition of a Member (Intersection) Record to the Beginning (Head) of a Complete Buoy Chain.

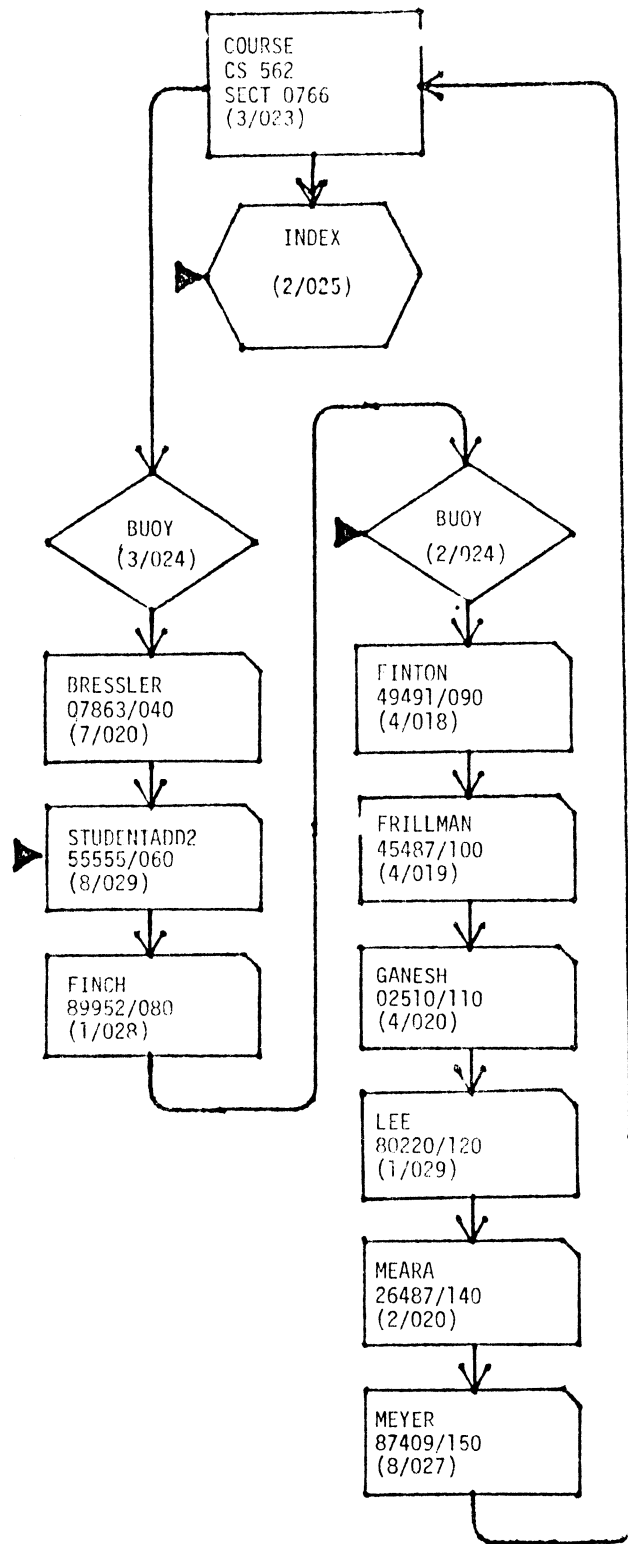


Figure 4. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Second Position in the Chain).

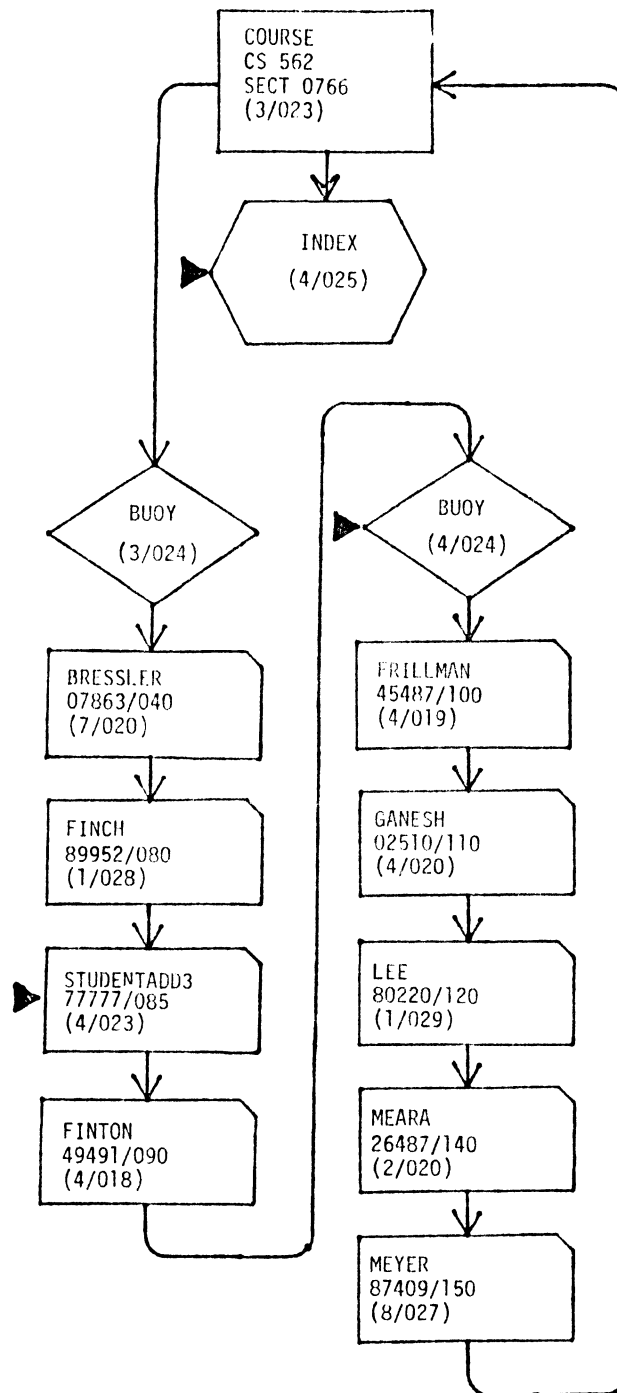


Figure 5. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Third Position in the Chain).

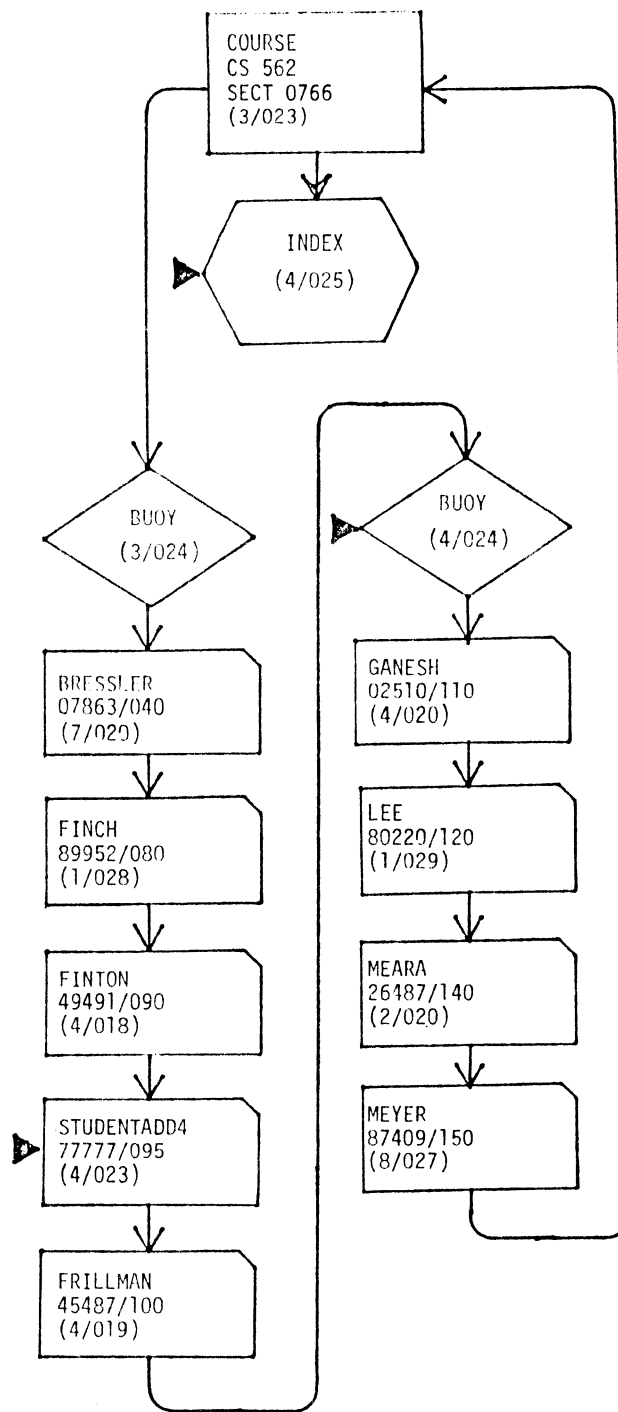


Figure 6. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Fourth Position in the Chain).

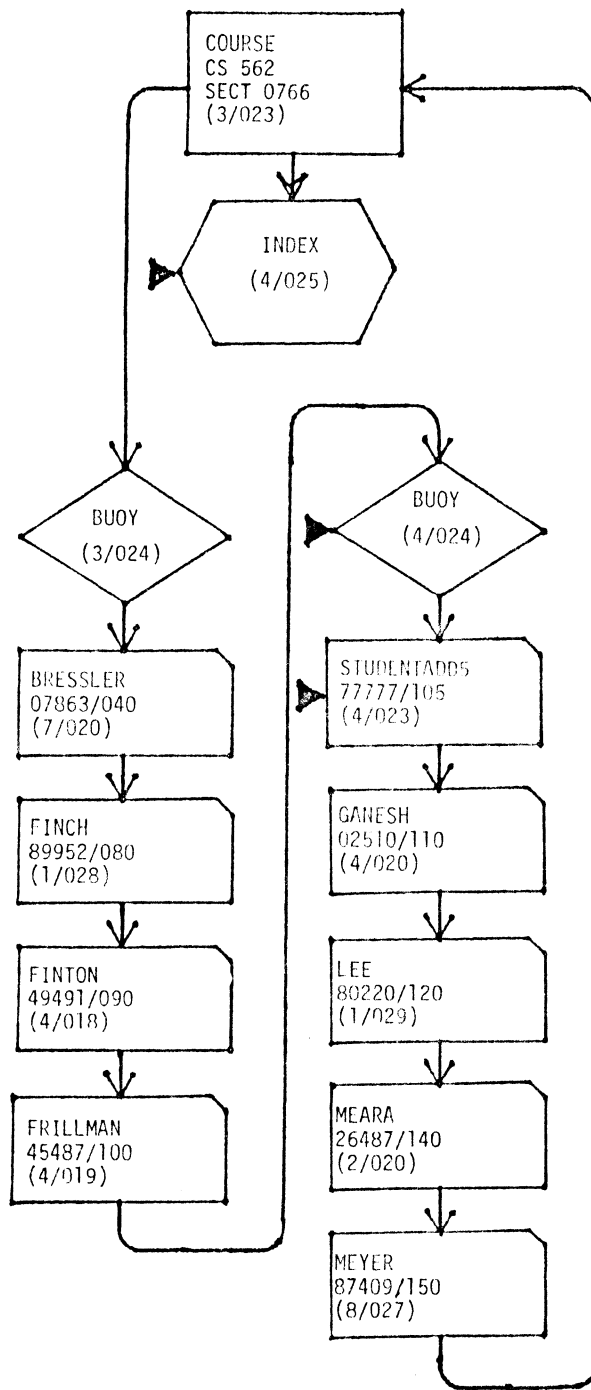


Figure 7. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Fifth Position in the Chain).

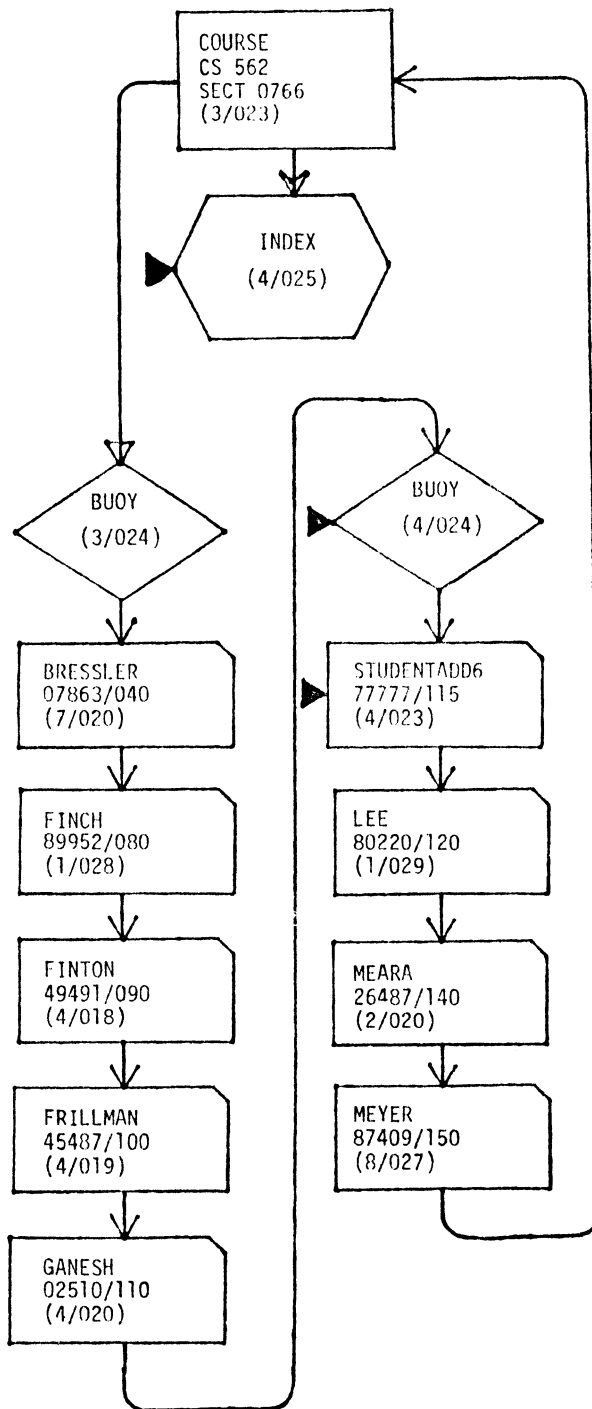


Figure 8. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Sixth Position in the Chain).

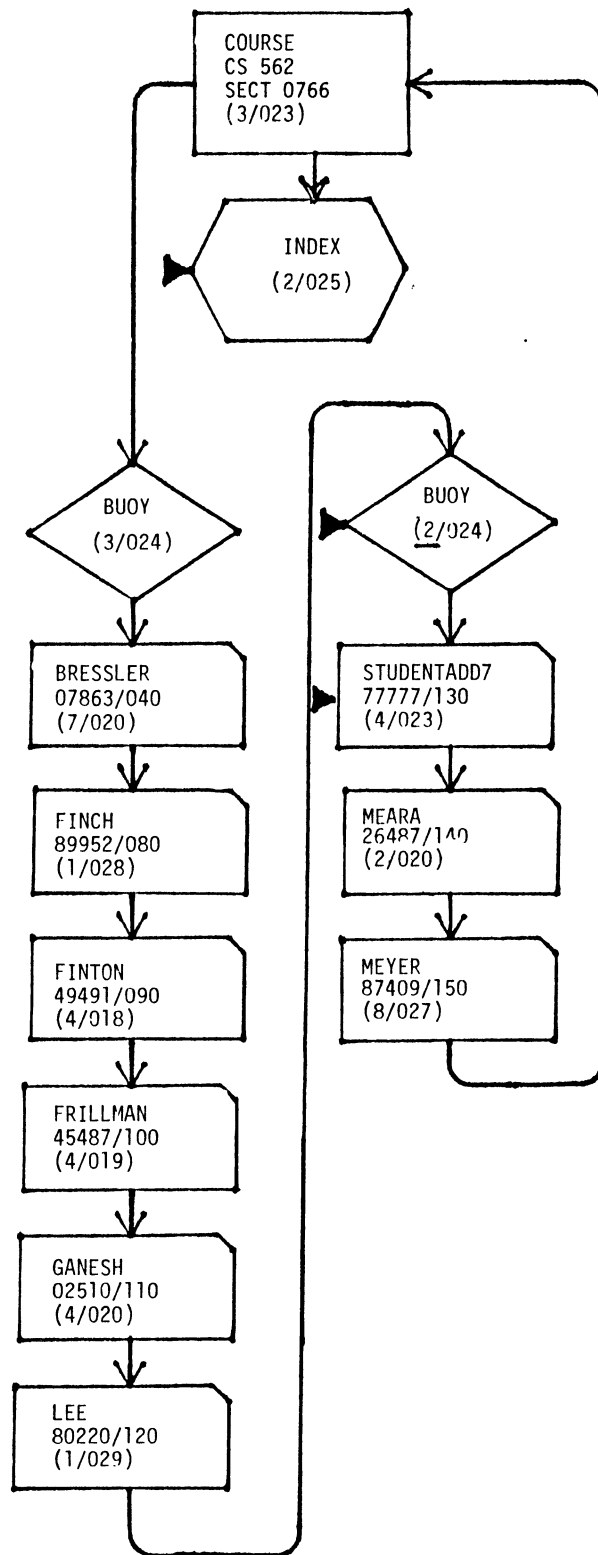


Figure 9. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Seventh Position in the Chain).

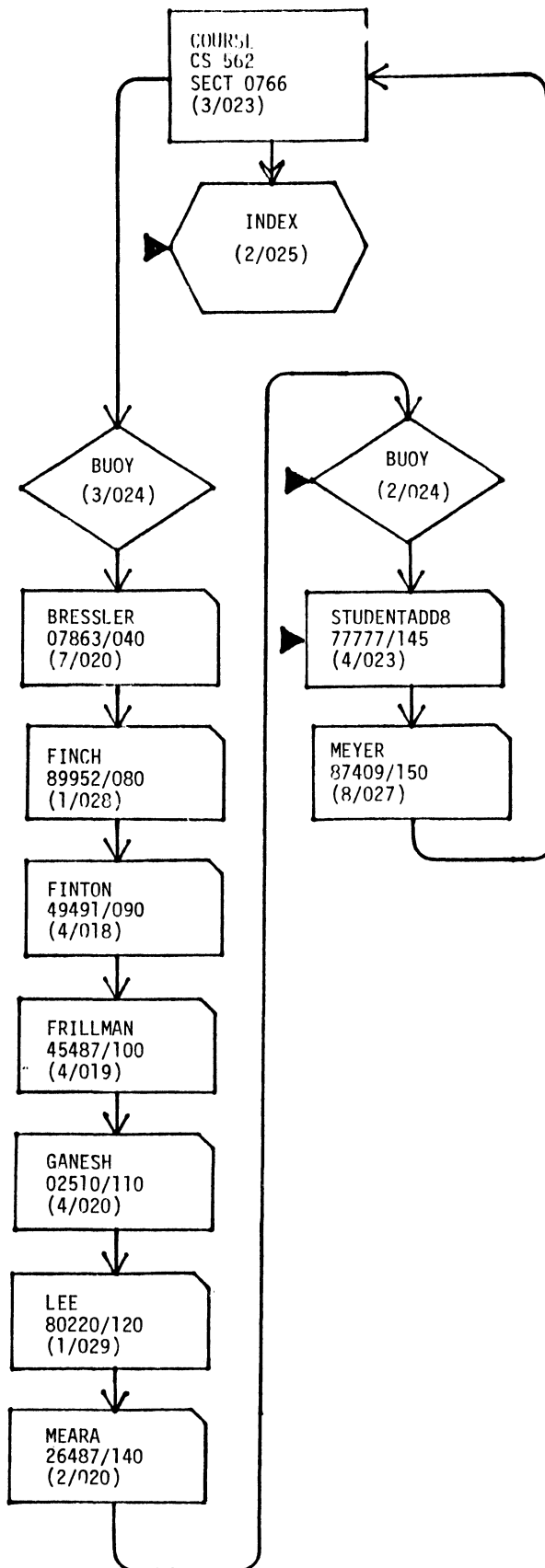


Figure 10. Addition of a Member (Intersection) Record to the Interior of a Complete Buoy Chain (Eighth Position in the Chain).

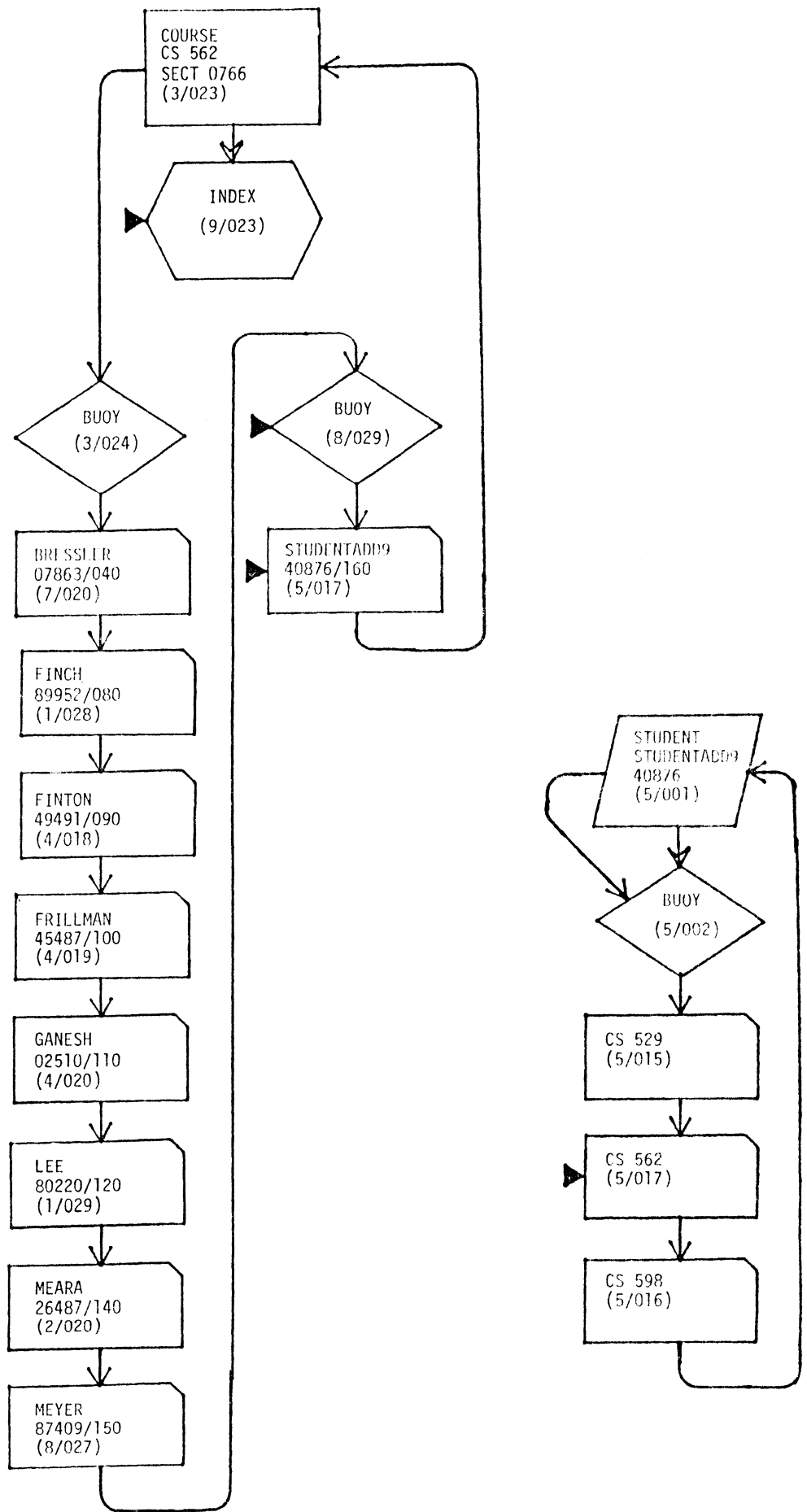


Figure 11. Addition of a Member (Intersection) Record to the End (Tail) of a Complete Buoy Chain.

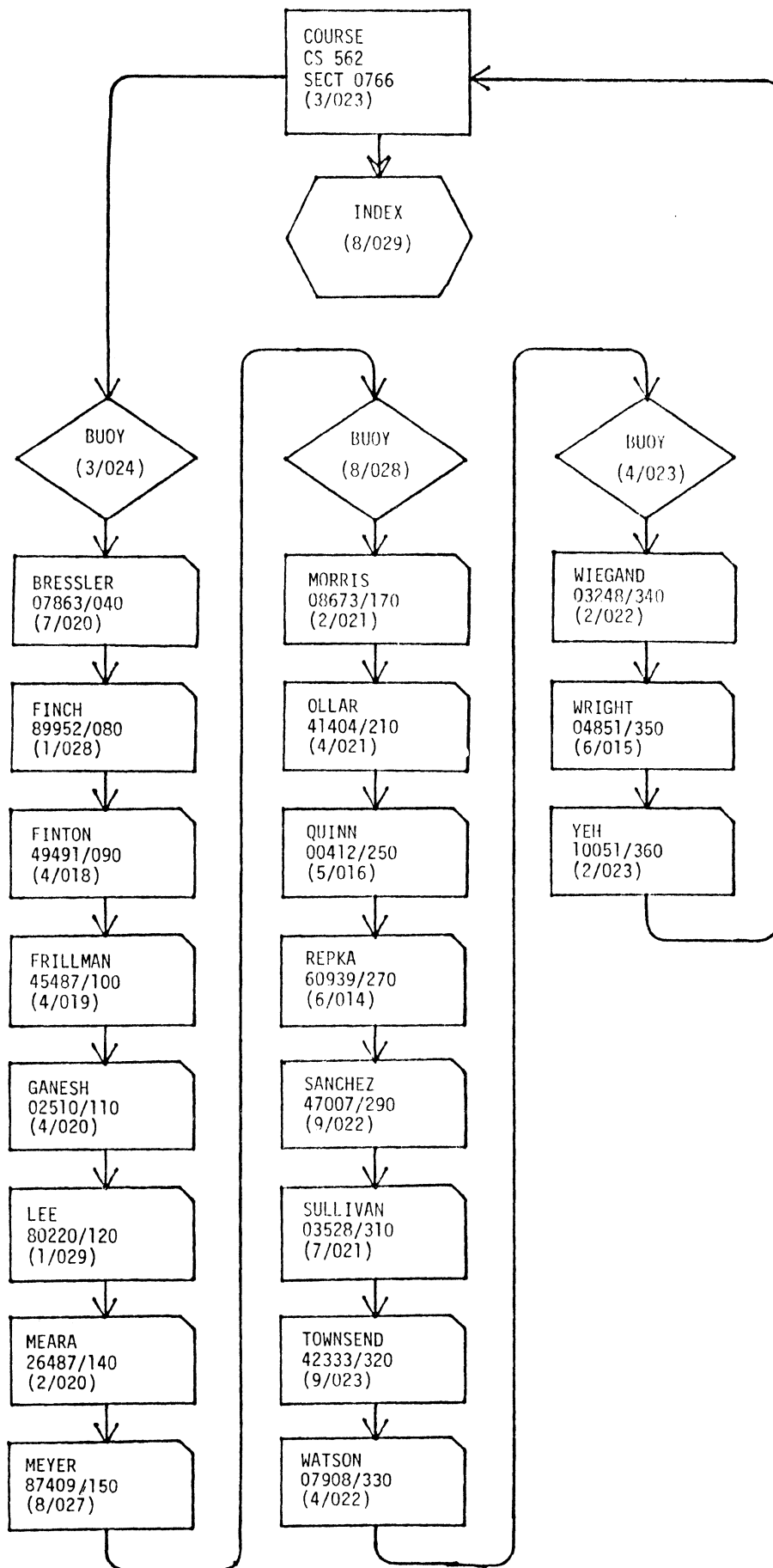


Figure 12. Second Loading of the Database.

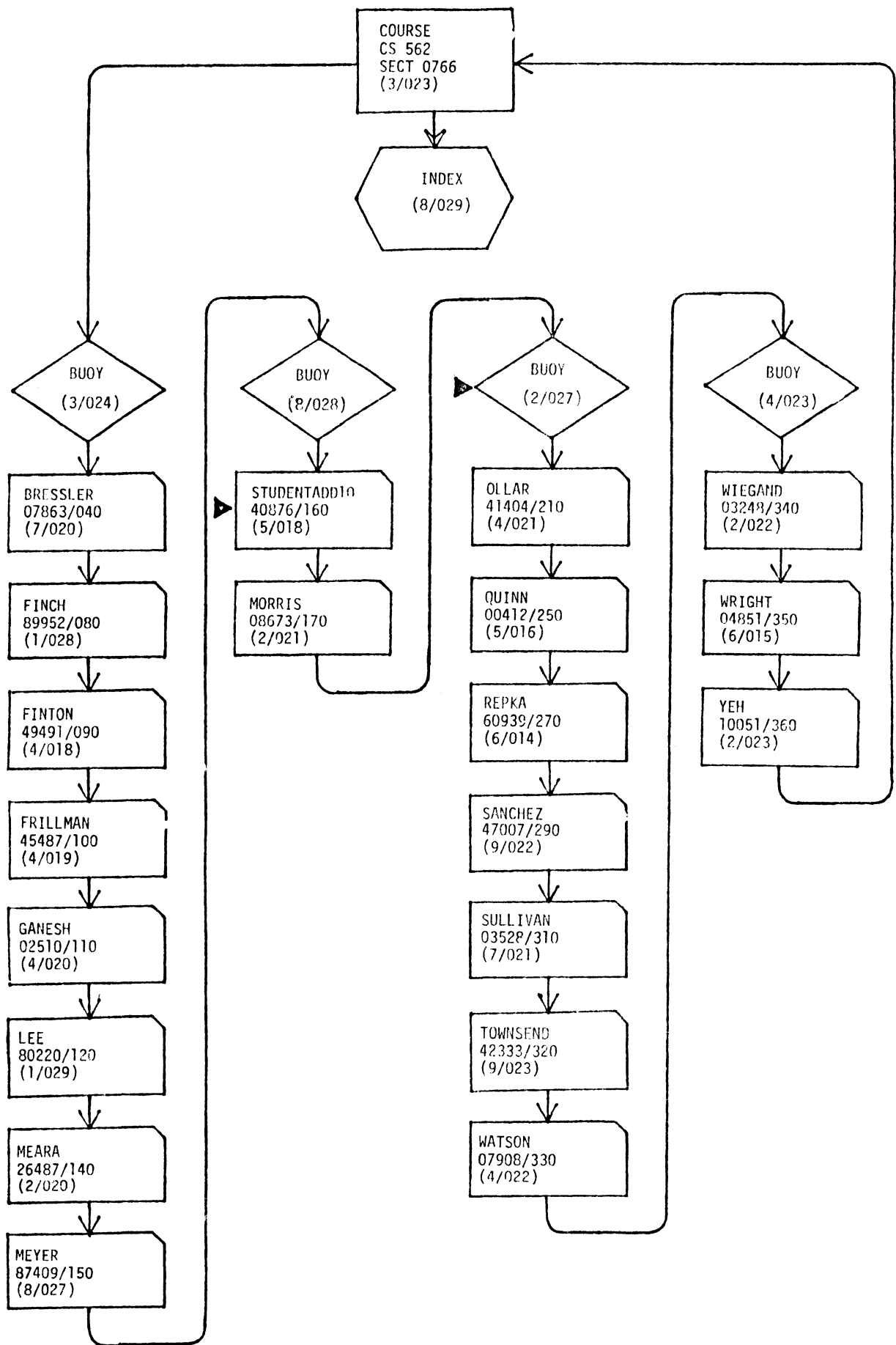


Figure 13. Addition of a Member (Intersection) Record Between Complete Buoy Chains.

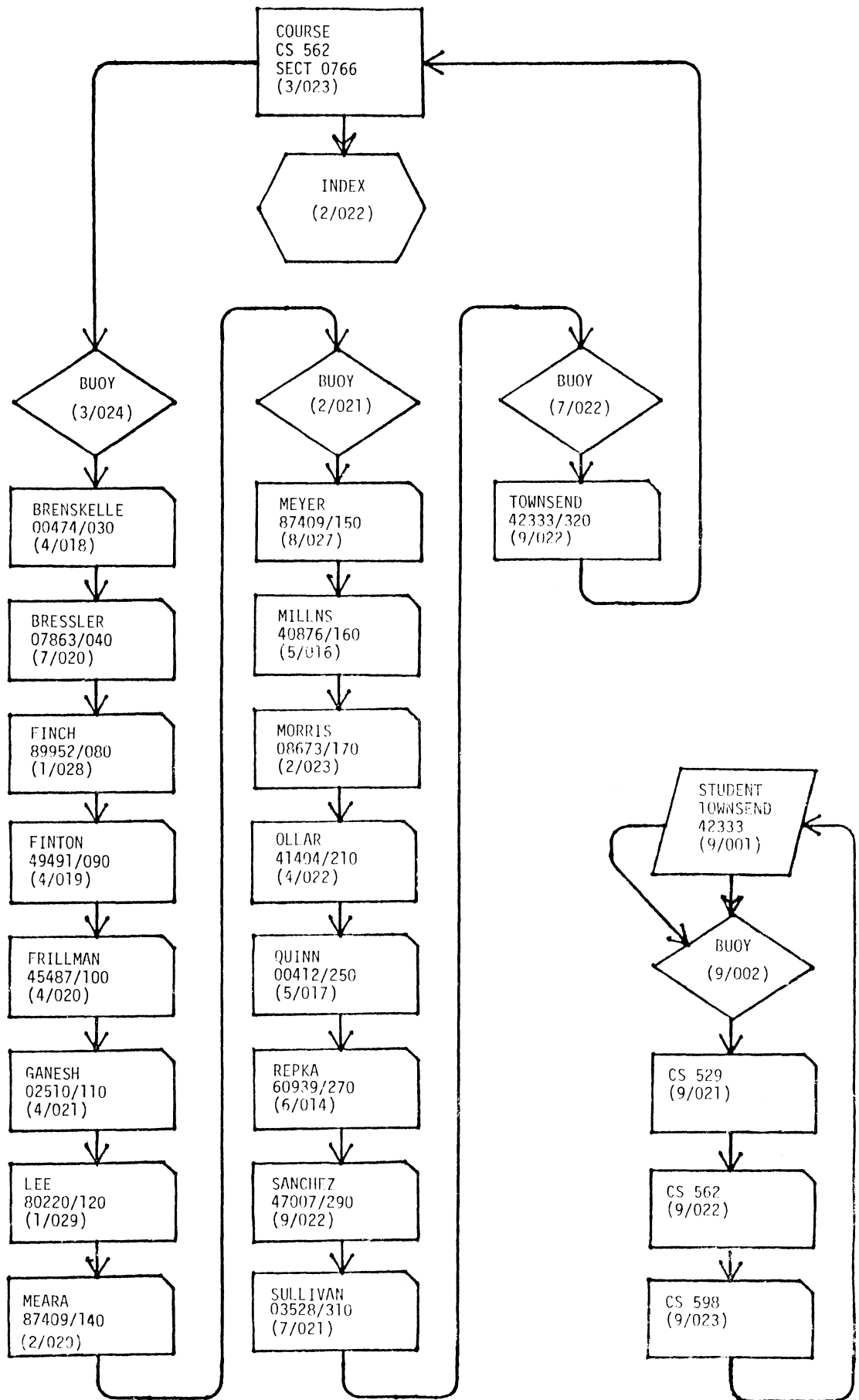


Figure 14. Third Loading of the Database.

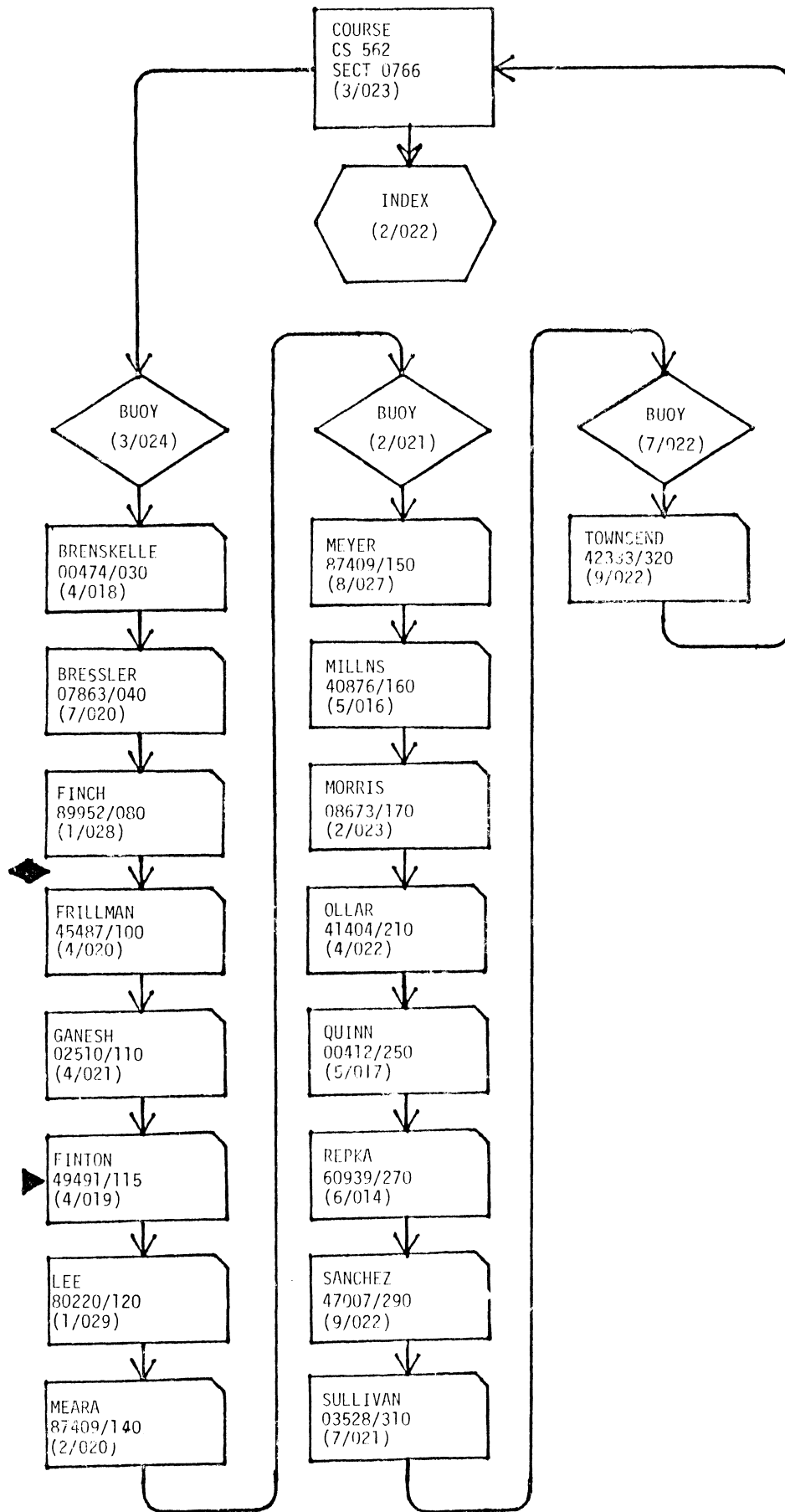


Figure 15. Change of a Sort Key within a Buoy Chain.

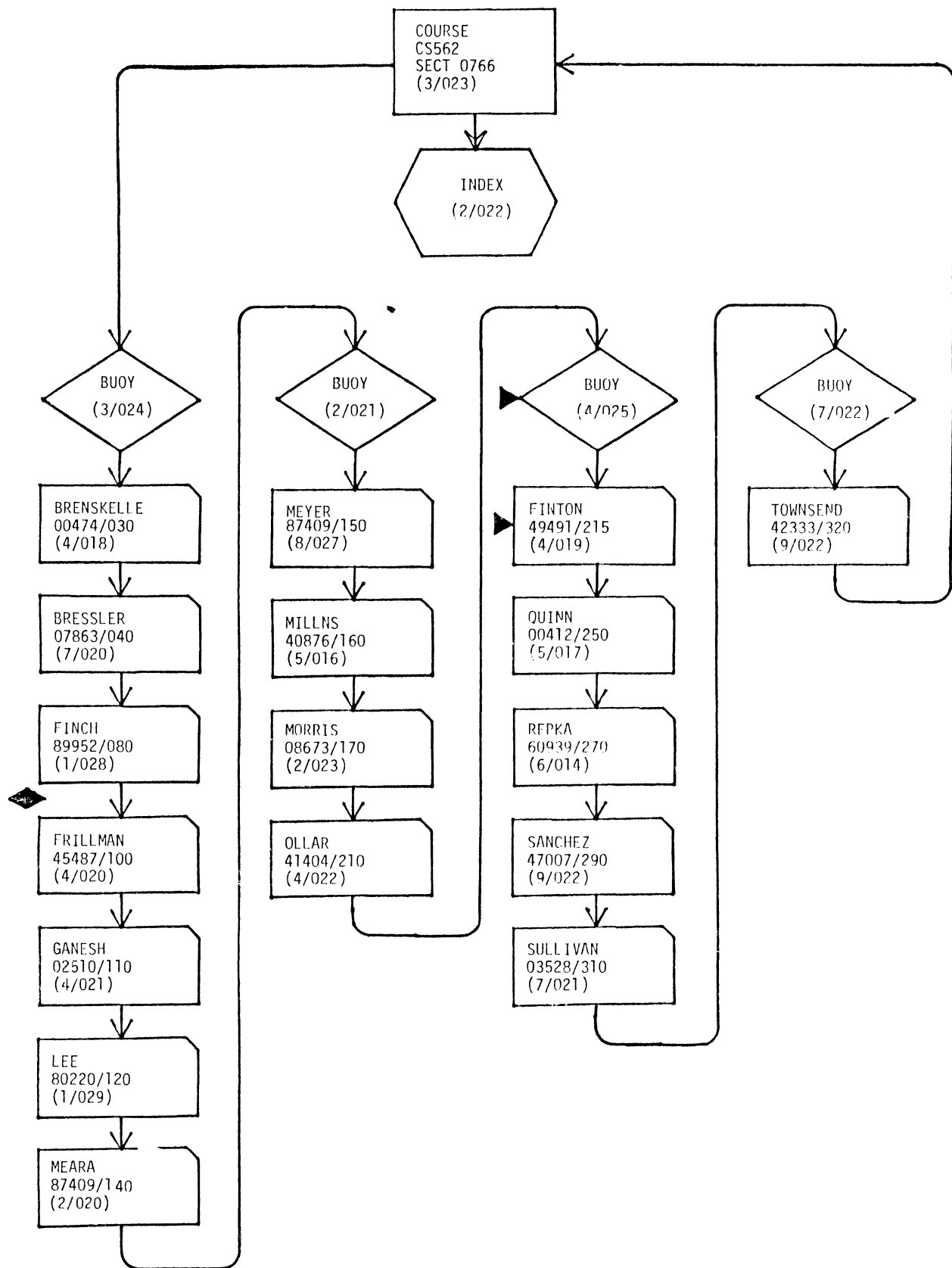


Figure 16. Change of a Sort Key so as to Divide a Buoy Chain.

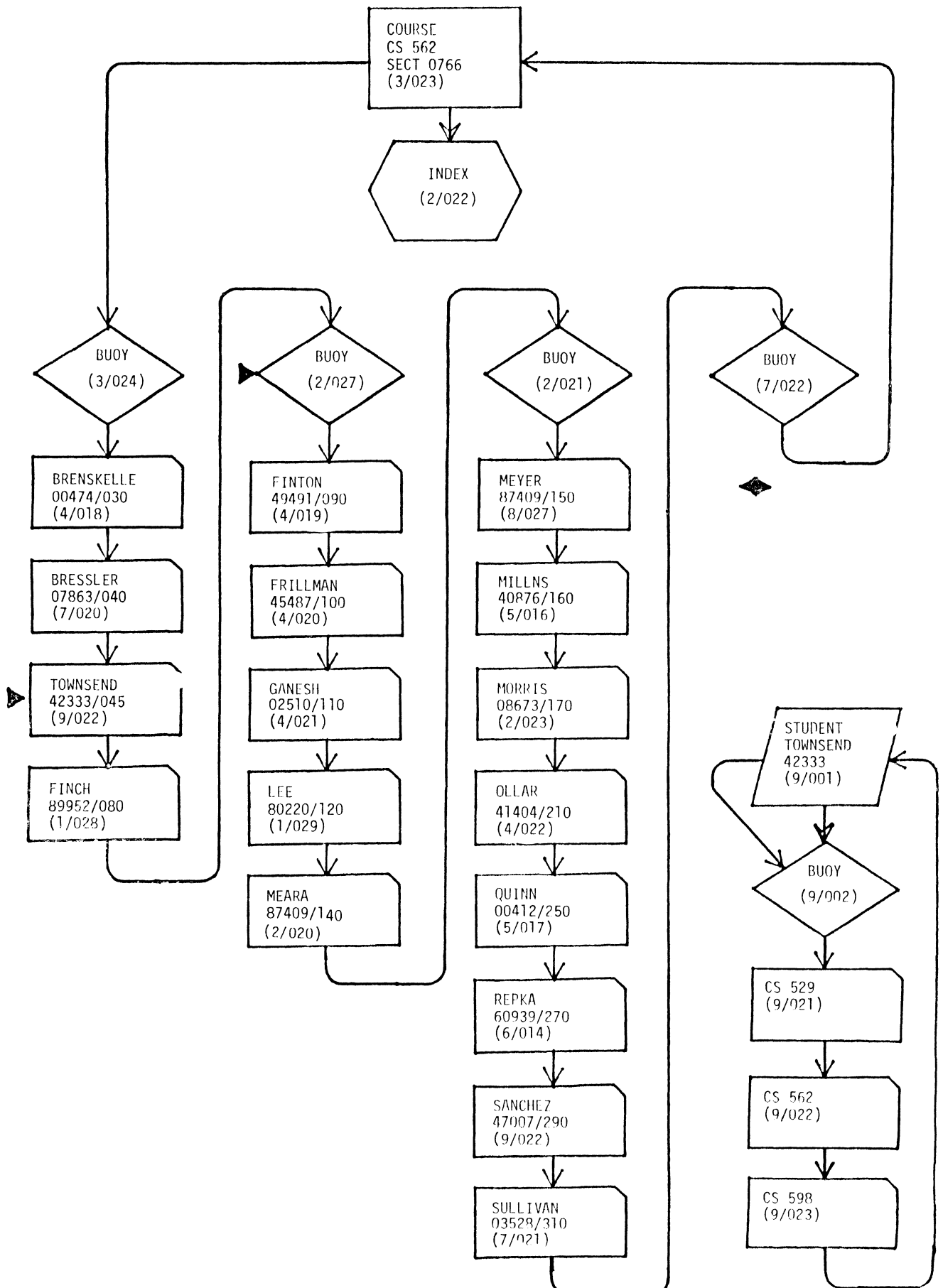


Figure 17. Change of a Sort Key so as to Divide a Buoy Chain.

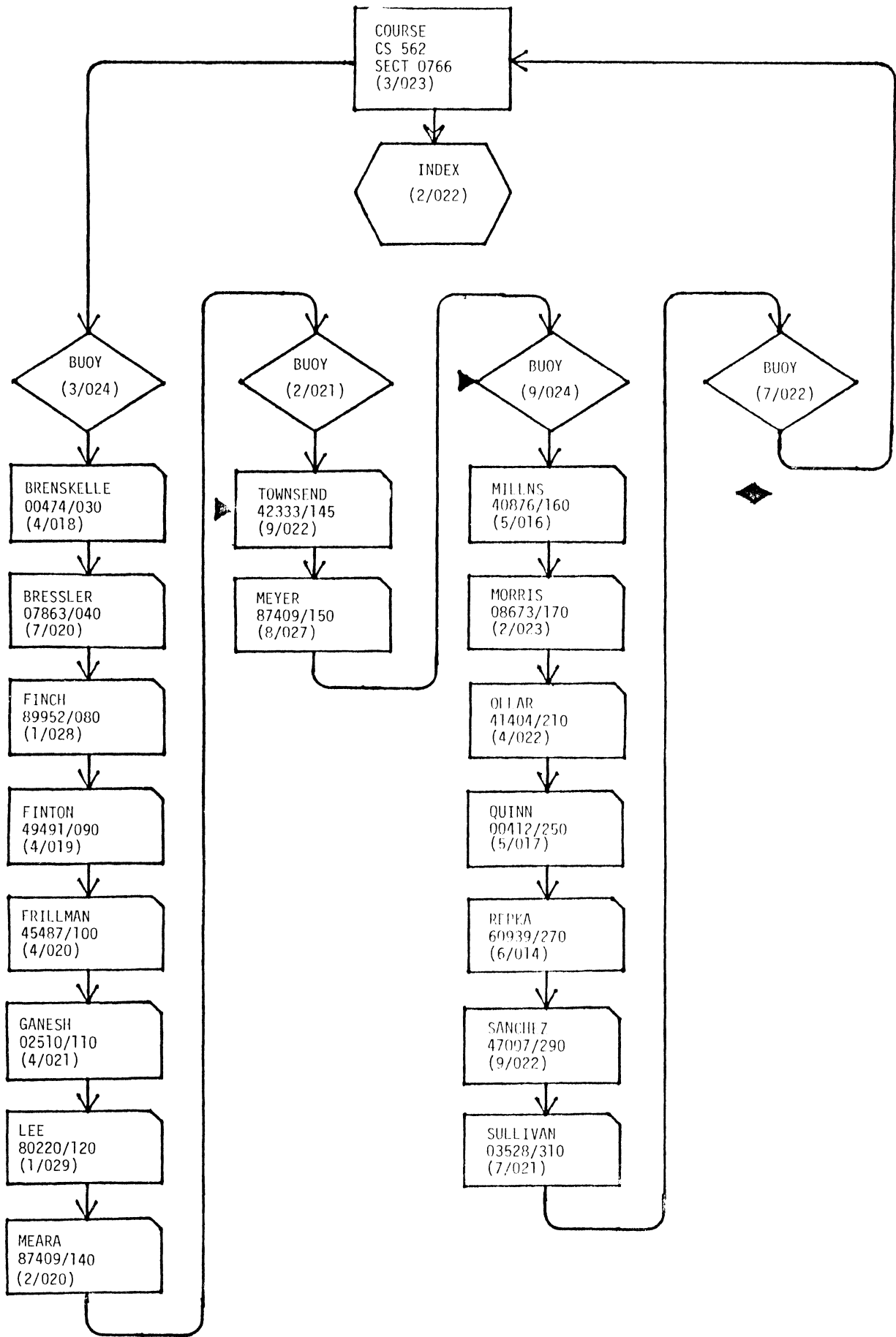


Figure 18. Change of a Sort Key so as to Split Two Buoy Chains.

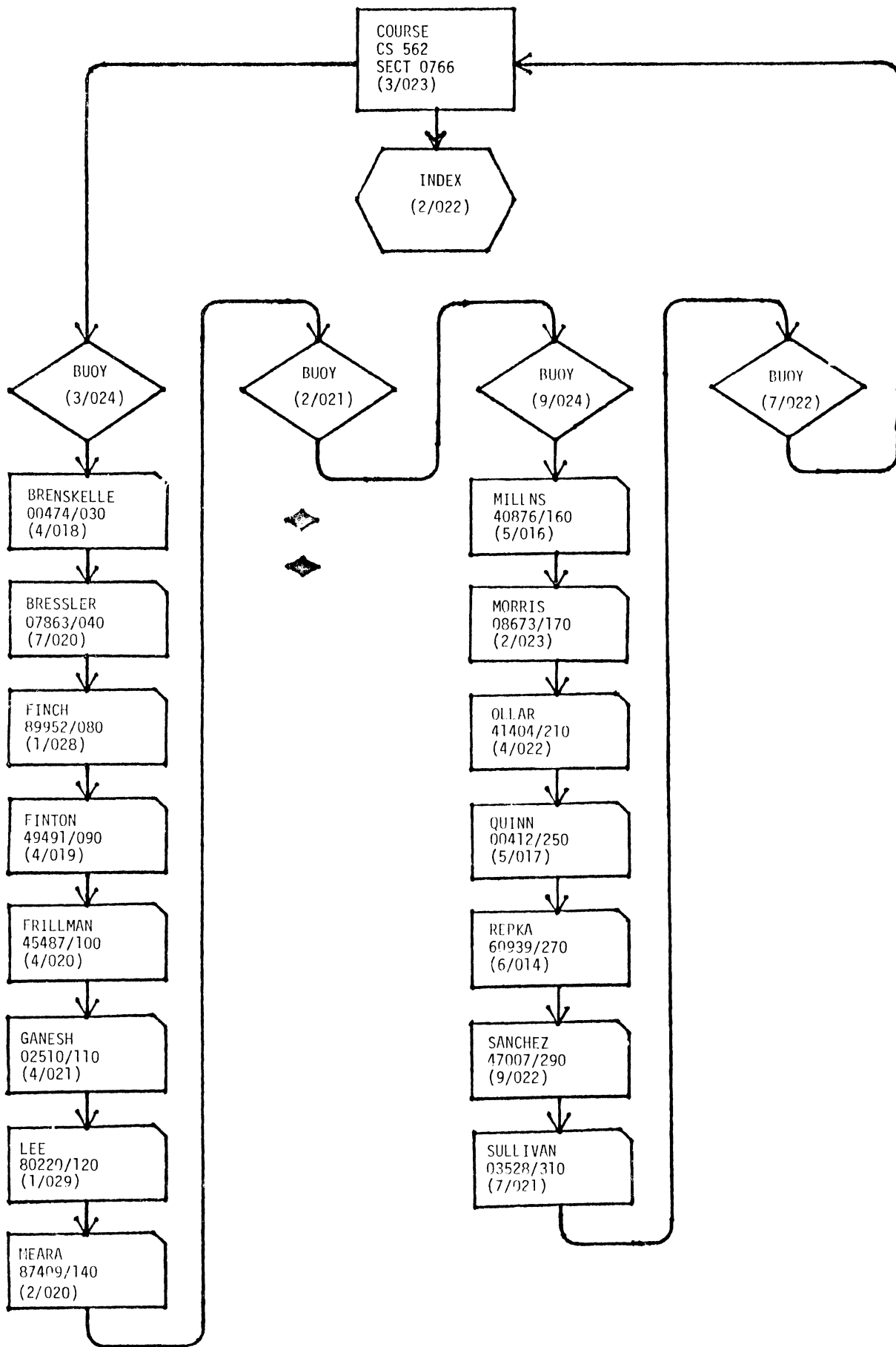
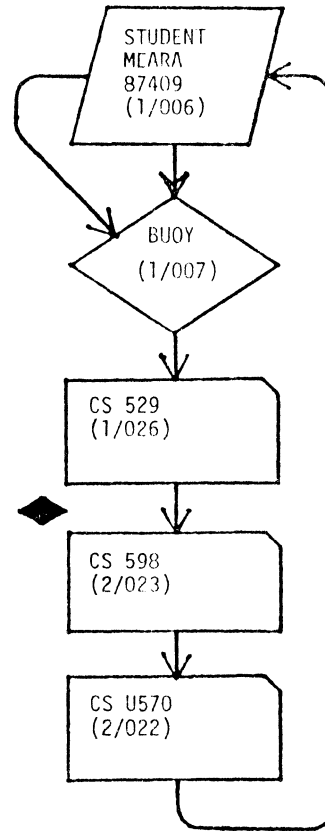
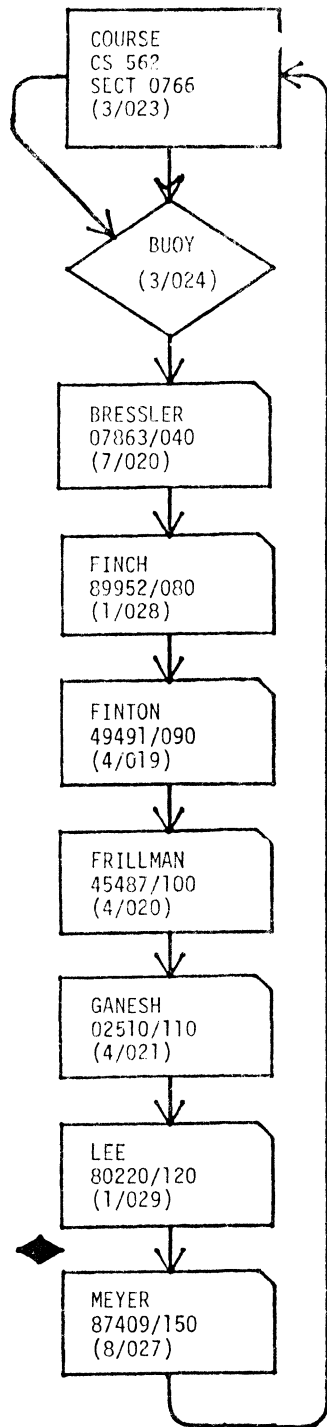


Figure 19. Removal or Deletion of All Member (Intersection) Records from an Intermediate Buoy Chain.



REMOVE GRADE FROM ALL SETS

Line 2/019: at +239 uses 10 words
 Line 2/020: at +249 uses 10 words
 Line 2/021: at +259 uses 10 words

SET NAME	NEXT	PRIOR	OWNER
STUDENT-GRADE	0/000	1/026	1/006
COURSE-GRADE	0/000	1/029	3/023
POINTER STATUS	Null	Same	Same

(Record exists with null NEXT pointers.)

DELETE GRADE

Line 2/019: at +239 uses 10 words
 Line 2/021: at +249 uses 10 words

(Deleted space has been reclaimed.)

Figure 20. Removal or Deletion of a Member (Intersection) Record from a Buoy Chain

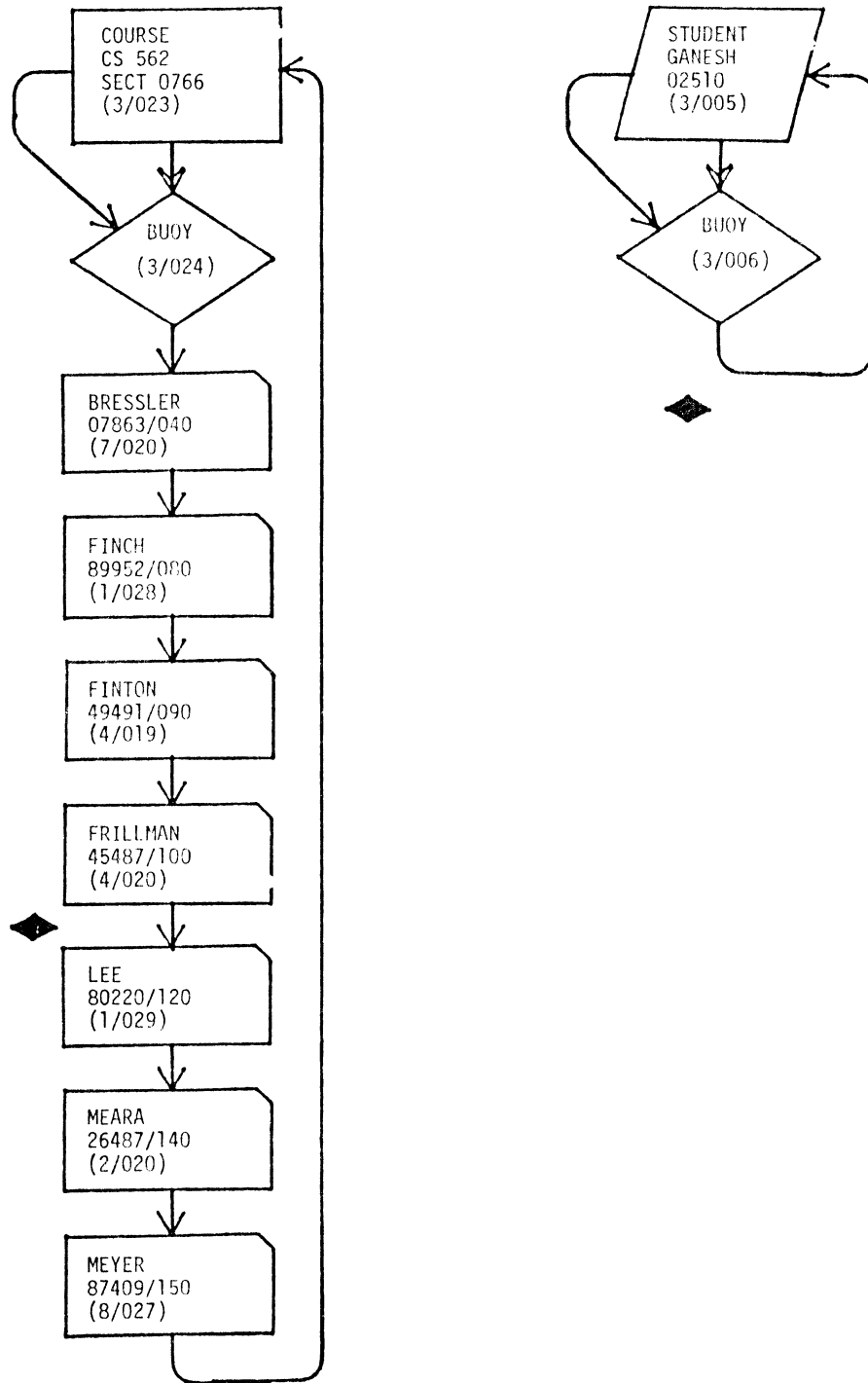


Figure 21. Removal or Deletion of a Member (Intersection) Record from a Buoy Chain -- Example 2.

A PROGRAMMER'S DATABASE SYSTEM FOR SOFTWARE
DEVELOPMENT AND MAINTENANCE

Rachel Schwab, Division of Nuclear Medicine and Biophysics,
Department of Radiological Sciences & The Laboratory
of Nuclear Medicine (DOE), UCLA School of Medicine,
Los Angeles, California 90024

ABSTRACT

Large software systems require tools for their management during both development and maintenance stages. A database is a good tool for managing the source files that compose a software system. The information in a software database falls into three areas: (1) information about each source file, (2) information about each module (i.e., program or subprogram) in each source file, and (3) information about the references each module makes to other modules.

This paper describes a database developed to monitor the source files of a large medical imaging processing system written in our division. This system runs on VAX 11/730 and VAX 11/780 computers and contains over 85,000 lines of source code.

INTRODUCTION

As a software system develops and grows, programmers must be aware of how the system is changing or else chaos will result. If programmers become confused about how the different programs and subprograms relate to each other, they are likely to make changes that cause unpredictable side effects on other system components.

In our division, four programmers worked for a period of 18 months on a large medical image processing system (UCLAPET), which is still undergoing additions and modifications. The sources for this system reside in 21 directories containing a total of 382 source files. These source files have code for 765 modules, 580 of which reside in one of the eight UCLAPET libraries.

When we began dreading even minor software changes, we realized something had to be done to keep our system organized and give us information about our sources. We developed a programmer's database system (PROGDB) using Relational Technology Inc.'s INGRES relational database management system to monitor the development and maintenance of our source files. PROGDB contains information about the contents of each source file along with cross reference data. Since we had already purchased INGRES to develop a clinical database system, we had a reliable tool to use to develop PROGDB.

This paper describes what data we put into our database, how we extract the data and organize it, and how PROGDB is used to retrieve information about our system.

DATABASE INPUT

The majority of our system source files are written in RATFIV, a FORTRAN preprocessor, with the remainder written in MACRO. Source files fall into three categories:

1. MACRO subprograms.
2. RATFIV subprograms.

3. RATFIV program with (optionally) program-specific subprograms.

All files in the first two categories exist in one of the UCLAPET libraries. These libraries are organized by functionality. For example, DISPLYLIB contains subprograms which handle image display, ROLLIB contains subprograms specific to the regions of interest program, etc.

Database data falls into two categories: data from source file headers and cross references. Whenever a programmer creates a new system source file, he or she uses the NEWPROG VAX/VMS command file to generate a source header. NEWPROG uses the appropriate comment marker (e.g., RATFIV="#", MACRO=";") for each language. Following is an example RATFIV source header:

```
# FILE.RA5 - one line file description.
#
# Last Edit           : 06-Jan-1986
# Application        : example.
# Author             : Rachel Schwab
# Modifications      :
# -----
# 06-Jan-1986 : Rachel Schwab : initial
#                                     version.
# -----
#
# ... other information
#-
```

PROGDB has routines which extract information from the source headers of MACRO and RATFIV files and put this information into the database. The information extracted directly from the source header includes: the last edit date for the file, the author, and the package name (e.g., Display, Regions of Interest, Profile, etc.).

Cross reference data is extracted using a variety of tools. For RATFIV sources, first RATFIV translates the source into FORTRAN. Then, FORTRAN produces a listing file; no object file is created. A PROGDB routine scans the FORTRAN listing and picks

off module and reference information. For MACRO sources, MACRO produces an object module. Then the LIBRARY utility creates a dummy library, inserts the object module into it, creates a listing file, and deletes the library. A PROGDB routine picks off module information from the listing file and inserts the data into the database. The LIBRARY utility is also used to determine when RATFIV and MACRO library modules were last inserted into one of the UCLAPET libraries.

IMPLEMENTATION

TABLES

The database contains four major tables: SOURCES, MODULES, REFS, and LIBMODS. Descriptions of each of these tables follows.

SOURCES. The SOURCES table holds general information about source files; there is one record per source file in the system. The following attributes are kept:

- | | |
|--------------|-------------|
| 1. file name | 5. package |
| 2. directory | 6. status |
| 3. author | 7. document |
| 4. last edit | |

The file name and the directory are determined from file lookups when the user appends, modifies, or updates source file information. The author, last edit, and package attributes are extracted directly from the file's source header. The status attribute can have a value of "okay" or "under development"; the default is "okay". The document field defaults to "none" which is hopefully not true! PROGDB's FIX option (described below) permits the user to change the values of the status and document fields.

MODULES. This table contains general information about the modules in each of the source files; there is one record per module in the system. The attributes for this table are:

- | | |
|--------------|------------------------|
| 1. file name | 4. module type |
| 2. directory | 5. library |
| 3. module | 6. library insert date |

As above, the file name and the directory are determined from file lookups when the user appends, modifies, or updates source file information. The module names are extracted using PROGDB routines. The module type may be either "PROGRAM" or "SUBPROGRAM". If a module resides in one of the eight UCLAPET libraries, the library where it resides and the date the module was last inserted into the library are also kept with the module record. This information is retrieved from the LIBMODS table (see below).

REFS. This table contains information for each reference made from a RATFIV module; MACRO references are not recorded. One record exists per reference, and presently there are 4159 records in this table. The following attributes are kept:

1. file name
2. directory
3. module
4. module reference

LIBMODS. This table is used when updating the MODULES table to determine which library a library module lives in and when it was last inserted into it. LIBMODS has the following fields:

1. module
2. library
3. last insert date into the library

The user must invoke the LIBMODS update procedure (described below) from the ADD, MODIFY, and UPDATE options to make sure the LIBMODS table is up to date. If LIBMODS is not up to date, the "library" and "last insert date into library" fields in the MODULES table will not be up to date either.

PROGDB OPTIONS

Upon initially activating PROGDB the following form is displayed to the user:

Programmer's Database System

Welcome to the programmer's database system
This database allows you to manage the source files in our system by using the following six main options:

1. ADD information to the database.
2. DELETE existing database information.
3. MODIFY existing database information.
4. Get INFOrmation on the database contents.
5. FIX status and document fields of SOURCES.
6. UPDATE database information.

```
-----
| Option (1 to 6) ? |
-----
```

The user can choose one of the six available options. Options 1, 3, and 6 (ADD, MODIFY, and UPDATE) append or update database data. Option 2 (DELETE) deletes information from the database. Option 5 (FIX) lets the user fix the status and document fields of source records. Option 4 (INFO) allows the user to query the data stored in the database. The various options are described in more detail below.

ADD and MODIFY Options:

Upon entering the ADD or MODIFY option the user is given two choices: update the LIBMODS table or proceed with the specified operation. If any of the UCLAPET libraries were changed since the last time the database was accessed for appending or updating, the user should update the LIBMODS table to insure that the library references are up to date before proceeding to add or modify source file information. The LIBMODS frame displayed below allows the user to specify the libraries that need their module information updated in the LIBMODS table.

Library Modules Update

Enter the name of the library you want to update module information for. Type "ALL" for all libraries.

Available libraries :

ATTEN, CONTOUR, DISPLYLIB, ECATIII
PROGUTIL, PROFILE, ROILIB, UTYLIB

```
+-----+
| Library ? |
+-----+
```

If the user is satisfied with the state of the LIBMODS table, the add or modify operation can proceed. A form is displayed, and the user can fill it with the device, directory, and file of the source file entries to add or modify. Wildcards may be used. For example, in the form below the user specifies to modify information about all MACRO files in the directory [UCLA.SOURCES.VAL] on the device UCLA\$SOFT. As information about each module is updated, the module name is displayed on the form.

Updating Source File Information

Operation = MOD

File : *.MAR

Directory : UCLA.SOURCES.VAL

Device : UCLASOFT:

```
+-----+
| Module = |
+-----+
```

UPDATE Option:

Similarly to the ADD and MODIFY options, upon entering the UPDATE option the user can either update the LIBMODS table or immediately perform the update operation. The user should update the LIBMODS table if any of the UCLA libraries have been changed since the last ADD, MODIFY, or UPDATE options were performed on the database. When finished updating the LIBMODS table, the update operation can begin. Any file with a directory date greater than or equal to the date the last updates were done, will have its database information added (for newly created files) or modified (for files that already have information in the database). Items such as the last edit date and library module insertion date need to be updated. Additionally, cross references may have changed.

The database maintains an internal table which keeps the last update date. For example, if the last updates were performed on December 1, 1985, all files with a directory date greater than or equal to December 1, 1985, will have their database information updated. Below is the form that is displayed to the user as the updates are performed.

Database Update Frame

Database updates will be performed on all files in the source directories which were created or modified on or after the date below.

```
+-----+
| Update Date : |
+-----+
```

File being processed :

Device =
Directory =
File =

The UPDATE option provides a way of automatically updating the contents of the database. We run it on a regular basis to make sure our database is valid.

DELETE Option:

The DELETE option allows the user to delete information about a source file from the database; this does not delete the source file itself. As with all other PROGDB options, this option simply deals with the information about a source file, not the file itself. This option is used only when a file has been removed from the source area. The programmer who deletes the source file from the source area must remember to invoke this option to delete the corresponding database information. Otherwise, the database will contain information about files that no longer exist on the source area.

The form displayed when the DELETE option is chosen is displayed below.

Deletion Frame

Enter the file name and the directory for the source file information to delete from the database.

File ?

Directory ?

FIX Option:

The FIX option allows the user to change the "status" and "document" fields in the SOURCES table. These fields default to "okay" and "none" respectively, and the only way to change these values is by invoking this option. The form displayed when the user chooses this option is shown below.

Source Record Update Frame

Fill in the form with specifications for the source record you want to fix. You can only change the status and document fields. Updates other fields must be done using the MODIFY option.

File : Directory :
 Author : Last Edit :
 Package :
 Status :
 Document :

INFO Option:

The INFO option allows the user to retrieve database information. There are five possible types of retrievals:

1. Retrieve information from the SOURCES table.
2. Retrieve information from the MODULES table.
3. Retrieve information from the REFS table.
4. Retrieve information about modules in a source file (join of the SOURCES and MODULES tables).
5. Retrieve information about references to or from a module (join of the MODULES and REFS tables).

Example queries for each of these five categories of retrievals follow:

1. Which files have a last edit date greater than date D? Which files were written by author A?
2. Which modules live in library L? What are all the program modules?
3. Which modules does program P call? Which modules call subprogram S?
4. Which modules reside in files written by author A?
5. Which modules inserted into any UCLAPET library after date D call subprogram S?

Users choose one of the five types of retrievals and a form is displayed on their terminals which can be filled with a query to process. Wildcards may be used in the query specifications. For example, the form below is displayed for the first type of retrieval (information from the SOURCES table). It is filled in with a query to retrieve all files in directory [UCLA.COMMONS] edited on or after January 1, 1986.

General Source File Information

File	Directory	Author	Last Edit
	UCLA.COMMONS		>=1-jan-1986

The data will be retrieved from the database and displayed on the terminal. Scrolling is provided if there is too much data to display on the screen at one time.

It is also possible to specify more than one query at a time. For example, if we wanted to specify the previous query (retrieve all files in directory [UCLA.COMMONS] that were edited on or after January 1, 1986) and also specify a query to retrieve source file information for all RATFIV sources with a file name starting with the letter "W", we would fill in the form as follows.

General Source File Information

File	Directory	Author	Last Edit
W*.RA5	UCLA.COMMONS		>=1-jan-1986

Below is another example of information retrieval from the database. This form corresponds to the fifth type of retrieval (information about references to or from a module). It has been filled with a query to retrieve information about all modules that call "INTVAL".

Module Reference Information

Fill in one or more attributes with specifications and type GO to execute the query.

Module :	Calls
File :	INTVAL
Directory :	
Type :	
Library :	
Date :	

CONCLUSIONS

PROGDB has proven to be a useful tool for managing the source files in our medical imaging application system. It allows easy transmission of information between programmers and facilitates the training of new programmers. Our database could be expanded to contain more information if needed. Programmers must be conscientious about updating source headers when they modify source files. We must develop methods to enforce this and ensure the validity of the information in the source header, and consequently, the database.

Philippe Collard, Division of Nuclear Medicine and Biophysics
Department of Radiological Sciences & The Laboratory
of Nuclear Medicine (DOE), UCLA School of Medicine,
Los Angeles, California 90024

ABSTRACT

The MATRIX package addresses the problem of file organization for image processing applications. The package was developed for both VAX/VMS and PDP 11/R SX systems.

From the operating system standpoint, MATRIX files are sequential access files with fixed record lengths (512 bytes). Both record and block I/Os may be performed on MATRIX files. These files are divided into blocks of data called matrices. Matrices are made up of contiguously numbered lists of virtual disk blocks and have two parts: the header (first matrix block) and the data (remaining disk blocks). The first virtual block in the matrix file is the file header. It contains information common to all matrices in the file.

A matrix directory is maintained in the file to keep track of virtual disk block allocation. This directory is a doubly-linked list of virtual blocks divided into four-word entries, one entry per matrix. Matrices can be created, read, written, deleted and "write protected" within the file.

records). The data block may be any size, even null, and matrices may have data blocks of varying sizes within the same file.

1. GENERAL DESCRIPTION

From the VMS standpoint, MATRIX files are sequential access files with fixed record lengths (512 bytes). Two primary criteria were considered during the file system design. One, since image processing was the target for this file structure, the record size had to correspond evenly to image size. Two, the file organization had to allow access optimization.

The chosen structure meets these two requirements. A record size of 512 bytes is useful since most image processing applications concern images with resolutions which are multiples of 256 (e.g., 1024, 2048, etc.). Therefore, an image row is stored in an even number of records or disk blocks. Access optimization is provided by the support of both record and block I/O. RMS is used as the primary interface. Therefore, either FORTRAN record I/O or RMS block I/O can be used, depending on how critical the access optimization is. By using FORTRAN and RMS, a high level of compatibility is kept with VMS and its layered products (e.g., DECNET).

MATRIX files have three components: the matrices, the directory list and the main header. The programmer may only access the matrices and the main header. The MATRIX package maintains the directory list. Following are detailed descriptions of these three components.

MATRICES. Matrices provide a way of orderly storing "objects" in a file. They are composed of contiguously-numbered lists of records and are divided into two parts: the matrix subheader (first record of the matrix) and the data (remaining

Several parameters are used to handle a matrix within a MATRIX file. Those parameters are saved in the "directory list", a structure described below. Each matrix is assigned a unique 32-bit word identifier when it is created. This identifier can be considered as an integer or as a four-character string. Therefore, matrices can be referenced by either a number or a name. The matrix identifier is one of the parameters saved in the directory list.

The subheader stores information specific to the object in the data block. For example, if one saves images in the data blocks, the following information could be saved in the subheader: image dimensions, image maximum, image minimum, scale factor, etc.

The MATRIX package contains routines for creating and performing I/Os on both the matrix subheader and the matrix data. The package maintains the structure's integrity and provides easy access to the data blocks and the subheaders. The user is responsible for handling the contents of the matrices.

DIRECTORY LIST. The directory list allows smooth management of the matrices in a MATRIX file. MATRIX routines operate on the directory list to create, retrieve, perform I/Os and delete matrices. A detailed description of the directory list organization is given below.

MAIN HEADER. The main header is the first record of a matrix file. One could consider it as matrix number zero with a null data block. Since the main header always is the same length for any MATRIX file, no entry exists for it in the directory

list. This structure provides storage space for information common to all matrices in the file.

2. DIRECTORY LIST STRUCTURE

The directory contains a list of "directory records" which are inserted, as needed, in a MATRIX file. Information about matrix location within the file and matrix status is kept in this list. A directory record has the same length as the other records of a matrix file (512 bytes). A matrix file can have one or more directory records and these records form a circular list. Each directory record points to its following one with the last directory record pointing to the first one.

A directory record is divided into 32 entries of four long integers. The first entry is used for managing and linking the directory records. The remaining 31 entries are used for matrix management.

When a MATRIX file is created, only the main header (record number 1) and the first directory record (record number 2) are inserted in the file. Also, the first entry of the directory list is filled and set to point to itself. As matrices are created, entries are allocated in the first directory record until there are no more entries in the directory record (i.e., creation of the 32nd matrix in the file). At this point, a second directory record is created, the entry for the new matrix is inserted into this record, and the pointers are updated to link the two directory records. New directory records are created whenever a directory record becomes full.

First Entry of a Directory Record:

-
- word 1 = number of available matrix entries in the directory record (initial value = 31; minimum value = 0)
 - word 2 = forward pointer to next directory record
OR
if record is the last in the list,
pointer to the first directory record.
 - word 3 = backward pointer to preceding record
OR
0 if record is the first one
 - word 4 = number of allocated matrix entries in the record (initial value = 0; minimum value = 31)

Notes:

-
1. word 1 + word 4 is always equal to 31
 2. When the file is created, the first entry of the initially-created directory record is:
word 1 = 31
word 2 = 2 (points to itself)
word 3 = 0
word 4 = 0
 3. The backward pointer allows another link between directory records.

Structure of a Matrix Entry in a Directory Block:

-
- word 1 = matrix identifier (integer*4 - matrix number or four-character string - matrix name)
 - word 2 = matrix subheader record number
 - word 3 = last record number of matrix data block
 - word 4 = matrix status
1=matrix exists (access = read/write)
2=matrix exists (access = read only)
-1=matrix deleted (access = none)

Since all directory record pointers and matrix pointers (words two and three of a matrix entry) are long integers, the MATRIX structure supports files of almost unlimited length. Additionally, the matrix identifier is also a long integer, and, therefore, the number of matrices one can store in a MATRIX file is very large.

The MATRIX package ensures the uniqueness of a matrix identifier within a MATRIX file, regardless of whether matrix names or matrix numbers are used. Within a file, any given identifier references one and only one matrix. With characters string identifiers, lower case and upper case characters are significant. For example, the identifier "IMA1" differs from "imal" or "Imal". When numbers are used as identifiers, matrices can be created in any order (e.g., matrix number 10 can be created before matrix number 1).

The second and third words of a matrix entry define where the matrix lives within the MATRIX file. Since matrices consist of contiguous records, if the record number of the matrix subheader (first record of a matrix) and the record number of the last record of the data block are known, the location of the matrix in the file is completely defined.

The fourth word of a matrix entry gives the matrix status. If this word has a value of 1, the matrix exists and can be read or written. If it is set to 2, the matrix exists but can only be read. If a MATRIX routine is called to write to a matrix whose status is 2, the routine returns an error since the matrix is "write protected". If the matrix status is -1, the matrix has been deleted. The space it occupies in the file can be recovered. Additional details on matrix creation and deletion may be found in the next chapter.

3. MATRIX ACCESS

A. Directory Lookup:

Almost all operations performed on MATRIX files involve directory lookups. This operation scans the directory list to check for the existence of a particular matrix. If the matrix exists, its directory entry is returned to the requesting module. Depending on the operation, the entire directory list or only a part of it will be scanned.

For example, when a new matrix is created, the entire directory list is scanned to check if the matrix already exists. When retrieving matrices, the lookup terminates when the specified entry is found; the list is completely scanned only if the matrix does not exist in the file.

- 1) OLD
- 2) NEW
- 3) UNKNOWN
- 4) SCRATCH

Scanning of the directory list is sequential, and, therefore, directory lookup operations are a performance bottleneck. Some features of the design help reduce this negative effect. The directory list has a circular structure which allows lookups to begin anywhere in the list. Also, a "cache" was implemented for directory lookup. This cache is large enough to hold one directory record. Directory records are read directly into the cache, and at any time, the cache contains a copy of the last directory record accessed by the package.

Directory lookups begin by scanning the cache. If necessary, the next directory record is read into the cache and the lookup continues. This scenario greatly improves directory lookup in two specific cases. First, when the number of matrices in the file is less than 31. In this case, the entire directory list is resident in the cache after the first lookup is performed on the file. Second, when matrices are stored in the file in the order they are most likely to be processed. Performing operations on a long series of matrices will only require a few file I/Os for the directory records, since the cache contains up to 31 matrices which can be sequentially processed.

Of course, if matrices are saved and retrieved in random order, the cache is of no great help. Another case where the cache is not useful is if several files are processed at the same time. Since the cache is labeled with the logical unit number (LUN) of the corresponding matrix file, the cache ownership changes as access switches from one file to another.

Whenever a modification is made on a directory record (e.g., matrix deletion), the operation is performed on the cache and the cache is written back to the file. This ensures that information in the cache is always up to date.

B. Opening and Closing MATRIX Files:

MATRIX files must be opened and closed by using special routines described below. The routine to open or create a MATRIX file is called by:

```
CALL MATST(FPA,ACCESS,STATUS,ERROR)
```

where:

FPA = file parameters array. A specially formatted array holding the LUN, the file name and other file attributes.

ACCESS = file access code. Can have one of the following values:

- 1) READ
- 2) WRITE
- 3) SHARE

STATUS = file status code. Can have one of the following values:

ERROR = error status:

- if positive, number of matrices in the file (0 = no matrices)
- if negative, file operation error

This routine opens a MATRIX file and allows the other MATRIX routines to operate on the file. The FPA is easily formatted by calling another MATRIX package routine not described in this article. It allows the specifications of various parameters, e.g., "multi-buffer count".

MATST is the only package routine which uses the FPA. All the other routines reference the file by its LUN (logical unit number) which is associated to it during the call to MATST.

The ACCESS and STATUS parameters have the same meaning as for an equivalent FORTRAN OPEN statement. For example, a file opened SCRATCH will be deleted when closed.

When a new file is created, the main header and the first directory record are inserted into it. MATST returns the number of matrices present in the file as its error status.

MATRIX files are closed by calling MATFIN.

```
CALL MATFIN(LUN,ERROR)
```

where:

LUN = the associated logical unit number
ERROR = the error status

After a call to MATFIN, the file is unavailable for further MATRIX operations. A file opened SCRATCH is deleted.

Additional explanations for the necessity of special open and close routines are given in the next chapter.

C. Creating and Deleting Matrices:

Creating a matrix in a MATRIX file is the most complex operation performed by the package. First the directory is scanned and during this process, several operations are simultaneously performed. One, the package checks that there is no matrix in the file with the same identifier as the one to be created. If there is a matrix with the same identifier, the operation aborts and an error code is returned. Two, the package keeps tracks of the holes in the files (i.e., deleted matrices) and determines which hole best fits the matrix that is to be created.

If a large enough hole to hold the new matrix is found, we have all the necessary information for creating the matrix. Only the space needed for the new matrix will be allocated to it if the hole is larger than needed. The directory block containing information about the hole is updated with the parameters of the new matrix and the creation completes.

When there is no hole available for the new matrix, the package checks for an available directory entry in the last directory record of the directory list. If one exists, the space needed for the new matrix is added to the file. In this situation, matrix creation occurs at the EOF (End Of File).

If there is no room for a directory entry in the last directory record, a new directory record is created at the EOF. Pointers are set up to link the new directory record to the directory list. Then the matrix creation resumes as in the previous case.

Matrices and new directory records are always added to the EOF. The package keeps track of the highest record number found in the file during directory lookup and this record points to the EOF.

No data is transferred when a matrix is created. The matrix creation only locates a place for the new matrix and updates all necessary information. In order to write data to the matrix, the appropriate routines must be used once the matrix is successfully created.

The call to the matrix creation routine is:

```
CALL MATCRE(LUN,MATNUM,LEN,ERROR)
```

where:

```
LUN      = the logical unit number
MATNUM   = matrix identifier (number or name)
LEN      = length (in number of records) of
           matrix data block (actual space
           occupied by the matrix will be
           LEN+1 records. LEN for the data
           block and one for the subheader).
ERROR    = the error status for the operation
           > 0: ok
           = 0: matrix already exists
           < 0: file operation error
```

A routine for deleting matrices from a MATRIX file also exists. This is a simple operation. The directory list is scanned for the directory entry of the matrix to be deleted. If found, it is updated to reflect its new status, i.e., removed from structure. From then on, the space formerly occupied by the matrix may be allocated to create a new matrix. The subroutine to delete a matrix is called as follows:

```
CALL MATDEL(LUN,MATNUM,ERROR)
```

where:

```
LUN      = the associated logical unit number
MATNUM   = matrix identifier (number or name)
ERROR    = the error status for the operation
           > 0: ok
           = 0: matrix does not exist
           < 0: file operation error
```

D. Reading and Writing Matrix Data:

Several routines are provided to perform I/O on matrix data blocks. They fall into three categories: record mode I/O, word mode I/O, and fast mode I/O.

When I/Os are performed on a matrix, the I/O parameters are checked against the allocation of the

matrix. For example, if the data block of a matrix is ten records long, it is impossible to read the eleventh record of it. The package does not know what the data blocks are used for, but it insures that the file structure remains valid.

RECORD MODE. In record mode the matrix data block is considered as a list of 512-byte records. The routine provided to perform I/Os in record mode can be called as follows:

```
CALL MATIOR(LUN,MATNUM,BUFFER,FIRST,NUMBER,
            INCREMENT,IOCODE,ERROR)
```

where:

```
LUN      = the associated logical unit number
MATNUM   = matrix identifier (number or name)
BUFFER   = I/O buffer (read or write)
FIRST    = first record of matrix to access
NUMBER   = number of records to access
INCREMENT = record number increment
IOCODE   = has one of two possible values:
           1) write operation
           2) read operation
ERROR    = the error status for the operation
```

```
> 0: ok
= 0: matrix does not exist
< 0: file operation error
```

The first record accessed is FIRST, then FIRST+INCREMENT, etc. until the I/O completes or a matrix boundary is reached. Since INCREMENT can be positive or negative, the boundary can be either the top or the bottom of the data block. Specifying a negative INCREMENT allows the record to be read in reverse order.

WORD MODE. In word mode, the matrix data block is considered as a list of short integers (integer*2). The routine for performing I/O in word mode is called as follows:

```
CALL MATIOW(LUN,MATNUM,BUFFER,FIRST,NUMBER,
            INCREMENT,IOCODE,ERROR)
```

where:

```
LUN      = the associated logical unit number
MATNUM   = matrix identifier (number or name)
BUFFER   = I/O buffer (read or write)
FIRST    = first word of the matrix to access
NUMBER   = number of words to access
INCREMENT = word number increment
IOCODE   = has one of two possible values:
           1) write operation
           2) read operation
ERROR    = error status for the operation
           > 0: ok
           = 0: matrix does not exist
           < 0: file operation error
```

FAST MODE. When repeatedly accessing the same matrix, the overhead caused by matrix lookup can be significant. Fast mode was designed to reduce this overhead. Before performing fast mode I/O an I/O identifier must be associated to the matrix. This causes the matrix parameters to be placed in a table where they are known to the package. I/O identifiers can be reused, but only the last association is known to the package.

When an I/O is performed on the matrix, the caller specifies the associated I/O identifier. The I/O parameters are then retrieved from the internal table without directory lookup. The directory lookup will be done only once, regardless of the number of I/O operations performed on the matrix.

A matrix is considered as a list of short integers (integer*2) in this mode.

Associating an I/O Identifier to a Matrix:

```
-----  
ERROR = MATIMG(LUN, MATNUM, IDENTIFIER)  
                                     (FORTRAN function)
```

where:

LUN = the logical unit number
MATNUM = matrix identifier (number or name)
IDENTIFIER = I/O identifier to associate
to the couple (LUN, MATNUM)
ERROR = error status:
 > 0: ok
 = 0: matrix does not exist
 < 0: file operation error

Reading from a Matrix in "Fast Mode":

```
-----  
ERROR=MATGET(IDENTIFIER, BUFFER, FIRST, NUMBER)
```

where:

IDENTIFIER = associated I/O identifier
BUFFER = user's I/O buffer
FIRST = first word to read from the matrix
NUMBER = number of words to read
ERROR = error status

Writing from a Matrix in "Fast Mode":

```
-----  
ERROR=MATPUT(IDENTIFIER, BUFFER, FIRST, NUMBER)
```

where:

IDENTIFIER = associated I/O identifier
BUFFER = user's I/O buffer
FIRST = first word to write
NUMBER = number of words to write
ERROR = error status

The I/O list defined by FIRST and NUMBER, for the MATPUT and MATGET routines can be any size. Therefore a very large matrix (e.g., a 1024 by 1024 image) can be read or written in one call.

During package implementation, it became obvious that for efficiency's sake I/O operations would have to be done using other facilities than simple FORTRAN READ and WRITE statements. Reading or writing large quantities of data requires fast I/Os. On the other hand, it was not desirable to use VMS features such as direct QIOs to the file ACP since this would have made package implementation sensitive to operating system changes.

The developed design allowed an intermediate solution: RMS block I/O mode. Both record and block I/Os are supported by the file design, at the RMS level. When I/O is to be done, two checks are

performed. First the routines determine if consecutive records or blocks will be accessed. If INCREMENT is different from 1 (MATIOR and MATIOW), this is not true. If it is true, the length of the I/O list is examined. If it equals or is less than two blocks, nothing special happens. If it is longer than two blocks, the package switches to block I/O mode, performs the I/O, and returns to record I/O so that following FORTRAN I/O statements work properly. This entire processing is completely transparent to the user.

It was experimentally determined that for lists smaller than three blocks, FORTRAN I/Os and block I/Os are equivalent. With this implementation scheme, I/O throughput was greatly improved. Comparative performance evaluation between this scheme and another involving direct QIOs to the VMS file ACP were performed. Degradation in performance due to RMS overhead was on the average 10%, which was satisfactory.

The last chapter of this article contains more information about the implementation of the block I/O mode on VAX/VMS systems.

E. Reading and Writing Main and Subheaders:

```
-----  
Reading or writing the file main header or any  
matrix subheader is done using the MATHED  
subroutine. A call to this subroutine has the  
following format:
```

```
CALL MATHED(LUN, MATNUM, HEADER, IOCODE, ERROR)
```

where:

LUN = the logical unit number
MATNUM = matrix identifier (number or name)
HEADER = 512-byte array to be written
into or to receive the designated
header record
IOCODE = two possible values: read or write
ERROR = the error status for the operation
 > 0: ok
 = 0: matrix does not exist
 < 0: file operation error

F. Write Protect Switch:

```
-----  
Matrices can be individually protected against  
write operations performed by the I/O routines of  
the package. Write protected matrix data can be  
read but not rewritten. Whenever an I/O module  
tries to write to a write protected matrix, it  
returns an error code. Since this switch works on a  
per matrix basis, it provides an easy method for  
protecting critical data from corruption while still  
allowing free access to non-critical data.  
Additionally, a write protected matrix cannot be  
deleted from a file.
```

The call to the routine that sets the write protect switch is as follows:

```
CALL MATWPS(LUN, MATNUM, SWITCH, ERROR)
```

where:

LUN = the logical unit number
MATNUM = matrix identifier (number or name)
SWITCH = set or reset write protect switch
ERROR = the error status for the operation
 > 0: ok

= 0: matrix does not exist
< 0: file operation error

4. VMS IMPLEMENTATION DETAILS

G. Miscellaneous Routines:

It is often necessary to check the existence of a matrix in a file or to know which matrices currently reside in a files. Also, it is sometimes useful to know the largest and smallest matrix numbers of the matrices in a file. The following routines provide these facilities.

1. MATTST (FORTRAN function)

TEST = MATTST(LUN, MATNUM, FIRST, LAST)

where:

LUN = the logical unit number
MATNUM = matrix identifier (number or name)
FIRST = if matrix exists, matrix subheader record number
LAST = if matrix exists, record number of the last record for the matrix data block

The result of the function is:

> 0 = matrix exists
= 0 = matrix does not exist
< 0 = file operation error

2. MATLST

This subroutine returns a list of matrix identifiers existing in a file. It ignores deleted matrices.

CALL MATLST(LUN, LIST, ERROR)

where:

LUN = the logical unit number
LIST = buffer to receive the matrix identifiers
ERROR = the error status for the operation
> 0: ok
< 0: file operation error

LIST must be an array of long words (integer*4) large enough to receive all the matrix numbers extracted from the file. The end of the list is marked with a zero. The matrix numbers are returned as they are found in the file.

3. MATMIN and MATMAX

These two FORTRAN functions consider the matrix identifiers as numbers. They return the largest and smallest matrix numbers that exist in a file.

MAXNUM = MATMAX(LUN)

or

MATMIN = MATMIN(LUN)

where:

LUN = the logical unit number

This chapter gives some specific details of the implementation of the MATRIX package on VAX/VMS systems.

A. Opening and Closing MATRIX Files:

We mentioned previously that MATRIX files must be opened and closed using special routines. One reason for this is to allow block mode I/O. In order to do this, the File Access Block and the Record Access Block of the designated file must be updated to reflect that both record and block I/Os will be performed. This can be done by opening a file with a FORTRAN OPEN statement including a USEROPEN clause. This OPEN statement performs the allocation of the FAB and RAB and then passes control to a user-specified routine which makes the desired changes to the two access blocks and opens the file. More details on this process can be found in the VAX FORTRAN manual and the RMS/VMS manual.

To perform block I/O, we need the address of the Record Access Block (RAB). During execution of the user-specified routine, the address of the RAB is saved in an internal table of the package along with its associated LUN. When a block I/O is performed, the address of the RAB corresponding to the specified LUN is fetched from this table. This is another reason why MATST must be called to open a MATRIX file. The internal table has a limited, although large, number of entries. MATST checks that one is available when opening a new file and if so, opens the file.

When MATFIN is called, the file is closed and the allocated entry in the RAB address table is released. If the file was not close with MATFIN, this entry would never be released.

B. Long Block I/Os:

One of the limitation of RMS is that it does not allow I/Os longer than 64K bytes. This can be inconvenient for applications dealing with large matrices (e.g., 1024 by 1024 images). The block I/O routines (MATPUT and MATGET) subdivide I/Os longer than 64K bytes into several I/Os of 64k bytes (or less for the last one). This is one reason for the performance degradation observed in comparison to direct calls to the VMS file ACP. This degradation (less than 10%) was not considered severe enough to warrant a more efficient scheme. The developed system has a satisfactory throughput, a relatively simple implementation, and a high level of independence with lower levels of the VMS operating system. For example, the package supports, with no additional coding, remote file access through DECNET.

C. RSX vs. VMS Implementation Differences:

The MATRIX package is also available on RSX systems. The RSX implementation is completely compatible with the VMS one and files produced by

one package can be processed by the other. However, block I/O mode is unavailable on RSX systems, which makes the system inefficient when dealing with large data sets. In addition, the package contains a significant amount of code and given the limited address space of RSX systems, this could make its use quite cumbersome.

5. CONCLUSIONS

The MATRIX package was used by our group for the implementation of a large software system for medical image processing. It is a convenient package that serves as the base for our file organization. In our system, all files, except ASCII files, are MATRIX files. The MATRIX structure also serves as the basic layer for upper level file organizations we developed. It has proven to be a simple, versatile and very efficient tool.

6. REFERENCES

- VAX/VMS Service Routines: Record Management Services
- VAX/VMS I/O User's: Part I
- VAX FORTRAN 77 User's Guide

DATATRIEVE/4GL SIG

DATATRIEVE-11 to VAX-DATATRIEVE Conversion Panel

Joe H. Gallagher

Research Medical Center

Kansas City, MO 64132

Bart Z. Lederman

Greenberg Bros. Part.

New York, N.Y. 10010

Session Chairman:

Alex L. Lamb

USASATCOMA

AMCPM-SC-4G

Ft. Monmouth, N.J. 07724

Transcribed by B. Z. Lederman

This session was originally planned to be a recounting of users' experiences in converting to VAX-DTR from DTR-11. Due to various difficulties which arose between scheduling the session and the symposium (such as non-functioning equipment, and jobs being eliminated), the final panel had to speak partially from experience and partially on generic terms. Nevertheless, many important points were covered, and the information was judged to be quite useful by those attending the session. This is not intended to be an exact transcription of the session: rather, it simply presents the information in a readable form. I will not attempt to distinguish which person made what contributions. I would also like to acknowledge the presence of Suellen Harris and Bill Opalka from DEC, who sat in the audience and kept us from going astray.

Major Points.

DTR-11 is an almost perfect subset of VAX-DTR.

DTR-11 is a nearly perfect subset of VAX-DTR. (It was, of course, developed first, and VAX-DTR was developed later using the same syntax so users could migrate.) The primary point of difficulty has to do with a difference in how the hardware addresses memory. The PDP-11 cannot address a word which starts on an odd byte boundary, while the VAX can: therefore, such data types as REAL, DOUBLE, INTEGER, DATE, and other word, double word, and quad word variables, must be aligned to word boundaries on the PDP-11. DTR-11 will automatically do this by inserting invisible bytes where needed. Consider the following record definition:

```
01 RECORD.  
  10 FIELD1 PIC X.  
  10 FIELD2 PIC 99 USAGE IS INTEGER.  
;
```

FIELD1 is one byte long, and FIELD2 is two bytes long. In VAX-DTR, this record would be 3 bytes long: on the PDP-11, it will be 4 bytes long, as an extra byte has to be inserted between the two fields to make FIELD2 start on an even byte boundary. DTR-11 will do this wherever necessary (and only

where necessary), and does not issue warnings or error messages. This will become apparent if the data files are moved from one system to another, and the same record definition is used on the VAX without taking precautions. It is possible to make VAX-DTR allocate on word boundaries by using the clause

"ALLOCATION IS LEFT-RIGHT"

within the record definition; therefore, you should change the record definition to:

```
DEFINE RECORD TEST_REC  
USING  
ALLOCATION IS LEFT-RIGHT  
01 RECORD.  
  10 FIELD1 PIC X.  
  10 FIELD2 PIC 99 USAGE IS INTEGER.  
;
```

If you do this, VAX-DTR will allocate space in the same way as DTR-11, and there should be no problems. If you don't, you may get an error message similar to "RECORD LENGTH DOES NOT MATCH" when you try to READY the new domain on the VAX, which is an indication that you may have an alignment problem. See also the section on moving data below.

If you do get these messages, and don't do anything about it before storing new data, you will corrupt the data file. It would be a good precaution to be certain that the first time you READY the domain after moving the data that you do so READ ONLY, and look at the data. If there are any alignment problems, you will immediately see that data isn't coming out correctly, and can take corrective measures.

Planning the move: more dictionary options.

The Common Data Dictionary has more functionality than the Dictionaries on the PDP-11, and some thought should be given to using this. On the PDP-11, there is often a separate dictionary for each application, and the data files often aren't shared. On the VAX, all definitions go into the CDD, and makes sharing easier. The CDD has a hierarchical structure, and if you have many separate projects you will want to plan which sub-directories should hold what definitions, and which definitions should go into a site common directory (for everyone to use), which should go into project wide directories, and which may be left in individual directories. The protection options are also greater on the VAX than on the PDP-11: if you are not using protection now, then you can move with no protection without making any changes. If, however, you have been using UIC type protection on the PDP-11, then you may want to change, as it is unlikely that you will keep the same UIC scheme on your VAX disks that you had on your PDP-11 disks. (VMS does have UICs, but most people go to named directories and hierarchical sub-directories, as these tend to be easier to use and more closely follow the way most data is organized.) Therefore, you may want to change UIC and/or password protection, and use some of the new protection options available in VMS.

You do not have to learn much about the CDD before moving if you do everything in DTR. You can put everything into one dictionary like DTR-11, but you will get better performance if you make a good division from the beginning. You can move definitions from one sub-section to another later, but it's best to make some plans first. A quick read-through of the manual to understand the concepts, or attending some DECUS sessions are good ideas. For the new user, no new training is needed to begin with, as they will see a dictionary just like what they are used to (with the addition of version numbers), and can see one big dictionary or their own dictionary, whichever matches the PDP-11 installation; but the person planning the move and organizing the application should know something about the CDD. The CDD structure can copy the existing PDP-11 setup, but usually matches the company or application organization. Arranging the dictionary well has both performance and maintenance advantages. Note also that VAX-DTR can have startup files for individual users just as DTR-11 has, to put individuals into the proper dictionary and perform READYS or start procedures, as needed: also, the dictionary that the user sees for PLOTS can be different from that for all other dictionary elements, so the users can get all of their domains, records, procedures, and tables from an individual dictionary, and at the same time get the PLOTS from the system wide dictionary. You might also give them a procedure to set their dictionary into the

system wide DEMO library if you want them to play with YACHTS, EMPLOYEES, or other sample files supplied with DTR. There is also a command procedure NEWUSER.COM that comes with DTR that can be used to easily set up a dictionary for a new user. In addition, there is also a logical name that designates the CDD dictionary in which the user will start.

Moving the data.

One must move both the data files, and the dictionary objects (record definitions, domain definitions, tables, etc.). You can EXTRACT individual definitions, but DTR-11 is supplied with a utility called QXTR, which will extract all of the definitions in a dictionary at one time into a single command file in a manner similar to EXTRACT ALL in VAX-DTR. This utility inserts the "ALLOCATION IS LEFT-RIGHT" clause into the record definitions, which should simplify the move considerably. [This is probably true only for DTR-11 V3.0 and V3.1, and not for earlier versions of DTR. QXTR is not supplied with PRO-DTR. DTR-11 itself does not insert the ALLOCATION clause when extracting.] If you are using individual dictionaries for different users, you must extract each dictionary individually. QXTR also gives you the option of extracting the protection qualifiers, or not, as you choose. This single file can then be moved to the VAX and invoked to re-create all of your records, tables, domains, etc. One factor not covered in the original talk is that explicit references to disks and/or directories in domain definitions will probably have to be changed, as your device names will change, and you will probably be using symbolic device names and/or named directories, as mentioned before. To actually get the information across, there are several options.

Moving data on disks.

If you are using RSX (which in this paper includes 11M, 11M-Plus, IAS, and possibly 11D), and you have the same kind of disk drive on both systems, or are going to move the disk drive to the new system, then you can read your old disks on VMS. You will want to copy the information to a new disk, to take advantage of some VMS features like named directories, but the files can be read while on the old disk, or copied. This applies both to the data files, and the file created by QXTR (or the DATATRIEVE EXTRACT command). If you are running RSTS/E, then you are out of luck: no other operating system will read your disks. If you don't have a disk type in common on both systems and are not moving the devices, you might want to consider plugging in your old disk on the VAX just long enough to copy the data, if possible. If not, then the local DEC office may be able to copy the disks, or your local DECUS LUG may help you find a user who may let you do it, or you may be able to locate a commercial service company that will do it.

Moving data by Network.

If both systems run DECNet (no matter what the operating system), then any file that DATATRIEVE can read can be read over the network. You can COPY (or NFT or FTS) the file with the definitions to the VAX, define your domains, copy the data file, and be

ready to go. Alternatively, you can define a two domains with the same record definition, with one having a normal file specification on the VAX, the second having a file specification which includes the node name for the PDP-11 (something you can do with VAX-DTR but not DTR-11); or what may be better, if you have remote DATATRIEVE installed on the PDP-11 you can use the "READY domain AT node" feature of VAX-DTR, and simply read from the old domain on the PDP-11 to the new domain on the VAX. This will be a little slow, as DTR is not optimized for this type of operation the way DECNet utilities are, and the data will be going over the network, but hopefully it will only be done once. The advantage of doing this is that it can avoid the problems of record alignment mentioned before. By using the DTR-11 remote server on the PDP-11, the data will be read exactly as it has always been read: on the VAX side, you can define a record without the ALIGNMENT clause, and pack the data in without worrying about hidden bytes. You may also want to consider other methods of converting data mentioned below. Remote DATATRIEVE is available starting with V3.0, not in earlier versions. If you are not using DECNet, you may not have special communications devices normally used for networking. DECNet will work over normal asynchronous lines used for terminals, however. You may have to give up 8 or 16 lines for a while, as DECNet grabs the entire device on PDP-11s, and the maximum speed may be 9600 baud so transfers may be a little slow, but hopefully you will only do the conversion once. If you don't have DECNet, maybe you can persuade the local office to let you use it just for a few days while you move your data.

Moving data on Magnetic Tape.

If both systems have magnetic tape (or you are moving the tape drive), you may be able to store the definitions and information on tape. Once again, the RSX family provides the best compatibility, with most systems writing ANSI tapes which can be read by VMS. (On earlier RSX systems, this was a SYSGEN option, so check carefully before you write out the tapes and disconnect your PDP-11.) If not, you can write DOS-11 format tapes on all RSX family systems which can be read on the VAX with FLX (in compatibility mode) or the new EXCHANGE utility. Though you might possibly get indexed files over if transferred in image mode it will be much safer if you first convert the data to a sequential file and re-construct it on the VAX, as will be mentioned again later. If you have RSTS/E, then you are again stuck with a system which works differently than everyone else. You might be able to generate DOS-11 format tapes, or you may want to upgrade to V9.0: this is the latest version of RSTS/E, and it includes a utility which writes tapes that are compatible with the VMS BACKUP utility. For RSTS/E users this will probably be the easiest way to move files, though V9.0 is a new release and those of us on the panel have not yet had any feedback from users. Moving data from RSTS/E to other systems can be so much of a problem that some people plan to upgrade to RSTS/E primarily for the purpose of being able to use the new utility to write VMS readable tapes with the new utility.

There is an RMS utility set, BCK and RST, which is intended for backing up files to and restoring files from magnetic tape. These utilities automatically convert indexed files to a form which

will store properly on magnetic tape, and place error checking and other useful information on the tape for all types of files. Because they are supplied as part of RMS, they should be available on all PDP-11 systems (that can run DATATRIEVE). The problem is that VMS may not have a utility corresponding to RST to get the data off: so unless you have compatibility mode on your VMS system (and it is now an optional layered product), this may not be a viable option.

Other file transfer methods.

If none of the above are available, you may want to look into some of the communications packages which will operate over normal serial (terminal) lines. KERMIT is a public domain program available from the DECUS library (and elsewhere) which runs on virtually any computer and operating system, will work on serial lines, does error checking, and can transfer (in most cases) both text and binary files. It may be slow, but it will work. There are also other communications packages that perform similarly: you may even be able to use SET HOST/DTE/LOG on the VAX to go in to the PDP-11 and type the file out (this works best if the data is all text) or a similar "dumb" text transfer.

Re-Organizing your data.

VMS has some options not available on the PDP-11 for indexed files, notably Prolog-3, which allows some space saving. Since indexed files should be re-organized occasionally for best performance, and since the same data will occupy less space in a sequential file and take less time to transfer over networks, and transfer more easily on magnetic tape, the time of transfer would also be a good opportunity to re-organize and optimize the data file by converting it to sequential, and re-populating an indexed file on the VAX. Even if you can move your disk packs and can read your old files directly, re-organizing at this point is a good idea, though in this case you don't have to convert to an intermediate sequential file.

First, you need a description of the current indexed file: you can simply make a note of the record length (which you get from DTR when you define the record), and figure out where the keyed fields are in the record, but there are two RMS utilities, the older DSP and the newer DES which are designed to record file characteristics and define new files: it's a good idea to have such a description file for documentation purposes, even if you aren't anticipating a conversion. DES creates a file definition which is quite similar to that used by the VMS FDL utility: in actual tests, I was very surprised to find that FDL will actually read the descriptor file produced by DES! It may give you a few warning messages, especially if the SOURCE and TARGET fields are empty, but this shouldn't cause any serious problems. If the utility refuses to read your descriptor file and aborts, check to see that when you transferred the description file over it did not pick up trailing blanks on the description items, and that the file position qualifier says NONE with no numbers trailing. You can use a regular text editor such as EDT to go over the file before reading it with EDIT/FDL if necessary to touch it up. If you don't take this approach, then you can move the record definition,

define a domain and define a file: DTR will create a file that matches the record definition. You can then look at that file with EDIT/FDL if you wish, or do the optimization shown below. If you move your disk packs FDL can probably read your original data file, and may also be able to do this over the network if you are using DECNet.

Converting the data file from indexed to sequential on the PDP-11 is quite simple: the CNV utility will perform this conversion by default. Simply give the name of the indexed file on input, and the file name you want for the sequential file on output.

Once the file description and the data file are on the VAX, you can populate the file with CONVERT: however, now would be a good time to review the file design for possible performance improvements. the FDL utility (EDIT/FDL) has an OPTIMIZE script which can make the decisions a little easier, as it will look at your data file and give you some information about what can be done to improve access. The one factor which comes up quite often is bucket size: on the PDP-11, bucket size is almost always the smallest possible value that will hold the record size you are using, as larger buckets use up pool space. On the VAX, this is no longer a problem, and larger bucket sizes are a viable option. If you often retrieve records which are next to each other, such as retrieving a record by the primary key and then reading the next several records in order, then a larger bucket size may improve performance. If, however, you retrieve records scattered all over the file in no particular order, then a larger bucket size won't help and may hurt, but a different index structure may be of benefit. If you don't know what options to take, use the FDL utility and let it optimize: it will usually take reasonable options. Once this is done, you can populate a new file again with CONVERT.

Combining related domains.

Another consideration in DTR-11 is that the data is sometimes separated into several domains rather than one, to prevent the record definition and buffers from getting so large as to use up all of your pool space. When converting to VAX-DTR, you may want to recombine them by moving all of the individual files (and domain definitions) over, define a VIEW to combine them and a single record definition with all of the fields, and read from one into the other. This may be a little slow, but again, this will only be done once. You will then want to use FDL as described before to optimize the new combined file.

Reports.

For reports, the VAX-DTR works very much the same as DTR-11. If you have reports which are adjusted so that field breaks occur just at the beginning or end of a page, or are otherwise 'finagled' to match a pre-printed form, you may find that you have to do a little adjustment to the lines per page or number of lines skipped qualifiers, but most straight forward reports will work with no changes.

Tables.

While VAX-DTR has dictionary tables just like DTR-11, it also has domain tables: this allows data in a domain to also be accessed like a table. One field (preferably a keyed field for good performance) takes the place of the "left" side of the table, and another field (any one in the domain) takes the place of the table entry on the "right". Dictionary tables are faster for small tables: domain tables can be faster for large tables if keyed fields are used, can take the place of several tables, and can also be accessed as a regular domain, which makes changing the table the same as modifying the data in any other domain; therefore, tables which are modified often are easier to maintain as domain tables.

Functions.

VAX-DTR has all of the functions (MIN, MAX, TOTAL, etc.) that DTR-11 has, plus several more (standard deviation, and running count, for example). In addition, there is a new set of functions of the FN\$xxx family which can do things like convert lower case text to upper case, move sub-sections of text strings, format output strings, get system information, do mathematical operations, and many other functions. If that isn't enough, you can add your own functions. In many cases, things that you are doing, perhaps with some difficulty, with procedures or COMPUTED BY clauses may be much easier with the extra functions available in VAX-DTR, and a review of what you are doing may reveal some areas where improvements may be made. Though you may want to wait until your application is moved over and running, this is an area where you will want to do some work quite soon after the move

No Pool Space Restrictions!

In DTR-11, procedures are often broken into small pieces to conserve pool space, and more things were done in command files. You may consider moving more of the work into DTR procedures, especially when you can get large blocks into single WHILE or BEGIN-END blocks. This may take a little longer to compile, but once compiled will execute faster than separate blocks.

In DTR-11, especially if your application was just on the edge of available pool space, things tended to be pared down to the minimum, especially in record definitions. Variable names were very short, edit-strings removed, etc. On the VAX, you have the opportunity to put more descriptive field names back in, and make the fields more self-documenting, put in query names, edit strings, and so on. Don't drop your good habits, however. Some users think they have infinite space on the VAX, and don't FINISH unused domains or RELEASE unused variables. This will cost you something eventually on the VAX (usually in paging), and it won't be as obvious as running out of pool space was in DTR-11: your application, or the whole system, just slowly degrades. You also may interfere with other users who want to access the data if you are keeping files open and are possibly locking records. You can keep domains open if you expect to use them again and save the time it would take to READY the domain, but when you are done with a domain you should FINISH it.

Other improvements.

All users will be happy to learn that VAX-DTR uses EDT when editing, rather than the built-in editor of DTR-11. If you have an EDTINI.EDT command file in your account, it will be used when you edit something in VAX-DTR; keypad and all other commands will work, etc. You will also notice that with the newer versions of VAX-DTR, you have version numbers on definitions, so you can keep the old definition around until you find that the new version works. Users should be reminded to purge out old versions regularly.

An option available in VAX-DTR is the use of FMS or TDMS to have form driven screens. After your data is on the VAX and working, you may want to start thinking of converting some of your old procedures, especially if they were working like menus, to form driven screens. Something you do need to consider before you do this, however, is whether to buy FMS or TDMS: they look very much the same to DTR, so the choice is often determined by what other software you are using. Some packages will require one or the other, (for example, All-In-One uses FMS) while DTR can work with either, one at a time.

There are various other features that are present in VAX-DTR (three types of concatenation rather than two, CROSS statements, the CHOICE statement, and all of the graphics capabilities), which you will soon discover and will want to incorporate into your applications; but none are needed immediately for conversion, and you can wait to learn about them until after the problems of conversion are over.



COMPUTERIZED DECISION SUPPORT FOR COLLEGE ADMINISTRATORS

Walter H. Frey
Vernon M. Cline, Jr.
Carl Albert Junior College
Poteau, Oklahoma

The development of a Decision Support System for college administrators is described. The system is based on integrating a microcomputer with a DEC PDP 11/44. Software on the microcomputer expands the capabilities of the PDP 11/44 and its resident software. Novice computer users in the administration can access the college data base using plain English commands to produce a variety of information in different formats.

INTRODUCTION

This paper describes our efforts to provide a Decision Support System (DSS) for the administrators of Carl Albert Junior College. We define DSS as the interactive use of computers by the administrator to get information in an interesting format, when it is needed, to promote insight and the probability of making better decisions.

Two major problems had to be solved in this process: First, naive computer users, had to get access to data in a simple, "user friendly" manner. Second, the old minicomputer and its resident software could not be replaced due to budgetary constraints. In this presentation we will discuss:

1. The hardware and software configuration.
2. The advantages of using a microcomputer.
3. The downloading process.
4. Data manipulation on the microcomputer.
5. The payoff - DSS for administrators.

CONFIGURATION

HARDWARE - MINICOMPUTER

Our main computer system is a DEC PDP 11/44 with 4M bytes of memory. There are 670M bytes of disk storage available and a TS11 tape drive for backup. Sixty-four

peripherals are connected to the system. These user devices range from old VT52 terminals to microcomputers and a LN03 laser printer. An average of 30 to 40 users are on the system at any one time.

SOFTWARE - MINICOMPUTER

The computer operates under RSTS/E Version 8.0. Data on the system are managed by the POISE data management system. The data are collected in various application packages: Registration and reporting, student billing and receivables, financial-aid reporting and fiscal reporting. The payroll and personnel packages on the system were produced locally using the POISE computerized programmer. The other packages are customized POISE programs.

HARDWARE - MICROCOMPUTER

The system is an IBM compatible personal computer with 640K bytes of memory. It has two double density disk drives, which will be expanded by the additions of a 10M byte hard disk drive to better handle administrative requests involving large blocks of data. The system has a color graphics card, color monitor and a dot matrix printer.

SOFTWARE - MICROCOMPUTER

The operating system of our microcomputer is PC DOS Version 3.0. We have an integrated spreadsheet available for spreadsheet, graphic and limited data base functions. We also have a relational data base management system and an English language inquiry program. Downloading, communication, and MACRO key capabilities are provided by an emulator program.

ADVANTAGES OF THE MICROCOMPUTER SOLUTION

The biggest advantage of the microcomputer approach to DSS implementation was the cost, which was under \$6000.00 for hardware, software and computer services' time. This compared to alternative costs of \$20,000.00 or much more to get similar or improved results. A second advantage is that our 11/44 now has graphic capabilities and any suitable data on the system can be represented graphically. The relational data base management system allows us to access several POISE files at the same time and the English language inquiry program allows administrators to access their data in plain English.

THE DOWN LOADING PROCESS

THE MICRO - MINI CONNECTION

Because all the buildings on our campus are within 500 feet of the computer center, we have hardwired (null modem wires) all our terminals to the PDP 11/44. Wiring the microcomputer follows the same procedure used for a VTL00 terminal. Standard RS232 connectors are utilized, with pins 1, 2, 3, and 7 connected.

COMMUNICATIONS AND EMULATION PROGRAM

Several features are desired in this program and all but one are included. The program has to emulate a DEC VT series terminal and communicate with the PDP 11/44. Key MACROS can be produced so that data extraction and downloading are accomplished transparent to the user. Data are transported from the PDP 11/44 to the Micro's diskettes as ASCII delimited files. One feature we wanted, but could not find in the emulators we examined, was scripting. This feature would allow fuller automation of communication between the computers and simplification of operation for the user.

DATA SELECTION PROGRAMS ON THE PDP 11/44

Selection programs for the POISE data files are produced by our systems analyst/programmer in BASIC. These programs take the contents of selected fields in POISE data files and assemble them into ASCII delimited files for downloading.

In response to a structured request by an administrator, certain selection programs are activated to collect data from predetermined fields and files. These data are then displayed on the microcomputer in a standard format. Other selection programs, which are activated by an unstructured request, collect as many data fields from a file as can be stored on the

microcomputer. This larger, varied, data selection can then be manipulated on the microcomputer to produce information in an exploratory manner.

MICROCOMPUTER DATA MANIPULATION

All downloaded data is in the form of ASCII delimited files which are read into the relational data base management system. We chose RBASE 5000 by MICRORIM because of the ease with which a data base can be created with this program. Another major factor for choosing RBASE was its compatibility with an English language inquiry system. RBASE can output data as an ASCII delimited file. This capability is essential for smooth transfer of data to the spreadsheet program.

If the administrators cannot get the desired information from the data base manipulation, they may choose to have the program, via MACROS, transfer the data to an integrated spreadsheet program. We chose SYMPHONY by LOTUS Corporation, because it could accept data in our standard file format - ASCII delimited. Furthermore, SYMPHONY is easy to use and produces a variety of graphic output. Best of all, many of its functions can be made transparent to the user through the use of MACRO commands. Several companies have built program "shells" for SYMPHONY. These outline programs allow "what if" analysis, projection, and more with downloaded local data.

CLOUT, by MICRORIM, is the English language program we have chosen for the system. It is completely compatible with our relational DBMS and is truly user friendly. An administrator can become reasonably proficient in its use in one hour. The program is very forgiving of user's quirks and will adapt to the jargon of the user rather than requiring the strict syntax of the data base. Furthermore, it has the capacity to refer to multiple files within the database.

THE PAYOFF - DSS FOR ADMINISTRATORS

From the microcomputer terminal at his desk, the administrator can get direct access to the college database. He can do so without extensive training in syntax or computer esoterics. Structured, recurring information demands are filled when needed. Confidential data and "what if" analysis are perused by the administrator in privacy without interference of intermediaries. Information is explored in a variety of formats, quickly and easily leading to better understanding and greater utilization of the data.

In the case of novel, unstructured demands, computer services personnel can produce necessary modifications in a very short time (hours). This short response time provides administrators with the information while it can best be used to enhance the quality of a decision.

SUMMARY

Computerized decision support is being provided to college administrators. This has been accomplished at a cost of less than \$6000.00. The system can handle both structured and unstructured administrative requests. The English language inquiry option and the use of MACRO commands allows computer-naive administrators to use the system with ease. The system has released computer services personnel time by allowing administrators to serve themselves. Administrators are more satisfied because request flexibility is increased and turnaround time is reduced. The weakest link in the system at this point is the lack of scripting in the emulator program.

EVALUATING, SELECTING AND IMPLEMENTING AN ON-LINE LIBRARY CARD CATALOG

Rob Robinson
Northwestern College
Orange City, IA 51041

ABSTRACT

This paper will include a description of our assessment of our needs for library computing services, the process of evaluation of available software, our efforts in data conversion, and our implementation experiences with the BRS/SEARCH database program. It will include a discussion of the side benefits we've received elsewhere on the campus with text and information oriented database software. It is our desire that this paper may be used as an evaluation of the BRS/SEARCH program and as a guideline for other colleges and libraries who are considering similar projects.

INTRODUCTION

Northwestern College is a Christian liberal arts college with 850 students, located in the northwest corner of Iowa. Ramaker library, one of the newer additions to the campus, holds 72,000 titles. An additional 5,000 items of audio, visual and computer software are located in the Learning Resource Center in a separate building on campus. Both collections are managed and circulated by the library staff.

In the fall of 1983, NWC planned to move it's academic computer users from an administrative PDP-11/44 system to a new academic computer. At that same time, they also began a search for library automation software. The search for library automation software and the resulting implementation are detailed below, including our needs assessment, the evaluation and selection process and the implementation process.

NEEDS ASSESSMENT

The first step of our needs assessment involved listing the desired features

that a library automation package should have and setting the priorities for each feature. The top seven features in order of importance were:

1. On line card catalog
2. keyword searching and boolean logic
3. Local entry cataloging
4. Full MARC record searching
5. Circulation management
6. Serials management
7. Acquisitions management

The listing of these priorities indicate a major difference that exists between university libraries and public libraries. The emphasis at a university library is on the access to information rather than the management of the circulation. At a university, finding relevant material is the biggest job. At a public library, Circulation is the biggest. A typical patron of a public library will have a specific title, or author in mind when searching for a book and will usually check the book out to read at home. A typical patron at a university library will have a topic in mind, and will want to find a fairly sizeable list of books which address that topic. The books are usually used

in the library and not checked out. Consequently, a public library might have features five through seven at the top of the list rather than the bottom.

Another important part of our needs assessment was to choose between using a central data base on somebody else's computer or a local data base on our own. Initially, libraries created centralized databases such as the OCLC system. The high cost of computers and storage devices easily justified this move. Continuing high costs in telecommunication costs and falling costs for disk storage along with cheaper computers are creating a trend towards the use of a local data base right in the library. It was our desire to follow this trend and develop our own in-house database.

A third issue for our needs assessment was to choose between purchasing a separate computer for the library, or purchasing an academic computer which would service both the library applications and the teaching and research applications for the faculty and students. We felt that since the library is used by over 99 percent of the students and faculty, that it should be accessible from any terminal on campus. and it seemed more efficient to do that with one computer rather than trying to network two computers together.

Each of these issues in our needs assessment strongly impacted our evaluation and selection of library software.

SOFTWARE EVALUATION

The initial stages of the software search was done almost entirely by the librarian, using the idea that the software should be considered first without regard to the type of computer it runs on. This approach was very successful in giving a thorough view of what is currently available in the best of library automation systems. The most significant aid in this search was the book entitled Public Access to Online Catalogs (Joseph R. Mathews, Online Inc., 1982) giving a very good survey of

currently available systems. Conferences on library automation and colleagues from other libraries also helped to make the search. After this initial evaluation, the possible choices were narrowed to the following six packages:

NOTIS by Northwestern University
LS-2000 by OCLC Inc.
VTLS by Virginia Polytechnic Inst.
PALS by Minnesota State University
ATLAS by Data Research Associates
BRS/SEARCH by BRS Inc.
DOBIS-LEUVEN by IBM Corp.

Two of the systems were available only on large mainframes and one was developed on a minicomputer which did not have strong support in our area. Consequently, the list was reduced to four systems including the VTLS system on an HP-42 Hewlett Packard computer, DOBIS LEUVEN on an IBM 4331 computer and BRS/SEARCH and ATLAS which were both available for the DEC VAX. These three computers were also our final three candidates in our search for an academic computer. Each of the four final packages were studied in detail through demonstrations and telephone calls. They differed a lot in features and functionality, but each had enough good features, that the librarians were confident that a successful system could be developed with any one of them.

The final choice could not be made until the entire academic community had agreed on a computer system. Choosing an academic computer is always delicate, because the needs and uses are so many and varied. Sacrifices and compromises have to be made, because no computer rates first in every discipline. We felt that we had three main groups which should have equal weight in a decision: the computer science department who wanted power and up-to-date technology, a large group of faculty who wanted a user friendly system, and the library who wanted a special application. The VAX was chosen because it came the closest to meeting everybody's needs.

After selection of the computer, two software packages remained and were very different in features and functionality. The ATLAS package by DRA had a very good

system for circulation, serials and acquisitions but was weak on searching capabilities. BRS/SEARCH was very strong for searching capabilities but had no circulation, serials or acquisitions package. Our needs assessment played a valuable role at this point. With their priorities firmly in mind, the library decided that the searching capabilities were well worth the sacrifice of the other features. Since our purchase, The SIRSI corporation has integrated BRS/SEARCH into their total library system. The SIRSI corporation package is available for unix and Xenix based systems, but not for VMS based systems.

PREPARING THE DATA

In 1978, Ramaker Library installed an OCLC terminal and modem which was connected to the OCLC cataloging service in Ohio. The first person to catalog a record with OCLC enters all the information about the book via the keyboard. Future catalogers can recall that record and will only have to add their own local information to it. Once cataloged, the local version of the record is left on-line for two weeks, then archived onto a magnetic tape. A copy of the archived tape may be requested at anytime.

We started cataloging and creating MARC records in 1978, and had all 72,000 of our main holdings entered. Many libraries used this service to catalog only new purchases. The process of going back and cataloging previous books is called retroconversion, and if done by an outside service can cost as much as \$1.25 per title. Fortunately for us, the retroconversion project was done by our own staff in the early years when there was no charge for cataloging with OCLC.

The state library receives tapes each quarter from OCLC for holdings of all libraries in Iowa. Although tapes can be ordered directly from OCLC, we were able to save on costs by having the state library select our records from their tapes and send us a tape with only our holdings.

Defining The BRS data base was the next step. BRS/SEARCH uses the terms document and paragraph for database definition. The actual content and meaning of these terms are left for the user to define. We defined a document to be one MARC record, or one card in the card catalog. A paragraph is one field or logically related set of information. A paragraph may be a single number such as publication date, or a single line such as the call number, or several lines such as the title, or several text paragraphs such as the contents notes. A paragraph is preceded by a paragraph label. We used TI for title, AU for author, SU for subject, etc.

We followed the model used by Dartmouth and Purdue Colleges as a guideline in formulating our own paragraphs. (The appendix contains a list of the paragraphs that we chose.)

THE DATA CONVERSION

The catalogued records from OCLC come on a one-half inch, nine track, 1600 bpi magnetic tape. They are recorded in MARC format which is a standard set by the library of congress for catalogued records. Details of this format may be obtained by ordering the manual "OCLC-MARC Subscription Service Documentation" (OCLC Inc., Dublin Ohio, 1981). The data conversion process was the most time consuming and labor consuming part of the implementation. The MARC record contains a 24 character field of fixed information, and from 1 to 300 variable length tag fields, each with a different meaning. We had to decide which BRS paragraph should be used for the information from each tag field in the MARC record. Once converted, the BRS records were written onto an ASCII text file with an appropriate identifier in front of each paragraph. The following functions were performed during the data conversion process:

1. Fixed fields and tagged fields from the MARC record were grouped into BRS paragraphs.

2. Foreign language symbols (umlauts, etc.) were converted to ease the searching process.
3. A sortable field was created from the call number making it possible to generate a list of the books in shelf list order.
4. Inconsistent abbreviations were converted to consistent ones. (US, U.S., USA, United States, etc. were all converted to United States)
5. Improperly connected words were unconnected.
6. The format was standardized for the paragraphs for imprint, subject, collation, and contents notes.
7. Duplicate records were checked for and weeded out, taking only the most recent version.

Writing and designing the program took 200 hours. Running the programs took 22 hours of connect time and 7 hours of personnel time. We actually made two passes and reconverted the database after a few months. The necessity for Items 4-7 above was only discovered after we used the data for several months. Dartmouth College indicated that they have done this same process four times, so it's apparently fairly common with a large database.

INSTALLING THE BRS SOFTWARE

The software comes in executable load modules which are ready to use as soon as they are copied from the tape. Some editing of text files is required to define system hardware characteristics and database storage locations. The process took 4 hours for reading and preparation and 4 hours to install.

LOADING THE DATA BASE

We loaded our entire 71,000 records at one time with one command to the BRSLOAD program. The initial load on an otherwise idle VAX-11/750 took 46 hours of

connect time. We later revised the data conversion program and reloaded the database during the middle of the semester when use was moderate, requiring 60 hours of connect time.

The size of the source file for the database was 50 megabytes. After the database was loaded and the dictionaries and inverted index files were built, the database occupied a total of 70 megabytes, or almost 1000 bytes per record. During the middle of the load process, 120 megabytes of space were used, with the extra 50 megabytes being used for temporary work space.

PROVIDING EQUIPMENT

The following equipment is currently required by the library for BRS/SEARCH.

- 4 dedicated CRT terminals for public.
- 1 printer in the library.
- 1 CRT for the reference librarian.
- 1 CRT for technical services.
- 1 modem for dial-up
- 1 set of 8 communication ports
- 1 RA81 disk (250 MB for library use)
- 2 megabytes of memory

USING BRS/SEARCH

Searching

To search for a single word with BRS/SEARCH, just type the word and press RETURN. Searching for a single word results in an almost instant response of less than 1 second showing the number of documents containing that word. To search for combinations of words, type the words using a boolean operator like AND or OR in between each word. Using a boolean operator with two words requires more time, usually less than five seconds, but it depends on the number of individual occurrences of each word. In addition to the standard boolean operators like AND, OR and NOT, the special operators ADJ, WITH, SAME, and NEAR may be used to find words which are located next to each other or are close to each other. For example, the operator NEAR4 will select a document if the requested words are

within 4 words of each other. The SAME operator finds documents in which the words are in the same paragraph.

Parentheses may be used to construct statements that are as logically complex as the user can handle. For example, the statement:

(ONE OR SINGLE) AND (PARENT OR PARENTS)

will find in one search command, most documents which mention something about single parents. For the more typical person who has problems with complex logic statements, individual searches can be combined. For example, the requests:

- 1: ONE OR SINGLE
- 2: PARENT OR PARENTS
- 3: 1 AND 2

will find the same set of documents by combining search requests 1 and 2.

Searches may be restricted to a single paragraph by adding a period and the paragraph label. For example, BIRD.AU will search only for documents which have the word BIRD in the author paragraph.

The ability of BRS/SEARCH to search the entire MARC record, makes computer searching much more powerful than the card catalog. A good example of this is the concept of "world view". "World view" is a fairly new topic among Christian scholars dealing with the way the Christian views the world as compared with people of other beliefs. When one of our religion professors looked in the card catalog, he found no books on world view. The term is too new to be used in the subject index and the book could only be located in the title index, if the title actually began with the words "world view". Using the search term

WORLD ADJ VIEW

quickly brought up 15 different books in which the words "world view" were mentioned either in the middle of a title or in the notes about the book. The professor was delighted to find them.

Printing

Any word that is typed is considered as a search request. Special commands are given by preceding the command with two periods. The most common of these commands is ..PRINT which switches BRS/SEARCH from search mode to print mode. The system then asks, "which paragraphs and which documents?". Most people press RETURN twice indicating all paragraphs and all documents.

Printing to a paper printer in the library or in the learning resource center is also possible. The program remains in print mode until the command ..SEARCH is used to return to the search mode.

Changing Databases

The command ..CHANGE will switch from the card catalog to any other database and back again. We've found this option very useful, because it didn't take long for us to find other uses for the program.

Asking for Help

The ..WHAT command provides access to quick help. General categories for the WHAT command are: paragraphs, databases, commands, operators, terminals, and ikeys. Typing ..WHAT along with the category, will result in detailed information about that topic. The entire user's guide is also available for searching by typing "..CHANGE MMUC".

Updating Records

The ..MODIFY command may be used by anyone with the proper privilege to update a record at any time. Records modified in this way are placed on a waiting queue and will be added between 1:00 am and 6:00 am the following morning. The actual times that the program works on the queue can be set by the system manager.

Our biggest fear of the BRS/SEARCH system was the time and resources that might be used while the databases are

being updated. We could see that the searching was very quick and efficient, but we knew that all those special inverted index files that made searching so quick, would probably make updating slow. It was very nice to know that BRS had already resolved the situation by using the night queue. Librarians are able to make their changes to the database during the day, when they like to, but the real resource consuming work is done during the night when it won't hurt anybody else.

New records from new OCLC tapes can be added using the BRSLOAD program. Our experience on our VAX-11/750 has shown that adding 1000 records to the existing 71000 records took 4-1/2 hours of connect time. Adding 30 records took 1-1/4 hours of connect time.

The commands `..DELETE`, and `..ADD` may also be used to delete or add records. These commands are carried out through the same waiting queue that the `..MODIFY` command uses.

TRAINING STUDENTS AND FACULTY

Northwestern cooperated as a beta test site for the conversion of BRS/SEARCH from unix to VMS. Because of the beta arrangement, we decided not to publicize the program or to give training classes until the beta test was over. The dedicated terminals, however, were right next to the card catalog in the library. After a week, we noticed that many people were using it as is. We then added a two page handout and laid it beside the terminals. After two more weeks, we observed that more than half of the students and faculty were choosing the BRS/SEARCH terminals over the card catalog with nothing more than a two page writeup.

We now use the BRS/SEARCH program as the first introduction to a computer for beginning students and faculty. Formerly, we used word processing as the first introduction, but searching the card catalog, requires less instruction and less keyboard input and results in a wealth of information in return. The only obstacle we have to overcome is to

explain that the program operates in two modes: a search mode and a print mode and that it responds differently and obeys different commands in each mode. One can switch from search mode to print mode by typing `..PRINT` and back to search mode again by typing `..SEARCH`.

SIDE BENEFITS AND GOALS FOR THE FUTURE

A Campus Wide Information System

Since BRS/SEARCH is a text management information system, it can be easily used for other databases. It is especially good for relatively stable text-oriented databases where finding and retrieving information on individual records is the primary function. A good example of this is a database containing the minutes of all the various committees on campus. BRS/SEARCH does not work particularly well with databases in which the individual records must be grouped together for a summary report, or in which the individual records are changing often. An expendable supplies inventory is a good example of this type of database.

Data bases which we have either developed or are planning to develop are:

- Choral and instrumental sheet music
- Theatre costume inventory
- Art slide collection
- Government documents
- Vertical files
- The Bible

Each of these databases contain 4,000 - 6,000 items with 200 - 400 characters per item. The King James Version of the Bible will be a full text data base containing the entire 5 megabytes of text.

If you plan on developing other applications for BRS/SEARCH, check your license agreement. Some licenses are for one database only.

The Information Gateway Concept

Many campuses, especially those with a college of medicine or a college of law,

may find it worth while to purchase databases which can be searched locally. In this way, the campus computer system can function the same as the library by becoming a gateway to vast amounts of information. The full text capability, makes it possible to not only find the proper journal article, but to also retrieve the entire text of the article.

Adding Circulation and Other Functions

Several developers of library software are beginning to take note of the good searching capability of BRS/SEARCH and are adding it as a module of their systems. The SIRSI corporation has incorporated BRS/SEARCH as a part of their full library system under the Unix and Xenix operating systems. OCLC has added BRS/SEARCH to their cataloging service, allowing the searching of the most recent one million entries in the MARC record database. Northwestern University has announced plans of adding it to the NOTIS system. Nothing is available or announced yet for the VMS operating system.

CONCLUSION

In conclusion, we would certainly agree that bringing a text management database system to the campus was as exciting as it was when we started to realize the power of word processing. The BRS/SEARCH program and the online card catalog has impacted the entire campus. The library is used by virtually everyone on campus, and 75 to 80 percent of those who come into the library choose the online search system over the card catalog. We receive many compliments about the increased help it gives to locate information in the library. Within a few more months, we will actually make the break and remove the old card catalog from the library. The final step to a successful and beneficial library automation project.

APPENDIX

The following listing shows each paragraph in the card catalog database,

with the name, description, and the tag fields from the MARC record which were used to create the paragraph.

- LO The location field indicates the physical location of the item. MARC tag field 049 is used for the building, holding or collection code. The holding code is followed by the call number. Our own call number is used if available (tag field 099). If it's not available, tag field 090 is used. If tag field 090 is also missing, the LC call number (tag field 050) is used. If all three fields are missing, the letters "MISSING" are placed in the location field.
- AU The author field combines the main entry items from the MARC record tag numbers 100, 110, and 111. This field may contain the name of the author, the company or the meeting/conference which produced the work.
- TI The Title and edition statement are included here. Tag field 245 plus field 250.
- IM The imprint is derived from MARC tag field 260. It gives information about the name, place and date of publication.
- PD The publication date comes from the date portion of MARC tag field 008. It contains the year of publication as a four digit number.
- CO The collation information combines MARC tag fields 300 through 399. It gives the physical description for the item, including number of pages, duration, price, etc.
- SE The Series entry combines MARC tag fields 400 through 499.
- LN The Local note comes from MARC tag field 590.
- SU The subject paragraph combines all MARC tags from 600 through 695. It contains the standard library subject classifications.

- AE The added entry field combines MARC items 700-715.
- AT The alternate title paragraph combines MARC tag fields 130, 240, 241, 242, 246, 247, 730, 740, 830, and 840.
- NT The notes paragraph combines MARC tag fields 500 through 589 and 591 through 599.
- CP The country of publication comes from a portion of MARC tag field 008. It contains a 2 or 3 letter code.
- LANG The language code also comes from a portion of MARC tag field 008. It contains a 2 or 3 letter code.
- OCLC The OCLC number comes from tag field 001 and is the main accession key for records that are cataloged with OCLC. The OCLC number is an eight digit number.
- ISBN The International Standard Book number comes from tag field 020.
- LC The library of Congress call number comes from tag field 050.
- DDC The Dewey Decimal Code comes from tag field 082.
- LCCN The library of congress card number comes from tag field 010.
- ISSN The International Standard Serial number comes from tag field 022.
- PBNO The publishers number for music comes from field 028.
- CODN The CODEN designation comes from tag field 030.
- GPO The Government Printing Office Number is in tag field 074.
- REPT The Government Report Classification number comes from tag field 086.
- LPI The local processing information combines tag fields 910, and 949.
- SRT The sorting field is derived from the location field, but the characters in the sorting field are arranged so that the information can be sorted by computer into the same order as the order of the books on the shelf. Although it doesn't always get the books exactly the same as the shelf order, it comes very close. The sort field assumes that the call number is based on the library of congress format.

DAL Magic:
Some Surprising Features of DAL

Dr. Pete Boysen
Iowa State University
233 Computer Science
Ames, Iowa 50011

ABSTRACT

The Digital Authoring Language (DAL) was designed to simplify the task of building sophisticated computer-based lessons. In addition to its answer-judging and graphics capabilities, DAL also has some lesser known features which simplify the lesson designer's task.

INTRODUCTION

The use of the Digital Authoring Language (DAL) greatly enhances the computer-based instructional capabilities of the VAX/VMS system. Answer-judging, high-resolution graphics and scoring are just some of the features which enable the lesson designer to build effective instructional lessons using this structured language. However, DAL has some lesser known features which can also assist in the programming of lessons.

If you are a Pascal, FORTRAN or BASIC programmer, you should feel at home with most of the commands available in DAL. For example, IF, LOOP, FOR and subroutines (units) are all available to control execution. Other commands can set colors, display graphics, load fonts and access files, to name a few. You may not be as familiar with the answer-judging commands, but once you understand the answer-judging process, using commands like QUERY, RIGHT, SPECS and JUDGE are just as straight-forward as using the control structures.

Once you have mastered all of these commands, you might be content to program without further investigation of DAL features. But there are other features which are well worth investigating because they can simplify your programming task and enhance the capabilities of your lessons.

These features are described in the next few sections. Each feature is described in terms of an instructional task which needs to be solved. The DAL code for each task is shown in the Appendix. (In some cases, the code was simplified for clarity and may not contain all the code necessary to insure a good instructional lesson.) I hope these examples will broaden your view of what is possible in DAL.

TABLES

In a recent article (1), the advantages of "associative arrays" in the language AWK were described (you may wish to read this article for additional applications of associative arrays.) This same capability is available in DAL through the use of tables. Tables are essentially arrays in which the index to the array is a string rather than an integer. A table is unbounded and is defined with the syntax

```
DEFINE tab[] : INTEGER
```

The elements may be INTEGER, REAL, BOOLEAN or STRING. To access an element you use the syntax

```
ASSIGN x := tab["red"]
```

where the index is a string expression.

One common instructional task is to ask the student to name several examples from a larger collection of items. In the code in Figure 1, for example, the student is asked to name three of the four western states, making sure that the student doesn't get credit by naming the same state twice.

Three units are called which evaluate the student answer in different ways. In the first unit, the "brute-force" method of maintaining an array and repeating code is used: effective but inelegant. This is probably the first solution which comes to mind to most programmers.

The second solution uses tables. The table acts like an array, noting choices which have already been given. The advantage is that you don't have to repeat any code and the code is easily expandable to additional choices. Note that the BOOLEAN table entries are initially FALSE. The disadvantage of this scheme is that you cannot allow misspellings since any misspellings would have different entries in the table from the correct spellings.

The third solution uses synonyms. This solution will be discussed in the next section.

The second instructional task in which tables are useful is in the development of a grammar analyzer. Our goal here is not to build a perfect analyzer, but rather to build one which can identify possible problem areas for the student. The section of code shown in Figure 2 attempts to determine the voice of the sentence, identify some common misspellings and count adverbs and "wordy" words. The tense of each sentence starts in active voice and progresses through various states depending on subsequent words like "to be" words. The tables are used either as lists of target words, which we know most instances of, or as lists of exception words. For examples, a word ending in "ward" is likely to be an adverb unless it is one of several exceptions like "forward." Thus, each word in the composition is passed through these table "filters" to help identify various characteristics of the composition. While these methods

don't always identify all the adverbs or the correct voice of a sentence, they are sufficient to help students improve their compositions.

SYNONYMS

One of the judging features of DAL is the specification of synonyms for words in an answer. For example,

```
SYN + "noun", "boy", "girl"  
SYN + "verb", "runs", "walks"
```

indicates that "boy" and "girl" are to be added to the list of synonyms for "noun" and "run" and "walk" are to be added to the list of synonyms for "verb". Then the statement

```
RIGHT The <<noun>> <<verb>>
```

will match the four possible simple sentences "the boy walks", "the girl walks", "the boy runs" and "the girl runs". A minus sign removes the synonym from the list.

The synonym facility provides considerable flexibility to the judgment of student answers. But if you extend the concept of synonyms beyond the restricted meaning shown above, synonyms can be quite useful in other contexts.

The problem of state names previously mentioned is one example. If you think of "right__ans" as having the state names as synonyms, the command

```
RIGHT <<right__ans>>
```

will match any of the names the first time. If you then remove the state name just entered and add it as a synonym for "wrong__ans", you can easily detect duplicate names. Again, this method is easily extended but suffers from the spelling drawback described for the TABLE example. It will accept misspelled but correct states but there will be no way to remove a misspelled state name from "right__ans". But if you want to enforce spelling the code couldn't be much simpler.

PATTERNS

Many lessons require that a student type in a command consisting of words and numbers like "move x[3] to x[4]". While the answer-judging of DAL lets you judge such commands properly, it doesn't help you extract the numbers from the command for subsequent processing. What is needed is a string function which will extract the next number in a string and return it, removing it from the string so that a subsequent call to the function can retrieve the next number.

The function in Figure 3 does this. It uses a DAL feature called a pattern. A pattern is a string of alternative strings which are separated by CTRL/Ps. A pattern is created by using the ALT function:

```
assign digit := ALT("0", "1", "2", "3", "4",  
                  "5", "6", "7", "8", "9")
```

The functions INSTRING and DELETE can accept patterns as parameters. Thus, INSTRING (s,digit) would find the location of the first digit in the

string s. Note that it uses a breadth-first search, first looking for a "0" in the whole string, and if a "0" is not found, it looks for a "1" etc. That is why SUBSTR (str, firstloc,I) is used. It looks for a sign, digit or period at the firstloc location before the search moves to the firstloc+1 location. This application is reminiscent of the ['+', '-', ...] notation in Pascal, albeit less efficient. But patterns may consist of strings and can thus be extended to look for patterns like ALT("AND", "OR", "NOT"), for example.

MATRICES

Matrix operations are available in subroutine packages on most minicomputer systems. In DAL these matrix operations are built into the language. Thus statements like

```
ASSIGN X := A*B  
ASSIGN X := INV(Y)
```

perform matrix multiplication and inversion, respectively, assuming that the arrays are properly dimensioned. In Figure 4, I have included two units which perform three-dimensional rotation, translation, scaling and perspective of an array of points. Using standard computer graphics techniques, these units show how matrices can greatly simplify a complex application and how easily they are used in DAL.

BACKUP

A unit which we frequently use displays a "Press RETURN to continue" message on the bottom of the screen and then pauses until the student presses RETURN. This is not adequate, however, when we are displaying a series of screens in a help sequence because the student often would like to quit in the middle of the sequence and return to a menu.

Figure 5 shows a unit called PFRETURN which solves this task. A unit name is passed as a parameter to PFRETURN. If the student presses PF4, the new BACKUP command is used to return gracefully to the unit which called the help sequence. If the student presses RETURN, the lesson continues.

A second use of the BACKUP command is shown in Figure 6. This code is taken from a Pascal compiler written in DAL which accepts a subset of the Pascal language and produces pseudo-code which runs on an interpreter also built in DAL. In this case, unit Workbench calls compile which calls other units. The nesting of these units can be several levels deep and, prior to the BACKUP command, the code to handle syntax errors was much more complex. Now only one error unit needs to be called if an error is detected.

MACROGRAPHS, SET AND SAVE

For English applications, we often want a student to enter an entire sentence as an answer. If a mistake is made, the student normally must retype the whole sentence. This process is error-prone and time-consuming for the student and detracts from the instruction. Furthermore, the students often type outside of the space on the screen you have provided, destroying the information already on the screen.

In Figure 7 is a unit called GetString which collects the student response. It is essentially a screen editor for one line of text. It displays a cursor and lets the student type in text, use the left and right arrow keys, move to the beginning and end of the line, and delete the next word, the remaining text or all of the text. If a non-empty string is passed, the student may edit that string. It also limits the typed text to the width of the line. This code demonstrates the use of a macro-graph to display the cursor, the SET commands needed to pick up individual characters, and the SAVE and RESTORE commands which help reset the terminal status to conditions which occurred prior to the call to GetString. If you would like to use it as part of a QUERY, try the code

```
; force judging after a second
ASSIGN qelapsed := 1
PROMPT "" $$ don't show prompt
QUERY 205
DO GetString(s,205,30,col)
ASSIGN response := s
RIGHT .....
ENDQ
ASSIGN qelapsed := 0
```

You may wish to use JUDGE AGAIN if you want the student to reenter the response to avoid the one second delay at the QUERY.

CONCLUSION

The Digital Authoring Language has a variety of features which assist the lesson author in developing sophisticated computer-based instruction. An understanding of these features can greatly enhance the lesson development process.

REFERENCES

1. Bentley, Jon. "Associative Arrays," Communications of the ACM, Vol. 28, No. 6, June 1985.

APPENDIX

```

;*****
; Assignment: Create a unit which will ask for three of the far western
; states. Make sure that you identify duplicate names.
;*****
lesson      states
define      cnt      : INTEGER
do  brute_force
do  use_tables
do  use_syn

unit  brute_force
; maintain lists of states which have already been given
define      given[4] : BOOLEAN
erase
at  205
write Name three far west states
assign      cnt := 0
query *
erase 305;480
at 305
assign response := lower(response)
right washington
.  if      given[1]
.  .      write You already gave that!
.  .      judge ignore
.  else
.  .      assign given[1] := TRUE
.  .      assign cnt := cnt + 1
.  .      if      cnt < 3
.  .      .      judge  ignore
.  .      endif
.  endif
right california
; same code as washington but use given[2]
right oregon
; same code as washington but use given[3]
right nevada
; same code as washington but use given[4]
wrong
.  write Please type a state.
endq

unit  use_tables
; use a table to keep track of given states
define      given[] : BOOLEAN
erase
at  205
write Name three far west states
assign      cnt := 0
specs exact  $$ need this for table
query *
erase 305;480
at 305
assign response := lower(response)
right Washington | Oregon | Nevada | California
.  if      given[response]
.  .      write You already gave that!
.  .      judge ignore
.  else

```

FIGURE 1: Using Tables with States

```

.      .      assign given[response] := TRUE
.      .      assign cnt := cnt + 1
.      .      if      cnt < 3
.      .      .      judge ignore
.      .      endif
.      endif
wrong
.      write  Type a state name.
endq
specs noexact

unit use_syn
; maintain synonym lists of no yet given and already given answers
syn  +"right_ans","washington","oregon","california","nevada"
erase
at 205
write Name three far west states
assign      cnt := 0
query *
erase 305;480
at 305
assign response := lower(response)
right <<right_ans>>
. write Good!
. assign cnt := cnt + 1
. syn ~"right_ans",response
. syn +"wrong_ans",response
. if cnt < 3
. . judge ignore
. endif
wrong <<wrong_ans>>
. write you already gave that!
wrong
. write Type a state!
endq
endlesson

```

FIGURE 1 (Cont.): Using Tables with States

```

;*****
; Determine voice of sentence, possible adverbs, wordy words and common
; misspellings.
;*****
unit getstyle
define      temp,wrld      : STRING
           i                : INTEGER

assign      misspellcnt := 0
assign      numberbegin := FALSE
for i:=1,nsent
.   assign  voice[i] := active
.   assign  nadverbs[i] := 0
.   assign  nwordy[i] := 0
.   assign  nwords[i] := 0
.   assign  temp := lower(sent[i])
.   assign  wrd := word(1,temp)
.   if      isnumber(wrd)
.   .       assign  numberbegin := TRUE
.   endif
.   loop    wrd <> ""
.   .       if      (right(wrd,4)="ward") AND (not wardword[wrd])
.   . .       assign  exadvword := wrd
.   . .       assign  exadvsent := i
.   . .       assign  nadverbs[i] := nadverbs[i] + 1
.   . .       branch $nextwrd
.   .       endif
.   .       if      (right(wrd,2)="ly") AND (not lyword[wrd])
.   . .       assign  exadvword := wrd
.   . .       assign  exadvsent := i
.   . .       assign  nadverbs[i] := nadverbs[i] + 1
.   . .       branch $nextwrd
.   .       endif
.   .       if      tobeword[wrd] AND (voice[i] = active)
.   . .       assign  exbword := wrd
.   . .       assign  voice[i] := tobe
.   . .       branch $nextwrd
.   .       endif
.   .       if      voice[i] = tobe
.   . .       if      (right(wrd,2)="ed") AND (not edword[wrd])
.   . . .       assign  voice[i] := passive
.   . . .       branch $nextwrd
.   . .       endif
.   . .       if      (right(wrd,3)="ing") AND (not ingword[wrd])
.   . . .       assign  voice[i] := progressive
.   . . .       branch $nextwrd
.   . .       endif
.   . .       if      strongword[wrd]
.   . . .       assign  voice[i] := passive
.   . . .       branch $nextwrd
.   . .       endif
.   .       endif
.   .       if      wordyword[wrd]
.   . .       assign  exwordy := wrd
.   . .       assign  nwordy[i] := nwordy[i] + 1
.   . .       branch $nextwrd
.   .       endif
.   .       if      misspell[wrd]
.   . .       assign  misspellcnt := misspellcnt + 1
.   .       endif

```

FIGURE 2: Using Tables for Sentence Analysis

```

.      .      $nextwrd
.      .      assign  nwords[i] := nwords[i] + 1
.      .      assign  wrd := word(nwords[i],temp)
.      endloop
.      test   voice[i]
.      value  tobe
.      .      assign  extobesent := i
.      value  passive
.      .      assign  expassive := i
.      endtest
.      if     nwordy[i] > 1
.      .      assign  exwr dysent := i
.      endif
.      assign  nwords[i] := nwords[i] - 1
endfor

```

FIGURE 2 (Cont.): Using Tables for Sentence Analysis


```

;*****
; GetNum : Get the next number in the string STR and return it. Remove
;         it from str also so that subsequent calls can remove remaining
;         numbers. Numbers consist of an optional sign, digits and decimal
;         point. Return a null string if no number is found.
;*****
funct getnum( str ):STRING
define      str          : STRING  $$ the string which may contain a number
            numstr       : STRING  $$ the string representation of the number
            digit        : STRING  $$ pattern of digits
            firstloc     : INTEGER  $$ location of first char of number
            endloc       : INTEGER  $$ location of end char
            foundperiod  : BOOLEAN  $$ TRUE if a period was in number

; digit will match any string which is a digit
assign     digit := alt("0","1","2","3","4","5","6","7","8","9")
assign     firstloc := 1
assign     numstr := ""
; look for beginning of number a character at a time. A number may begin
; with a sign, digit or decimal point
loop instring(substr(str,firstloc,1),alt(digit,"-","1",".")) = 0
.   assign firstloc := firstloc + 1
.   if     firstloc > len(str)
.       assign getnum := ""      $$ no number found
.       return
.   endif
endloop
assign     endloc := firstloc + 1
; allow for spaces after sign
if instring(substr(str,firstloc,1),alt("-","+")) > 0
.   loop  ascii(str,endloc) = 32 $$ loop while there are spaces
.       assign endloc := endloc + 1
.       if     endloc > len(str)
.           assign getnum := ""
.           return $$ sign only so ignore
.       endif
.   endloop
endif
assign     foundperiod := FALSE
; keep looping as long as digits or one decimal point found.
; quit early if second decimal point found.
loop instring(substr(str,endloc,1),alt(digit",".")) > 0
.   if     substr(str,endloc,1) = "."
.       outloop foundperiod
.       assign foundperiod := TRUE
.       endif
.       assign endloc := endloc + 1
outloop   endloc > len(str)
endloop
; now pull it out of the string. It is between firstloc and endloc.
assign     numstr := substr(str,firstloc,endloc-firstloc)
if isnumber(numstr)      $$ double check that it is a number
.   assign str := delete(str,numstr)
.   assign getnum := numstr
else
.   assign getnum := ""
endif

```

FIGURE 3: Using Patterns for Lexical Analysis

```

;*****
; Transform: Perform rotations, translation and scaling on the array
;            of points PT. Each row of PT is of the form:
;            (X,Y,Z,1) where X,Y and Z are real-world coordinates
;            The parm array is of the form:
;            (Rx,Ry,Rz) representing Rotations in degrees
;            (Tx,Ty,Tz) representing Translations in linear units
;            (Sx,Sy,Sz) representing Scaling
;*****
UNIT Transform(pt,parm)
DEFINE      pt[PT] : REAL $$ Nx4 array of points
           parm[3] : REAL $$ 3x3 array of rotations,translation and scaling
           c[3],s[3] : REAL$$ temporary storage for sines and cosines
           i : INTEGER
           xform[4,4] : REAL
IF          (DIMAX(pt,2) <> 4) OR (DIMAX(parm,1) <> 3) OR (DIMAX(parm,2) <> 3)
.          ERASE 2200;2464
.          SIZE 1
.          AT 2200
.          WRITE The point matrix must be Nx4 where N is the number of points
.              The parm array must be 3x3
.          PAUSE
.          RETURN
ENDIF
; temporarily store cosines and sines to save computation
FOR i:=1,3
.  ASSIGN c[i] := COS(RAD(parm[1,i]))
.  ASSIGN s[i] := SIN(RAD(parm[1,i]))
ENDFOR
; now create xform MATRIX
ASSIGN xform[1,1] := parm[3,1]*(c[1]*c[2])
ASSIGN xform[1,2] := parm[3,2]*(s[1]*c[2])
ASSIGN xform[1,3] := parm[3,3]*(-s[2])
ASSIGN xform[1,4] := 0
ASSIGN xform[2,1] := parm[3,1]*(-s[1]*c[3]+c[1]*s[2]*s[3])
ASSIGN xform[2,2] := parm[3,2]*(c[1]*c[3]+s[1]*s[2]*s[3])
ASSIGN xform[2,3] := parm[3,3]*(c[2]*s[3])
ASSIGN xform[2,4] := 0
ASSIGN xform[3,1] := parm[3,1]*(s[1]*s[3]+c[1]*s[2]*c[3])
ASSIGN xform[3,2] := parm[3,2]*(-c[1]*s[3]+s[1]*s[2]*c[3])
ASSIGN xform[3,3] := parm[3,3]*(c[2]*c[3])
ASSIGN xform[3,4] := 0
ASSIGN xform[4,1] := parm[2,1]
ASSIGN xform[4,2] := parm[2,2]
ASSIGN xform[4,3] := parm[2,3]
ASSIGN xform[4,4] := 1
; now do all the hard work!
ASSIGN pt := pt*xform

```

FIGURE 4: Using Matrix Operations

```

;*****
; Perspective:      Convert the points in array PT to two-dimensional values
;                  so that the image appears to be viewed from DISTANCE away
;                  from the eye.
;*****
UNIT Perspective(pt,distance)
DEFINE      pt[?]  : REAL $$ array of points (X,Y,Z,1)
            distance : REAL $$ length between eye and center of object
            pers[4,4] : REAL$$ array used to produce perspective constants
            i,j      : INTEGER

IF      distance <= 0
.      ERASE      2300;2480
.      AT        2300
.      WRITE      The distance from the object must be positive.
.      PAUSE
.      RETURN
ENDIF

; to transform the normalized array PT into perspective coordinates, use
; the pers array of the following form:
; (1,0,0,0)
; (0,1,0,0)
; (0,0,0,-1/D) where D is distance from center of object
; (0,0,0,1)
ASSIGN      pers := IDEN(4)
ASSIGN      pers[3,3] := 0
ASSIGN      pers[3,4] := -1.0/distance
; do the hard work again!
ASSIGN      pt := pt*pers
; the 4th column now contains the adjustment factor for that row.  Convert
; the X and Y coordinates appropriately.
FOR      i:=1,DIMAX(pt,1)
.      IF      pt[i,4] <> 0
.      .      ASSIGN      pt[i,1] := pt[i,1]/pt[i,4]
.      .      ASSIGN      pt[i,2] := pt[i,2]/pt[i,4]
.      ENDIF
ENDFOR
ENDLESSON

```

FIGURE 4 (Cont.): Using Matrix Operations

```

LESSON      test
DEFINE      pt[26,4] : REAL
            xf[3,3] : REAL
            npt[26,4],xpt[26,4] : REAL

DO          init
DO          try

UNIT        init
DEFINE      rec : STRING
            i : INTEGER
OPEN        "edge.dat",1,READ
GET         1,REC
ASSIGN      i := 1
LOOP        NOT EOF(1)
.           ASSIGN pt[i,1] := NUMBER(WORD(1,REC))
.           ASSIGN pt[i,2] := NUMBER(WORD(2,REC))
.           ASSIGN pt[i,3] := NUMBER(WORD(3,REC))
.           ASSIGN pt[i,4] := 1
.           ASSIGN pt[i+1,1] := NUMBER(WORD(4,REC))
.           ASSIGN pt[i+1,2] := NUMBER(WORD(5,REC))
.           ASSIGN pt[i+1,3] := NUMBER(WORD(6,REC))
.           ASSIGN pt[i+1,4] := 1
.           ASSIGN i := i + 2
.           GET         1,REC
ENDLOOP
ASSIGN      xf[2,1] := -25
ASSIGN      xf[2,2] := -25
ASSIGN      xf[2,3] := -25
ASSIGN      xf[3,1] := 1.0
ASSIGN      xf[3,2] := 1.0
ASSIGN      xf[3,3] := 1.0
do          transform(pt,xf)
ASSIGN      xf[2,1] := 0
ASSIGN      xf[2,2] := 0
ASSIGN      xf[2,3] := 0
ASSIGN      xf[3,1] := 1.0
ASSIGN      xf[3,2] := 1.0
ASSIGN      xf[3,3] := 1.0
CLOSE 1

UNIT        try
DEFINE      i,j      : INTEGER
            p        : REAL
            axis      : STRING
ERASE
at          205
write Rotate about X,Y or Z?
query *
right x
.           assign j:=3
right y
.           assign j:=2
right z
.           assign j:=1
endq
assign      axis := UPPER(response)
at          305
write Perspective (1-1000)?
input *

```

FIGURE 4 (Cont.): Using Matrix Operations

```

assign      p := NUMBER(response)
assign     xpt := pt
for      i:=0,360,10
.   ASSIGN xffl,1l := 0
.   ASSIGN xffl,2l := 0
.   ASSIGN xffl,3l := 0
.   ASSIGN xffl,jl := 10
.   DO      Transform(xpt,xf)
.   ASSIGN npt := xpt
.   DO      Perspective(npt,p)
.   erase
.   at      250
.   write  <<s,axis>> at <<s,i>> degrees
.   do      display
.   pause
.   mode   replace
endfor
pause
redo

UNIT display
DEFINE    i      : INTEGER
CORIGIN  380,240
FOR      i:=1,DIMAX(npt,1),2
.   GLINE  INT(nptf i,1l),INT(nptf i,2l),INT(nptf i+1,1l),INT(nptf i+1,2l)
ENDFOR

ENDLESSON

```

FIGURE 4 (Cont.): Using Matrix Operations

```

;*****
; PFReturn:  Display "RETURN Continue PF4 Exit" at the bottom of the screen
;            and then pause. Continue if RETURN is pressed, otherwise
;            BACKUP to u_name if PF4 is pressed.
;*****
unit  pfreturn(u_name)
define  c_stat,q_stat,k_stat,t_stat      : INTEGER
        u_name                          : STRING
        continue                        : BOOLEAN
; save current status and restore at end of unit
save  10,prompt,where,size,italics
prompt  ""
assign  t_stat := termstat(t_typeahead)
assign  k_stat := termstat(t_fkey)
assign  c_stat := fcolor
assign  q_stat := qlength
; discard any characters pressed prior to the pause
set  typeahead,off
set  fkey,terminate
assign  qlength := 1
charset  "standard"
mode  fixed
size  1,2
italics  0
at  254,460
do  keychar("RETURN")
write  Continue
do  keychar("PF4")
write  Exit
; loop until CR or PF4 is pressed
assign  continue := TRUE
set  echo,off
loop  continue
.  input  0
.  test  response
.  value  "", "[PF4_KEY]"
.  .  assign  continue := FALSE
.  endtest
endloop
set  echo,on
erase  2325;2460
set  typeahead,t_stat
set  fkey,k_stat
assign  qlength := q_stat
restore  10
at  where
if  response = "[PF4_KEY]"
.  if  upper(u_name)="TOP"
.  .  backup  TOP
.  else
.  .  backup  u_name
.  endif
endif
endlesson

```

FIGURE 5: Using BACKUP for Return from Help

```

;*****
; present options to Edit, Compile, Delete etc.
;*****
unit workbench
test col
value 1
.   fcolor  green
.   erase   200;1959
.   do      editor(source,200,18,80,linecnt,colcnt,TRUE)
.   do      showmessage("")
.   assign  col := 8
.   assign  in := ""
value 8
.   do      compile
.   if      in = "" $$ there was an error
.   .       assign  col := 1
.   else
.   .       assign  linecnt := 1
.   .       assign  colcnt := 1
.   .       do      showmessage("Compilation is complete!")
.   .       assign  col := 18
.   endif
;...other code
;*****
; compile the Program in Source and return the P Code in IN
;*****
unit compile
define      filelist      : STRING
           progname       : STRING
           info,str,id     : STRING
           i,j             : INTEGER

do  CompileInit
do  gettoken
if  token <> "PROGRAM"
.   do  comperror("'PROGRAM' expected.")
endif
do  gettoken
assign  progname := token
do  gettoken
if  token = "("
.   do  gettoken
.   do  GetIdList(filelist)
.   if  token <> ")"
.   .   do  comperror("'')' expected.")
.   endif
.   assign  info := "PROC,N," + filelist + ","
.   .   do  AddSymbol(progname,info)
.   .   do  AddSymbol(filelist,"FILE")
.   .   do  gettoken
endif
if  token <> ";"
.   do  comperror("';' expected.")
endif
do  gettoken
do  procbody(progname)
if  token <> "."
.   do  comperror("'.' expected.")
endif
;*****

```

FIGURE 6: Using BACKUP in a Pascal Compiler

```

; report error and return to workbench
;*****
unit comperror(msg)
define      msg      : STRING
open  HISTFILE,2,update
put  2,"Compile:"+msg
close 2
; now compute column it was on
assign      colcnt := 0
loop  ASCII(in,inptr) (> 13
.      assign colcnt := colcnt + 1
.      assign inptr := inptr - 1
endloop
assign      in := ""
assign      errfound := TRUE
do  showmessage("Compile Error: "+msg)
backup      "WORKBENCH"

;*****
; get a list of identifiers which are separated by commas
;*****
unit GetIdList(list)
define      list      : STRING
if  not IsID(token)
.  do      comperror("Identifier expected.")
endif
assign      list := token
do  gettoken
loop  token = ","
.  do      gettoken
.  if      not IsID(token)
.  .      do      comperror("Identifier expected.")
.  endif
.  assign  list := list + "," + token
.  do      gettoken
endloop

```

FIGURE 6 (Cont.): Using BACKUP in a Pascal Compiler


```

;*****
; GetString : Get a string starting at LOC position (Cross grid) on the
;             screen. Allow up to width characters to be typed. The
;             initial and final value of the string will be in LINE and
;             the initial and final location of the cursor will be in
;             COL. The cursor keys and DELETE key work as normal the
;             additional functions provided by pressing SELECT or
;             PF1 are provided:
;             S goes to Start of Line
;             E goes to End of Line
;             W deletes next Word
;             L deletes rest of Line
;             X deletes whole line
;*****
UNIT GetString(line,loc,width,col)
DEFINE      line      : STRING
           loc        : INTEGER
           width      : INTEGER
           col        : INTEGER
           i,j        : INTEGER
           Margin     : INTEGER
           CurRow     : INTEGER
           s          : STRING
           d_stat     : INTEGER
           k_stat     : INTEGER
           q_stat     : INTEGER
           Bell       : STRING

; save old settings
ASSIGN      d_stat := termstat(d_stat)
ASSIGN      k_stat := termstat(t_fkey)
ASSIGN      q_stat := qlength
ASSIGN      qlength := 1
SET         fkey,terminate
SET         delete,off
SAVE       10,prompt
SET         echo,off
ASSIGN      Bell := CHAR(7)
PROMPT     ""
IF         LEN(Line) > WIDTH
.          ASSIGN Line := LEFT(Line,Width)
ENDIF
ASSIGN      Margin := (Loc MOD 100)-1
ASSIGN      CurRow := 100*INT(Loc/100)
; the following macrograph will display a cyan block cursor by using a
; combination of INVERSE, ERASE and complement. The text consists of
; a space followed by a BACKSPACE to put the cursor back at the original
; location.
MGRAPH     "X"
.          REGIS "T(B)(A0)(W(NICI(C)))"(E)"
ENDMGRAPH
MODE       REPLACE
$START
; make sure cursor stays within field
IF         Col < 1
.          ASSIGN Col := 1
ENDIF
IF         Col > LEN(Line)
.          IF         LEN(Line) < Width
.                  .          ASSIGN Col := LEN(Line) + 1

```

FIGURE 7: Using Save and Restore for Student Editing of Responses

```

.     ELSE
.     .     ASSIGN Col := Width
.     .     ENDIF
ENDIF
AT CurRow+Margin+col
MPLOT "X"    $$ plot the cursor
INPUT *
MPLOT "X"    $$ erase the cursor
TEST keypressed
VALUE 32..126 $$ add the text
.     ASSIGN Line := LEFT(line,col-1)+char(keypressed)+RIGHT(line,col)
.     IF LEN(Line) > Width
.     .     ASSIGN Line := LEFT(Line,Width)
.     .     ENDIF
.     WRITE <<s,RIGHT(line,col)>>
.     ASSIGN Col := Col + 1
VALUE 127    $$ delete a character
.     IF Col > 1
.     .     IF (Col = LEN(line)) AND (Col = Width)
.     .     .     ASSIGN Line := LEFT(line,col-1)
.     .     .     ELSE
.     .     .     .     ASSIGN Line := LEFT(line,col-2)+RIGHT(line,col)
.     .     .     .     ASSIGN col := col-1
.     .     .     .     ENDIF
.     .     .     AT CurRow+Margin+col
.     .     .     ASSIGN s := RIGHT(line,col) + " "
.     .     .     WRITE <<s,s>>
.     .     ELSE
.     .     .     IF Col = Width
.     .     .     .     ASSIGN Line := ""
.     .     .     .     AT CurRow+Margin+Col
.     .     .     .     WRITE
.     .     .     .     ENDIF
.     .     .     ENDIF
.     .     ENDIF
VALUE 256,314 $$ PFI or SELECT for options
.     input *
.     ASSIGN LastKey := 256
.     IF (keypressed<32) OR (keypressed>126)
.     .     CONTROL Bell
.     .     ENDIF
.     TEST UPPER(char(keypressed))
.     VALUE "X"    $$ (delete whole line)
.     .     AT CurRow+Margin+1
.     .     WRITE <<s,BlankPad(" ",LEN(LINE))>>
.     .     .     ASSIGN Col := 1
.     .     .     ASSIGN Line := ""
.     .     VALUE "W"    $$ (delete to end of word)
.     .     .     ASSIGN i := COL
.     .     .     LOOP (i < LEN(Line)) AND (ASCII(Line,i) <> 32)
.     .     .     .     ASSIGN i := i + 1
.     .     .     .     ENDOLOOP
.     .     .     LOOP (i < LEN(Line)) AND (ASCII(Line,i) = 32)
.     .     .     .     ASSIGN i := i + 1
.     .     .     .     ENDOLOOP
.     .     .     IF (i = LEN(Line)) AND (ASCII(Line,i-1) <> 32)
.     .     .     .     .     ASSIGN i := i + 1
.     .     .     .     .     ENDIF
.     .     .     WRITE <<s,BlankPad(Right(Line,i),LEN(LINE)-Col+1)>>
.     .     .     ASSIGN Line := LEFT(Line,Col-1) + RIGHT(Line,i)
.     .     VALUE "L"    $$ (delete to End of line)

```

FIGURE 7 (Cont.): Using Save and Restore for Student Editing of Responses.

```

.      .      WRITE  <<s,BlankPad(" ",LEN(LINE)-Col+1)>>
.      .      ASSIGN  Line := LEFT(Line,COL-1)
.      VALUE  "S"    $$ (go to start of line)
.      .      ASSIGN  Col := 1
.      VALUE  "E"    $$ (go to END of line)
.      .      ASSIGN  Col := LEN(Line)+1
.      .      ENDTEST
VALUE  276
.      .      ASSIGN  Col := Col-1
VALUE  277
.      .      ASSIGN  Col := Col+1
VALUE  13,256..316
.      .      BRANCH  $RET
ENDTEST
BRANCH  $START
$RET
ASSIGN  LastKey := keypressed
ASSIGN  qlength := q_stat
SET    fkey,k_stat
SET    delete,d_stat
SET    echo,on
RESTORE 10

```

FIGURE 7 (Cont.): Using Save and Restore for Student Editing of Responses

8088 MACRO ASSEMBLER ON THE RAINBOW MICRO COMPUTER

Robert S. Workman
Southern Connecticut State University
New Haven, Connecticut

ABSTRACT

Rainbow 100A micro computers were used as the laboratory computer in an 8088 Macro Assembler Programming course. The goals of the course were to introduce the architecture of the Intel 8088 micro processor and its associated coprocessors and to teach 8088 Macro Assembler Language. Emphasis was placed on the use of modular programming and on line debugging techniques.

INTRODUCTION

Almost all books and articles about the Intel 8088 Microprocessor and the 8088 Macro Assembler Language are written with the assumption that the reader is working with an IBM PC. This paper assumes that the reader has access to and is familiar with IBM PC oriented Macro Assembler Language material, but is using a Rainbow 100 with the MS-DOS operating system. Methods for organizing code to take advantages of Rainbow features and changes that must be made to accommodate Rainbow differences will be emphasized in this paper.

Most IBM PC oriented material is directly applicable to the Rainbow. The MS-DOS operating system appears to the user to be almost identical to PC-DOS that is used on the IBM. The 8088 microprocessor, its architecture and instruction set are identical for both machines. The Macro Assembler Language appears to be the same Microsoft product. So does the line editor EDLIN, the on line debugger DEBUG,

Many differences between the Rainbow and the IBM PC arise from the Rainbow's version of Microsoft's Disk Operating System use of different interrupts and interrupt function codes than the Microsoft's DOS written for the IBM. Other differences between the Rainbow and the IBM PC stem from different implementation or lack of implementation of specific hardware feature such as graphics, audio capability, and disk storage. A knowledge of a few differences between Rainbow DOS and IBM PC DOS will allow one to use most of the material in IBM PC oriented books or

articles that do not deal with IBM PC specific features.

This paper will first present a demonstration Assembler Language program, illustrating program the organization that is recommended for beginning level Macro Assembler coding. Some of the most useful Rainbow interrupt functions will then be summarized along with keyboard related differences in the use of EDLIN. Finally a Rainbow translation will be given of a recently published procedure written for the IBM PC. The procedure demonstrates how to replace the DOS keyboard I/O interrupt handler with user written code. The IBM to Rainbow translation is representative of what must be done to transport code to the Rainbow or make use of IBM PC oriented reference.

A DEMONSTRATION MACRO ASSEMBLER LANGUAGE PROGRAM WRITTEN FOR THE RAINBOW

Program 1 is an example of a first assembler language program. The program displays a message, the user's response to the message will cause one of two responses to be displayed. The program illustrates Macro Assembler Language organization. The version of the program will be a COM file. Segment registers will be set to the same value. Very simple macros are used to display messages, accept a response, and terminate the program. The macros are usually placed in a separate library file name MAC.LIB. When this is done the assembler directive, INCLUDE MAC.LIB must be put in the program in place of

```

TITLE Demonstration Macro Assembler Language Program
;=====start of macros may be placed in MAC.LIB=====
    READ_KEYBOARD_AND_ECHO MACRO ;pg1-34 DOS prog ref man
        PUSH AX
        XOR AX,AX
        MOV AH,01H ;waits for character to be typed,
        INT 21H ;displays it and returns it in AL
        POP AX
    ENDM
    DISPLAY_CHARACTER MACRO ;pg1-35 DOS prog ref man
        PUSH AX
        MOV AH,02H ;displays character in DL
        INT 21H
        POP AX
    ENDM
    DISPLAY_STRING MACRO ;PG 1-44 DOS prog ref man
        PUSH AX
        MOV AH,09H ;DX must contain offset of string,
        INT 21H ;The string must in with "$".
        POP AX
    ENDM
    PRINT_CHARACTER MACRO ;pg1-38 DOS prog ref man
        PUSH AX
        MOV AH,05H
        INT 21H
        POP AX
    ENDM
    STOP MACRO ;pg1-134 DOS prog ref man
        MOV AH,4CH
        INT 21H
    ENDM
;==End of macros, remove ";" from next line if macros are in MAC.LIB==
;INCLUDE MAC.LIB
CODE SEGMENT
ASSUME CS:CODE,DS:CODE,ES:CODE
ORG 0100H
    MAIN: JMP BEGIN
        START_MESSAGE DB 'ARE YOU HAPPY? ENTER N OR Y $'
        YES_MESSAGE DB ' THATS GREAT!$'
        NO_MESSAGE DB ' CHEER UP!$'
        ERROR_MESSAGE DB ' Y OR N PLEASE !',0AH,0DH,'$'

    BEGIN:
        PROC1 PROC NEAR
            MOV DX,OFFSET START_MESSAGE
            DISPLAY_STRING
        READ_KEY:
            READ_KEYBOARD_AND_ECHO ;macro character in AL
            CMP AL,'N'
            JE DISPLAY_NO_MESSAGE
            CMP AL,'Y'
            JE DISPLAY_YES_MESSAGE
            MOV DX,OFFSET ERROR_MESSAGE ;output a string of characters
            DISPLAY_STRING
            JMP READ_KEY
        DISPLAY_NO_MESSAGE:
            MOV DX,OFFSET NO_MESSAGE
            DISPLAY_STRING
            JMP DONE
        DISPLAY_YES_MESSAGE:
            MOV DX,OFFSET YES_MESSAGE
            DISPLAY_STRING
        DONE:
            STOP
        PROC1 ENDP
    CODE ENDS
    END MAIN

```

Program 1. An 8088 Macro Assembler Program showing macro definition and use.

all the macro code.

When more complex input or output is needed it is highly recommended that code from a source such as Bluebook of Assembly Routines for the IBM PC & XT[7] be used. SGNDEC16IN which accepts a signed decimal number from the keyboard and converts the number to internal signed two's complement 16-bit binary form is typical of the code in the reference.

```
COPY MASM.* E:
COPY LINK.* E:
COPY EDLIN.* E:
COPY %1.ASM E:
COPY EXE2BIN.* E:
COPY G.BAT E:
COPY DEBUG.* E:
COPY MAC.LIB E:
COPY PROC.LIB E:
E:
DIR
```

Figure 2. Batch file for copying programs to memory.

When writing beginning level programs it is recommended that the line editor EDLIN be used and that the programs use to assemble, link, changed to a COM file, and execute the source program be placed in a batch file. Figure 1. shows a sample batch file. As can be seen from the batch file, programs that are used in the assembly process have been place in E: drive using the Rainbow's virtual disk drive feature. On systems with at least 320K bytes of memory all needed programs including DEBUG and EDLIN will fit in memory. The use of E: drive greatly speeds up the assembly process as long as the programs being tested do not frequently crash the system. Figure 2. shows the batch file used to place the programs and the macro and procedure libraries on E: drive. Before ending an E: drive session the user should be sure to copy the new version of the source program to a non virtual diskette.

One of the most interesting features of Macro Assembler Language use is that programs can be traced with DEBUG an interactive debugger[3]. DEBUG displays the contents of all the 8088 registers and allows the programmer to step through the program one instruction at a time. At any time the contents of memory may be displayed either as data or unassembled as instructions. The experience of seeing the immediate effect of an instruction on registers and memory is an invaluable help in aiding students in understanding the operation of the microprocessor.

```
MASM %1;
LINK %1;
EXE2BIN %1 %1.COM
%1
```

Figure 1. Batch File for assembling, linking, converting, and executing a Macro Assembler Source Program

SOME RAINBOW MS-DOS INTERRUPTS AND INTERRUPT FUNCTION CODES

Figure 3 contains a summary of interrupts used in Program 1 and two other interrupts useful to beginning assembly language coders, Print Character and Read Keyboard Without Echo. To use these interrupts move the function code to AH first. See the examples in Program 1. A complete list of Rainbow MS-DOS interrupts may be found in Microsoft MS-DOS Operating System Programmer's Reference Manual[6]. A number followed by a "h" indicates that the number is written in hexadecimal, format.

Function	Use
01h	Read Keyboard and echo Character returned in AL
02h	Display Character
05h	Character displayed in DL
08h	Print Character
09h	Character to print in DL
0Ch	Read Keyboard no echo
09h	Character returned in AL
09h	Display String
4Ch	Offset in DX, end with "\$"
	Terminate a Process
	Preferred end of program

Figure 3. Some frequently used Interrupt 21h functions.

The Fall 1985 special issue of "Byte" is devoted to articles about the IBM PCs. As with most IBM PC material a great deal, but not all of the material relates to the Rainbow. A representative example of this may be found in "Writing Desk Accessories"[11]. This article shows how to modify the IBM PC keyboard interrupt handler. For this material to be useful to the Rainbow programmer a few of changes must be made to the code.

The first problem encountered in converting the program to the Rainbow is that the IBM PC cassette I/O vector is modified during testing. As cassette tape I/O is not supported on the Rainbow and the cassette I/O vector use is not essential to the program, reference to it may be removed. Next the IBM interrupts shown in Figure 3 are replaced by their Rainbow equivalents. The equates are shown in Figure 4. In all, thanks to Wadlow's well written code, only five changes must be made. One of which has to do with a hardware difference and the others with interrupt numbering differences. This type of revision is typical of that encountered when working with IBM PC oriented material.

```
DOS_keyboard_io equ 16h
DOS_replace_vector equ 15h
DOS_function equ 21h
DOS_terminate_resident equ 27h
get_vector equ 35h
set_vector equ 25h
```

Figure 3. IBM PC version of equates keyboard interrupt handler routine [11].

```
DOS_keyboard_io equ 21h ;set ah=01
DOS_replace_vector equ 21h
DOS_function equ 21h
DOS_terminate_resident equ 31h
get_vector equ 35h
set_vector equ 25h
```

Figure 4. Rainbow version of equates for keyboard interrupt handler routine.

KEYBOARD DIFFERENCES THAT EFFECT THE USE OF EDLIN

The EDLIN line editor is frequently used to edit 8088 Assembly Language programs. As it is short it loads quickly and may also be placed on E: drive. Two EDLIN instructions may cause the beginning Rainbow programmer problems. The IBM PC format for the Replace Text instruction is:

```
[line][line][?]R[string][<F6>string].
"<F6>" refers to Function Key 6 on the IBM PC. This is a control Z. On the Rainbow the Exit Key or simultaneously pressing the Ctr Key and Z do the same thing as Function Key 6 on the IBM PC.
```

A minor inconsistency exists on the Rainbow that may cause some confusion. For EDLIN users, the "escape" character is entered by using the "interrupt" key not the "escape" key.

CONCLUSION

This paper has presented some methods for organizing Macro Assembler Language programs on the Rainbow and given examples of how IBM PC oriented material can be modified for Rainbow use. It has been my experience that IBM PC oriented texts and articles can be used with little problem in courses where the Rainbow is the primary laboratory 8088 microcomputer.

REFERENCES

1. Able, Peter, Assembler for the IBM PC and the PC-XT, Reston Publishing Company, Inc., Reston, Virginia, 1984.
2. iAPX88 Book, Intel Corporation Santa Clara, Ca, 1981.
3. Microsoft Debug Utility for 8086 and 8088 Microprocessors, Microsoft Corporation.
4. Microsoft Link Linker Utility for 8086 and 8088 Microprocessors, Microsoft Corporation.
5. Microsoft MS-DOS Operating System Macro Assembler Manual, Microsoft Corporation, Bellevue, Washington, November 1984.
6. Microsoft MS-DOS Operating System Programmer's Reference Manual, Microsoft Corporation, Bellevue, Washington, 1983.
7. Morgan, Christopher L., Bluebook of Assembly Routines for the IBM PC & XT, A Plume/Waite Book New American Library, New York, 1984.

8. Norton, Peter, Inside the IBM PC, Access to Advanced Features and Programming, Robert J. Brady Co., Bowie, Maryland, 1983.
9. Rainbow MS-DOS V2.05 Programmer's Guide, Digital Equipment Corporation, November 1984.
10. Rainbow MS-DOS V2.11, Update Notes, Digital Equipment Corporation, November 1984.
11. Wadlow, Tom, "Writing Desk Accessories, Design you own memory-resident programs for the IBM PC, available at the touch of a key", Byte, (Fall 1985), 105-122.

GRAPHICS APPLICATIONS SIG

TCHART: Development of a Device Independent
Chart Drawing Program

Judith Bardell
Boeing Computer Services
P.O. Box 24346, M/S 1E-32
Seattle, Washington 98124
(206) 241-3079

ABSTRACT

New graphics terminals and hardcopy devices are being introduced every year. The current challenge is to provide application software which is compatible with many different devices. TCHART is an interactive chart drawing program based on the DI-3000 graphics package (Precision Visuals Inc.). TCHART is executed on the VAX/VMS operating system and runs on a variety of DI-3000 supported graphics devices. The advantages as well as limitations of basing interactive software on a well developed graphics package will be discussed. It will be seen that a program like TCHART acts as a doorway allowing an unsophisticated user almost complete access to the graphics package capabilities.

The Problem and Its Environment

TCHART was developed by the Boeing Aerospace Company (BAC) Support Division of Boeing Computer Services (BCS). This group is specifically tasked with graphics software support of BAC owned VAX computer centers. Graphics software is provided to more than 50 sites intended for engineering use. Site computers include VAX 730, 750, 780, 785, 8600, and Micro VAX.

As a graphics software support group, we develop and maintain engineering application programs and act as a distribution point for licensed software. The group also produces a newsletter (The VAX Graphics News) on a monthly basis to insure that engineers are kept informed about the changing availability of software/hardware products for their use.

Variety is the rule in the computing environment in which our graphics software is used. Each site has its own set of interactive terminals and hardcopy plotting devices. For this reason, the decision was made several years ago to produce device independent software wherever possible. The Precision Visuals Inc. (PVI) DI-3000 software was chosen as a basis for new applications. DI-3000 is a rich graphics package with a large number of device drivers supported. For devices not supported, PVI makes it relatively easy for the programmer to write his own driver.

The problem addressed by TCHART was to create a chart drawing program which was easy and natural to use interactively and which could eventually be linked to output produced by other products we support, such as:

EGG	Engineering Graphics Generator
PLOTMAKER	Bar, Line, Contour plots
PILGRIM	Text slide generating program

TCHART would be executed on devices ranging from TEKTRONIX 4014 to IRIS 2400. TCHART would be written in the DI-3000 graphics language. These two given conditions created constraints on the original

goal of producing the most natural and easy-to-use drawing package.

The Goal - A Good Interactive Program

The factors of a good interactive program are defined all the time by disgruntled users, but seldom thought about carefully by programmers actually writing the program. This is because programmers are dealing with what is possible given the actual program and the time available to make changes.

In the case of TCHART, there was ample time to think about a good interface because the program was modified from a primitive form to the current version over a two-year period. A lot of comments from users were received, some contradictory, all useful.

Eventually, we decided that a good interactive drawing program must have the following properties:

1. It does what the user expects. The program is consistent and symmetric in its menu/command structure.
2. It is easy for him to learn. A program is easiest to learn if it is simple in structure. It is also best if there is little the user has to remember. For example, a menu/command set which is always available for viewing is best.
3. It requires a minimum number of steps to accomplish a function. The user should not be forced to traverse menu lists and/or restate information he has previously given the program. The program should retain all reusable information.
4. The response to an accomplished function is immediate and obvious. The screen is immediately updated in response to user chart changes. Messages to the user appear if functions cannot be performed.

5. It saves the user from himself. The user is not allowed to move chart segments to irretrievable locations outside the screen. He is protected from any abort which will terminate the program prematurely. Error messages keep him informed if he makes a mistake.
6. It allows control of all aspects of the process being performed. All of a developing figure's attributes are under user control.
7. It provides for hierarchy of detail. Scale functions are provided. Previously generated chart figures may be read into specific areas of the screen.
8. It allows precision control for picking functions. The user is allowed to specify in some way the density of possible picking locations. This allows him to line up drawn figures and/or to locate specific chart points without using inhuman precision.
9. Chart files are saved on user command. This allows the user to save different versions of the developing chart without exiting from the program.
10. Chart files are read on user command. The user may read any number of chart files during program execution. This eliminates the need to recreate chart elements.
11. It allows the user to save his drawing environment. Specific attribute information such as character type and font, color, etc., may be read/saved at any time to assist the user's memory between chart drawing sessions.
12. It has a help facility. New users can receive information on menu or command functions during program execution.
13. It is expandable for future enhancements. If new functions are required, they can be added without much alteration to the user interface.

Attempts to Reach the Goal - TCHART Development

Can a program based on a graphics package, which runs on many different devices and which contains no device-specific code, reach the goals outlined above? The story follows.

TCHART was originally written to provide users with a chart drawing tool to be executed primarily on the TEKTRONIX 4014, thus its name T-CHART. It was based on DI-3000 and was a well-structured program.

Since we had no product like it at the time, a programmer was assigned to look at the program and "clean it up." Naturally, as soon as it was cleaned up enough for anyone to make much use of it, complaints about the way in which it worked started to come in.

Over a period of two years, as time has been available, TCHART has been modified. Some of the major changes have been the following:

1. The data base was completely reformatted.

2. FORTRAN 77 DI-3000 routines were substituted for FORTRAN 66 routines.
3. The menu structure was consolidated from two levels to one.
4. New drawing functions were added.
5. Input/Output functions have been added.
6. Attribute functions have been enlarged to make use of most DI-3000 capabilities.

TCHART today is an interactive program based on DI-3000 which creates and modifies charts and diagrams. The user draws diagrams on the graphics terminal and plots them to the specified hardcopy device.

When TCHART is executed, the user is prompted for the name of the chart file he wishes to generate or update. This chart is then displayed with the TCHART menu set. The TCHART menu set consists of six menus:

ACTION	Chart operation to perform
OBJECT	Type of figure upon which operate
SWITCH	Switch function options
OPTIONS	Grid and attribute options
INPUT/OUTPUT	Plot output, chart file generation and input
EXIT	Exit with/without saving existing chart

The user picks from the menu set with the terminal locator input device. No particular order of selection is enforced. The function performed in chart area is determined by the user's selection for the ACTION and OBJECT menus. Figure 1 shows the TCHART menu during a user chart drawing session.

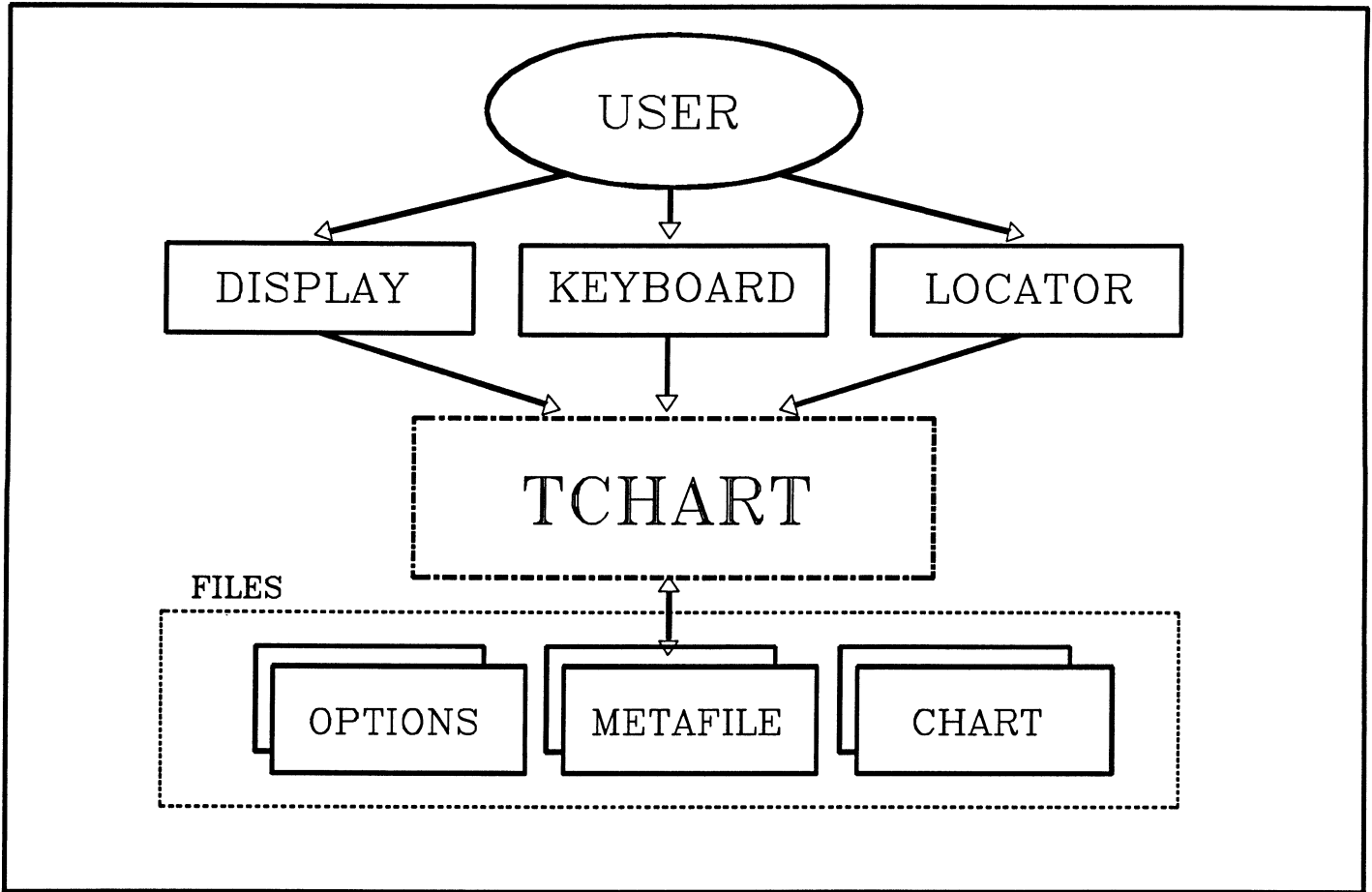
The ACTION menu defines the operation to be performed on existing chart figures. The selections available in the ACTION menu are as follows:

ADD	Add a figure to the chart
MOVE	Move a figure to another location
COPY	Copy a figure to other locations
ROTATE	Rotate a figure a specified number of degrees
SCALE	Scale a figure's size up or down
CHANGE	Change a figure's existing attributes
DELETE	Delete a figure from the chart

The OBJECT menu defines the type of figure to be created or modified. Creation of the following objects is accomplished through the user selection of a minimum number of defining points:

ARC (3)	PARABOLA (3)
ARROW (2)	POLYGON (n)
CIRCLE (2)	RECTANGLE (2)
DIAMOND (3)	SYMBOL (2)
ELLIPSE (3)	TEXT (1+Text)
LINE (2)	TRAPAZOID (3)
PARALLELOGRAM (2)	TRIANGLE (3)

Attributes such as color, line style, line width, fill, character style, font, and size may be selected from the OPTIONS menu to apply to the



	ADD MOVE COPY ROTATE SCALE CHANGE DELETE	TEMPLATE LINE ARROW DIAMOND RECTANGLE POLYGON TEXT ARC CIRCLE	PARABOLA ELLIPSE ANY	MENU ON/OFF MENU TOP/BOT AREA ON/OFF CENTER TEXT BOX TEXT GMAJ DISP GMIN DISP HELP ON/OFF	COLOR FILL CHARACTER LINE EDGE CONSTR SPEC	READ WORKING SAVE WORKING READ OPTIONS SAVE OPTIONS READ META SAVE META *PLOT FOTO	REDRAW RESTART EXIT QUIT
--	--	---	----------------------------	--	---	---	-----------------------------------

Figure 1

figures and text generated. The TCHART OPTIONS menu also allows the user to define the mesh density of a snap-to grid. This grid (only displayed if desired) is used to constrain the user locator input to assist him in lining up his picture elements.

At any time during TCHART execution, the working chart may be written to disk without interrupting chart development. In the same way, previously generated working files may be added to user-defined windows in the developing chart.

Grid and attribute selections from the OPTIONS menu may be written to file for later retrieval to provide uniformity between chart generating sessions.

DI-3000 metafile output may be generated for use by other DI-3000 based routines. In the same way DI-3000 metafiles may be read by TCHART so that plot output generated by other routines may be "decorated" by the user.

Figures 2 and 3 are examples of TCHART generated charts.

Some Goals Unmet

The most obvious loss associated with the use of any graphics package is in performance. DI-3000 overhead is particularly noticeable during figure generation and the execution of picking functions.

Use of a graphics package as a device interface meant that the TCHART program lost direct control over the update of display list memory (when present). This was an important negative result from the standpoint of the immediate response goal because it made a complete redraw of the screen necessary to update the chart following DELETE or MOVE operations. Dragging functions could not be performed. Because a complete redraw of the screen is so time consuming for a complex chart, screen updates of this kind only occur when the user requests a redraw.

The lack of a textport (ASCII screen) control made necessary a somewhat clumsy user message system which overwrites the developing chart until the user requests the screen to be redrawn. This difficulty has also prevented the implementation of a help facility because of the additional screen clutter.

It should be noted that most of the limitations imposed by the use of device independent software were only important with reference to the newer terminals which would allow the programmer display list and/or ASCII screen control. In practice, many of our engineers are still using TEKTRONIX 4014 terminals where immediate screen update without complete redraw is impossible in any case.

Benefits of the Use of a Graphics Package

The basic benefit of a device independent graphics package is the simplification of code to do output to a variety of interactive and hardcopy devices. An extra benefit in the use of DI-3000 is the large number of other tasks performed by the software. Many TCHART capabilities are really DI-3000 functions slightly transformed into a form which is natural for the user.

Circle, arc, rectangle, line, polygon, generation functions were directly passed from TCHART to DI-3000 subroutine calls.

DI-3000 keeps track of figure locations, freeing TCHART of much of the code needed for picking functions.

Users were given indirect control of DI-3000 attribute functions: line style, color, fill pattern, and character definition.

Extensive software character fonts were provided for Graphic Arts and Polygonal character types.

Advantage was taken of the DI-3000 segment transformation capability to provide TCHART ROTATE and SCALE functions.

An unexpected benefit of the use of DI-3000 was the metafile capability. This feature allowed readable file output to be generated from any DI-3000 based plot routine. By writing code to read this file, TCHART was able to take advantage of engineering and presentation chart output generated by other routines.

Conclusion

It seems inevitable that some departure from TCHART device independence will be made in the future. Additional code will be added to enquire the device type and to bypass DI-3000 to perform such functions as:

1. Background writes to "delete" figures from the screen.
2. Textported message output scrolled in a defined area of the screen.

These capabilities can be added as time is available. "Unrecognized" devices will utilize the vanilla device independent capabilities. Even though device independence has proved to fall short of full utilization of device capabilities, it has been the wisest place to begin program development. Starting with device independence as a goal has resulted in development of a program which is usable on any device while not precluding special device tailored software at a later date. When such software is implemented, it will be functionally separate from the main design of the program, and as such program changes will be easy to control as devices come and go in the future.

SANTA'S GIFT INVENTORY

Total Requested Gifts

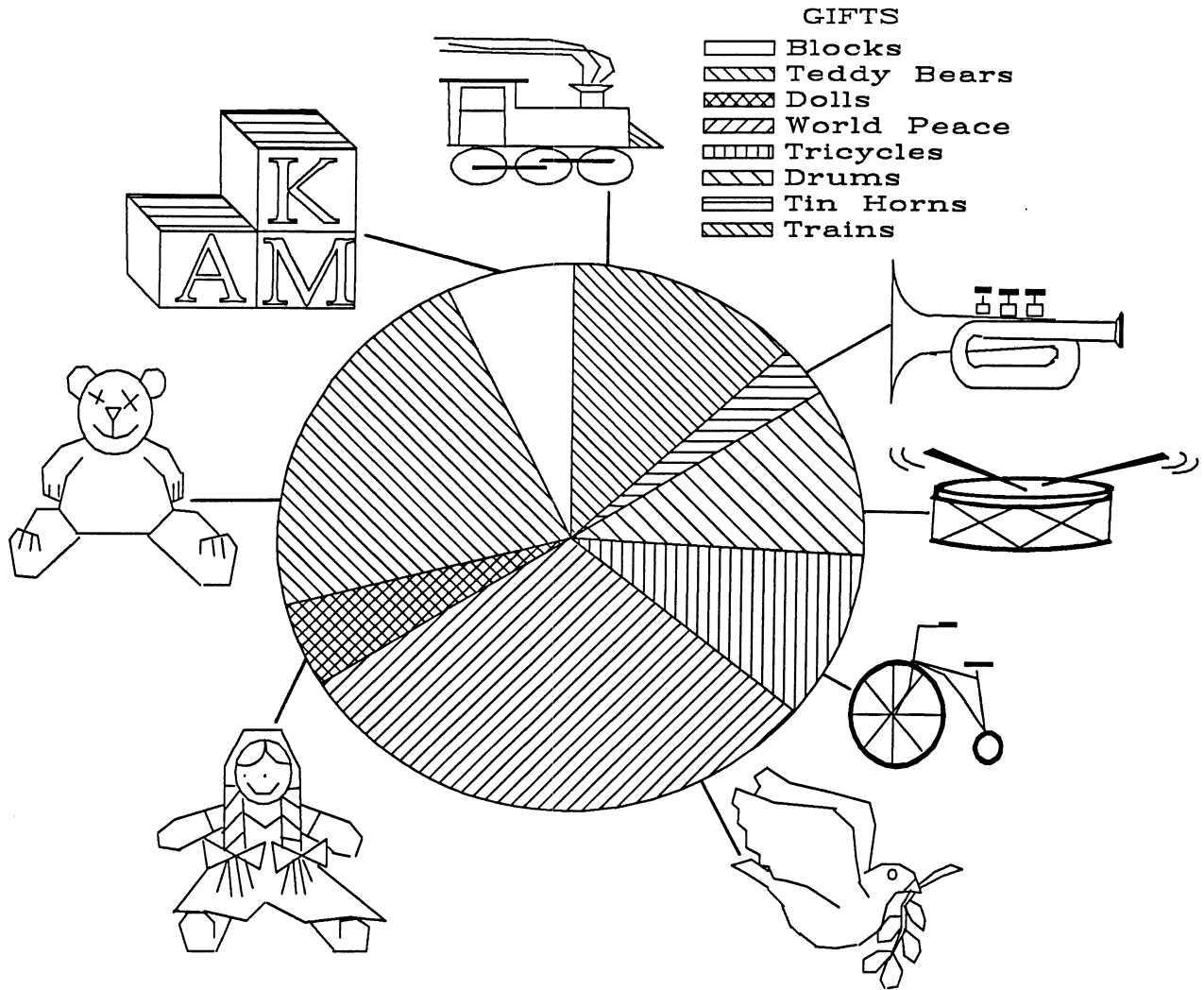


Figure 3

BSD TECHNOLOGY VAX's

(2.25 - 3)

Ethernet/DECnet Trunk

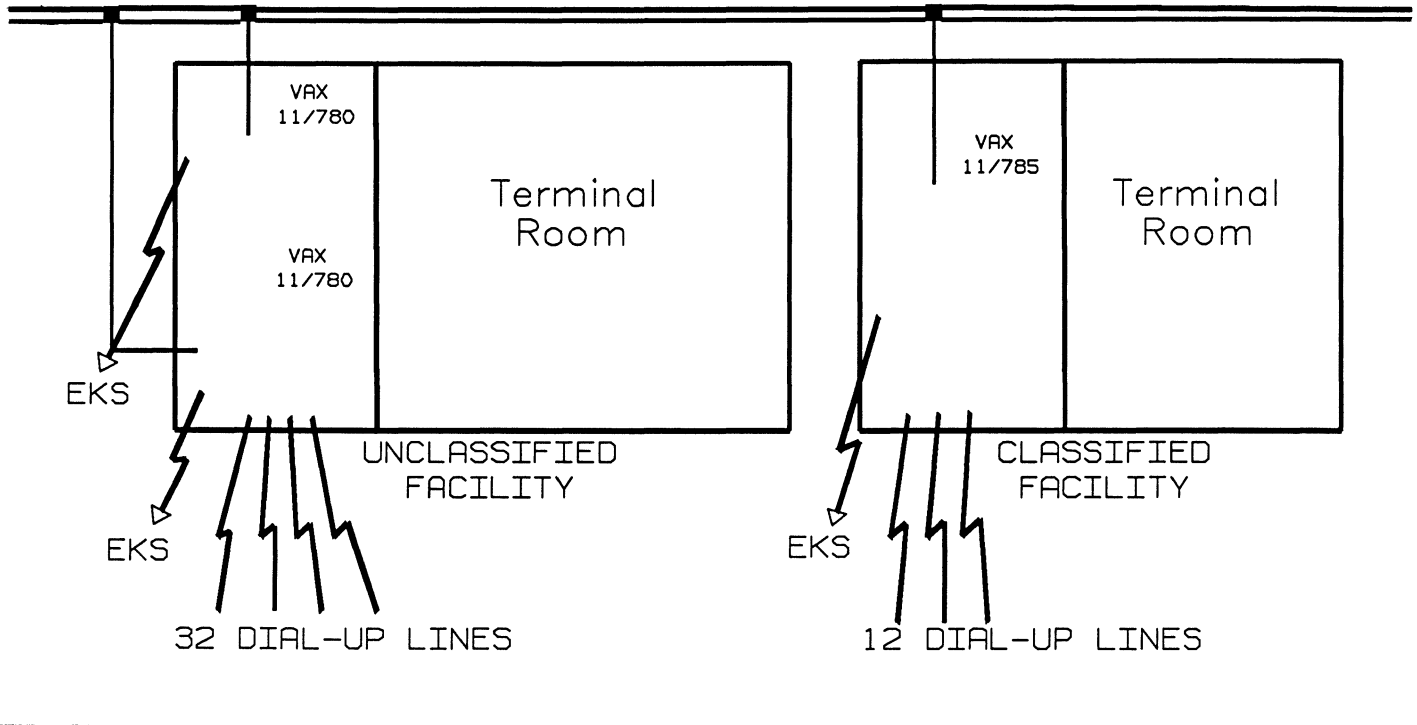


Figure 2

A SOFTWARE DISPLAY SYSTEM FOR MEDICAL IMAGE PROCESSING

Luc Bidaut, Division of Nuclear Medicine and Biophysics,
Department of Radiological Sciences & The Laboratory
of Nuclear Medicine (DOE), UCLA School of Medicine,
Los Angeles, California 90024

ABSTRACT

For years, medical imaging has been lacking a truly versatile system to display and process images coming from various sources.

The few systems commercially available were just able to handle local data coming from a specific device and nothing else. This situation was particularly sensitive in our division which daily deals with images coming from both X-ray Computed Tomography (XCT), Nuclear Magnetic Resonance (NMR), Single Photon Emission Computed Tomography (SPECT), Positron Emission Tomography (PET), and autoradiography digitizers.

This paper describes the design of an original display system intended to address this lack and to be commercialized with a new generation of Positron Emission Tomographs.

INTRODUCTION

In a medical research laboratory, the needs for a display system range from data display and processing to slide preparation for meetings.

The data manipulated during a display session can come from sources as varied as X-ray Computed Tomography (XCT), Nuclear Magnetic Resonance (NMR), Positron Emission Tomography (PET), Single Photon Emission Computed Tomography (SPECT), and autoradiography (Figure 1).

This diversity requires a system able to handle various high-level functions, some of them very linked to the nature of the data to process.

Such a system was not currently available in our field and it had to be designed and implemented in our division to counteract the new needs created by a rapidly increasing number and variety of the images we had to process and study.

The display system described in this paper has been designed to handle a wide choice of display functions, along with a complex display memory management.

By restricting the device dependent functions to a minimum (mostly device access), and handling all the memory management through high-level routines, the design allows transportability and easy further maintenance/development of the system.

REQUIREMENTS

The basic requirements of a medical imaging system can be summarized as follows:

Display images from various sources and of various dimensions, both spatially and dynamically. The images can come from several "medical" sources (XCT, NMR, PET, SPECT,

autoradiography,...) and have various spatial dimensions and dynamic range.

Drawing any kind of figure on the screen, with the possibility of relying on specific data. This feature is more particularly needed to draw Regions Of Interest (ROI's) on displayed image data which are not actually stored within the display memory, and to process these ROI's offline.

Ability to process images both before and after displaying them on the screen, which is requested to enhance whatever information needs it. A preliminary processing could be a smoothing or contrast enhancement, a subsequent one a color scale modification.

Display of any kind of informations on the screen, both by text and graphics, to allow a good understanding of the information provided to the user.

Some features may be added for convenience and to describe a truly versatile system:

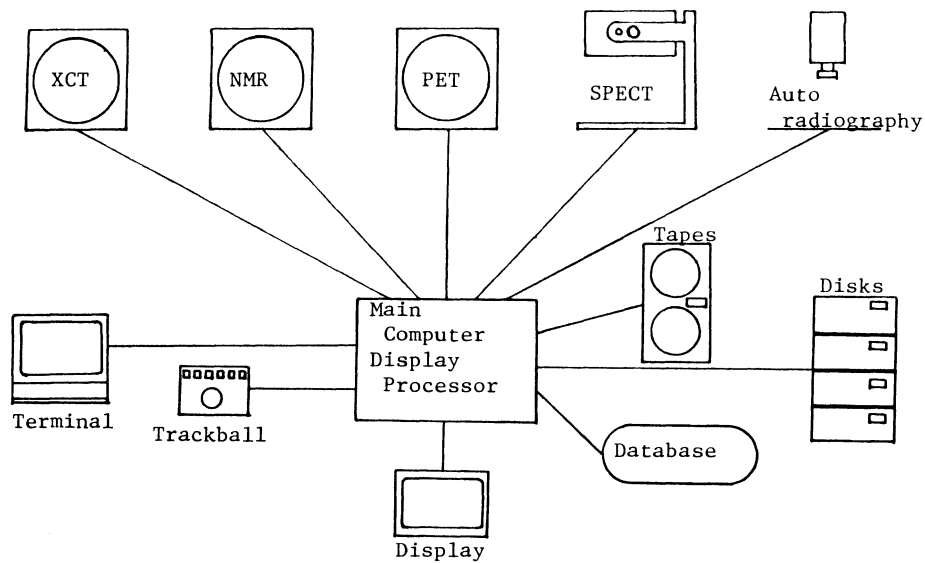
Extensive use of interactive medias for user input. This feature is almost unavoidable whenever ease of use and speed of processing are involved.

Several addressing modes and scan sequences to access the memory, which is needed to deal with various data types.

Several overlay planes to be used for plotting and text, permitting the independent handling of graphic and text information.

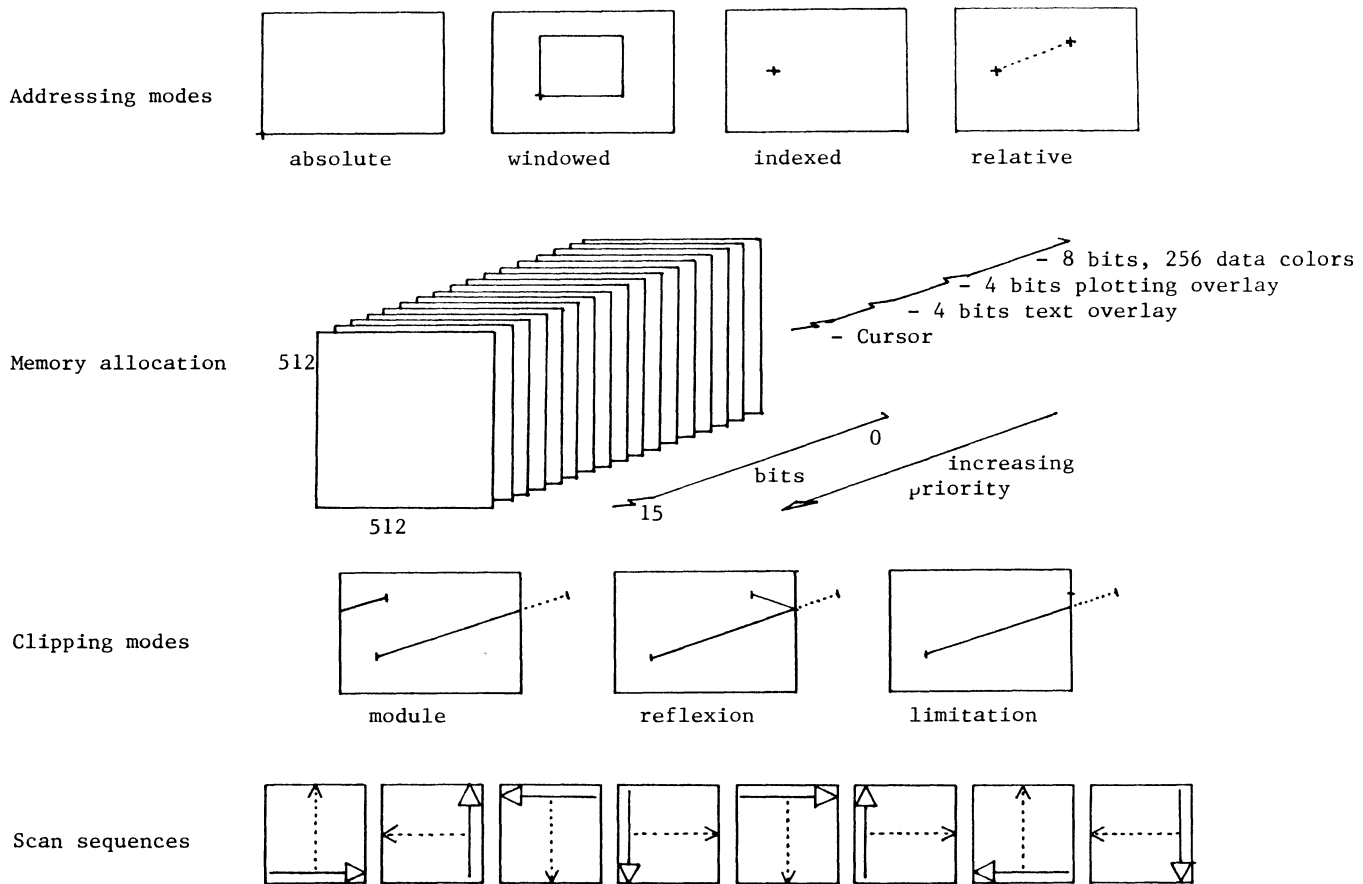
Give the user the possibility of defining his own display screen format and his own color scale to adapt the system to his application.

Saving within files of any color scale and screen format as defined by a user for further restoration whenever necessary. This feature makes the color and format definition step necessary



Medical Imaging Configuration

FIGURE 1.



Display Memory Management

FIGURE 2.

only once for a specific user application.

Providing the user with a screen editing utility to enhance information display and allow more attractive displays, up to slide creation.

Ability to save/restore the display screen as often as requested, which is particularly useful to keep track of a specific display session or of a complex screen editing.

Considering these needs, the system design has been divided into several categories:

- Color scale, screen format management
- Image display from the current file structure
- Pre-processing, post-processing of images and data
- Screen editing, saving/restoration.

HARDWARE BASIS AND ITS SOFTWARE UPGRADING

Besides the functionality required by our environment, we also sought a maximum independence from the hardware portion of the display system, to ensure that we would further be able to use new hardware while retaining the same processing power.

The hardware used for our application was consequently assumed to be a "minimum" one for the technology actually available:

- Screen definition of 512x512, one absolute addressing mode
- 16 bits data:
 - 12 bits (4096 levels) video look-up table with 3 fundamentals (red, green, blue) of 8 bits each (256 values)
 - 4 bits of overlay
- One trackball and cursor management.

From this basic hardware configuration, the memory management has been boosted by software to:

4 addressing modes:

- Absolute, which deals with the full screen
- Windowed, which restricts the memory access to the selected display window
- Indexed, which fixes an origin within the display memory
- Relative, which takes the last accessed point as the new access origin

8 display scan sequences: they set the way the data are going to be written/read within the current window of the display memory

3 clipping modes for plotting: modulo the current window, reflexion on its boundaries, limitation to its boundaries

Up to 256 color levels for data display that can be edited individually to define a color scale

4 bits overlay for plotting which will be written on top of the current color scale levels

4 bits overlay for text writing, on top of all the other colors.

Figure 2 describes the system characteristics.

Eventually, this software can take into account any function handled by other hardware (window management for example), which can shorten processing time, along with some possible limitations or new parameter values (screen definition, number of color levels, ...).

To allow a complex handling of every parameter, the system has been divided into several routines, each one dedicated to a specific parameter.

Whenever necessary, the modification resulting from one of these routines is passed along a display session by using a local section to store the current parameter values.

This design sometimes requires chaining from the current routine to another one, whenever a parameter (format, color scale, etc.) needs to be modified within the display session.

Figure 3 describes such a structure.

IMPLEMENTATION

The implementation of this system took place on VAX's (730 and 780), running VMS (3.6 and 4.0). Besides the display device access written in Macro 11, the whole package has been written in RATFIV (FORTRAN, 4.1 compiler) for easy transportability.

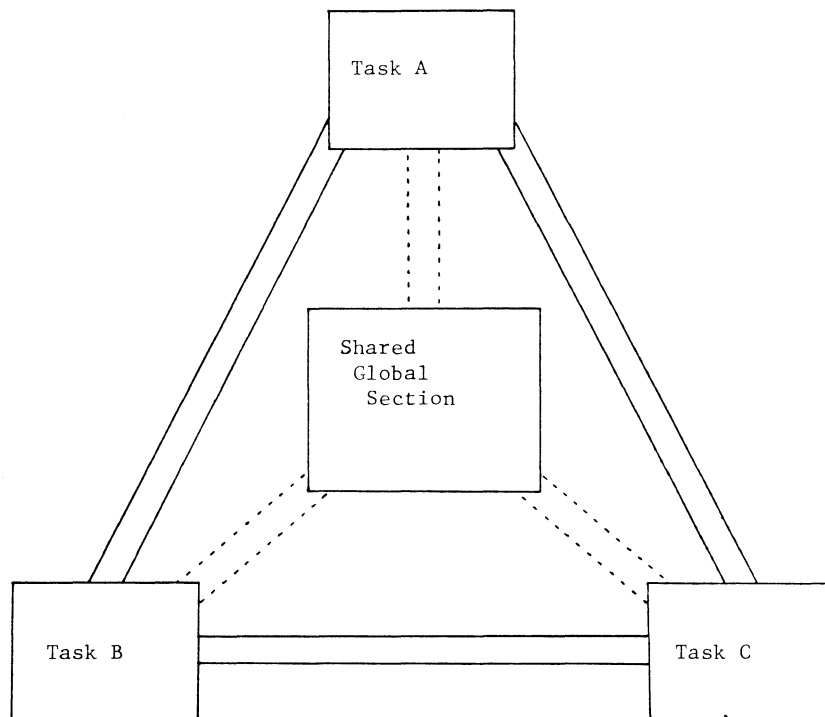
The routines have been divided in two sets: One needed to display images and one mostly used for data management (images, curves, ...) and customization.

The basic set of routines is:

- DISINIT to initialize the display system
- DISCOLR to handle the color scales
- DISFORM to handle the screen formatting
- DISDISP to display images

The extended set is:

- DISCINE to roam through the screen windows (movie mode)
- DISLABL to edit the screen (drawings, text)
- DISSAVE to save/restore part of the screen.



----- Parameters, Data transfers

==== Task chaining

Software interconnections

FIGURE 3.

THE WAY IT LOOKS

From your point of view (that of the user), this display system looks like a multiple parameter system, each one being accessible for a new definition, along with several utilities allowing you to display whatever information is needed, wherever you need it.

The default color scale or screen format that will be in effect anytime you enter a new display session can be defined at any time (DISFORM, DISCOLR). These defaults can either come from a software predefinition, a previous selection stored in a file, or an on-line modification.

Once a color scale and/or a format have been selected, they are in effect for the rest of the display session, at least until the next modification. This feature allows the display system to keep track of its contents during a session.

You can then display images, or process the images already displayed, within any window of the current screen format (DISDISP). These images mostly come from files where they are stored as matrices. For our application, as 16 bits data represent a sufficient dynamic and a good information/storage space ratio, most images come from integer files. Their storage format (directories, parameters, ...) was defined by taking into account medical imaging structure, but it is easily upgradable to accept any other kind of data, particularly that coming from our centralized database

SOME INTERESTING POINTS

Several parameters can affect an image display, and you can select any of them before any display operation:

Screen window: any of the ones defined in the current format: up to 64, any shape, size and location on the screen (even overlapping ones).

Scan sequence: 1 among 8. They can be matched to the data representation which the best suits the application: a scan can better be used to display images as they should appear, while another one can be best suited for matrix data representation.

Zoom to consider: square, rectangular, integer, real:
a square zoom will not modify the shape of the data, a rectangular will expand this shape along its largest axis. An integer zoom will be faster than a real one because of its equivalence to pixel duplication, but it will be less optimal for screen space occupation.

Data portion to zoom, which can be selected from a previous display (even already zoomed), and displayed in any current screen window.

Range of the data values to display: it can be used to actually compare data values when displaying and not only the way they look with their own dynamic: an image with data from 0 to 1 may look the same as an image from 1 to 2, but they will look different if the display value range is from 0 to 2 for both displays.

Range of the color levels used for displaying: it limits the actual portion of the color scale used for display. It can be used to display data on several color scales, each one being a portion of the current one.

Bit planes to be accessed within the display memory by the next display operations. This allows retaining some of the bit planes (data, overlay, etc.) while affecting others.

The plottings are all restricted to the current window, with the clipping modes previously defined (modulo, reflexion, limitation) (Figure 2).

Besides solid and various dot line selection, they can take place with 9 patterns (point, circle, square, +, x, star, triangle, lozenge, and even text), that can be combined together. Any new one can easily be defined.

The curve plottings can take place in 8 different referential orientations and be automatically matched to the current window dimensions.

The screen savings are done on all or part of the screen and on all or part of the display memory bit planes (16 total). The data read from the display memory (maximum 16 bits data) are run-length compressed on a byte basis (most significant [overlays] and least significant [data] bytes separately) before storage in a file. This compression is really efficient for most of the display screens which contain a relatively low number of pixels ("picture elements") actually used, and more particularly for graphics and text. The restoration can be done on any bit plane of the display memory and with any among 8 scan sequences (rotation/reflexion).

A summary of the options for each of the main routines follows:

The ones marked (*) are mainly handled through the interactive media (trackball).

DISINIT :

Soft initialization: initialize the communication with the display device. This step is needed whenever entering a display routine.

Color scale: loads the default color scale for the user. This scale can either come from a file or be a predefined one.

Screen format: loads the default screen format for the user. As for the color scale, this format can either come from a file or be a predefined one.

Hard initialization of the display device: initializes the hardware part of the display system by clearing memory and registers and loads the default color scale and format specific to the user.

At the beginning of a display session, the color scale and the format are set to the user's defaults as he defined them.

DISCOLR : This routine handles color scales on two levels:

A local scale which is only manipulated through the routine

A current scale which is the one used by the system along the display session

Once a local scale has been set, it can either become the new current one for the rest of the session or be stored into a file for further use.

Local scale options:

Number of color levels: this option is the main one for this routine from a display point of view for it affects the way the data are going to be written within the display memory. It sets the actual dynamic of the image visualization on the screen.

Display or Erase its representation on the screen: the color scale is represented on the right of the video screen. This reference can be disabled whenever the color scale does not add anything to the actual screen content (text slides, ...).

Modify color levels (*): interactively modifies the percentage of each fundamental (red, green, blue) within a specified color level of the color scale.

Switch, Invert, Copy portions of the color scale (*): exchange the location of 2 portions of the color scale, put a portion upside/down, copy a portion on another location.

Define new colors (*): interactively select the amount of each fundamental (red, green, blue) for a new color and copy it in a portion of the color scale.

Load part of the color scale with predefined scales (compression, expansion to match the actual portion to load) (*): These scales may be coming from a file.

Move a colored window through the color scale (*): fills all or part of the color scale with a fixed color.

Rotate part of it (*): interactively rotate all or part of the color scale, which sometime results in interesting visual effects.

Enhance the contrast by modifying the levels repartition (logarithmic and linear) (*): it interactively modifies the way the colors succeed within the color scale (shorten the upper levels portion and enlarge the lower levels one, or the contrary, ...).

Store the color scale issued from any kind of previous processing within a file for further use. The color scale is compressed before storage to

save space. It may be restored whenever necessary during a display session for further modifications or to use it as the current one for the rest of the session.

Current scale options:

Read the current scale for it to become the new local one

Write the new current scale as the actual local one.

DISFORM : This routine is similar to DISCOLR for screen formatting. Instead of manipulating color scale levels, it manipulates screen windows that will be further used to restrict the access within the display device memory.

A set of windows represents a format.

As for DISCOLR, there are two levels of format handlings:

A local format temporarily manipulated through the routine

A current format which is used for every display operation

Once a local format has been set, it can either be saved in a file for further use or become the new current format for the rest of the display session.

Local format options:

Display or Erase the format representation on the screen (green dotted lines around the window boxes).

Modify or Delete a window (*): interactively select a window to erase from the format description or interactively modify.

Load the format with predefined formats. These formats may be fixed size windows ones or may be coming from a file.

Store the local format within a file for further use. As for the color scale, it may be restored whenever necessary during a display session for further modifications, or may be used as the current one for the rest of the session.

Current format options:

Read the current format for it to become the new local one

Write the new current format as the actual local one.

DISDISP : This task deals with all the image display once a color scale and a format have been defined.

The options for this routine have been divided in two sets, the second one being reserved for skilled users wanting to modify the basic parameters of the display system to best match their own display requests.

Basic set:

Color scale handling on the flight: load basic software predefined color scales (UCLA, Hot Point, Tricolor, Black & White) as current for the rest of the session.

Screen formatting on the flight: chain on the formatting utility to modify the current format (modify it, load a previously defined one, ...)

Selection of the next display window and of the automatic increment to use between each display (*): it sets the way the display utility is going to access the windows of the current format. The increment chosen may be positive, negative or null (same window).

Selection of the type of image to display: it tells the display system what kind of information to look for when displaying the next images. This information is linked to the specific way any kind of data is stored within our file structure. We currently have 3 different types of data (images, scans, attenuations) but this number can be expanded or reduced according to any new application.

Enable/disable the writing of the image name: if enabled, the writing of the identifier (file name, image index within this file) for each image which is going to be displayed occurs in the window it is displayed in.

Show the parameters of an image (dimensions, values): it eventually allows the determination of a value range to display and compare a set of images, or a size of window for the screen format.

Set the range of values to consider for display: the display of images becomes absolute: images can be compared on a value basis instead of just the dynamic of their representation with the current color scale.

Display an image: access image data and display them according to the current parameters.

Smooth a displayed image (*): eventually smooth a display window when too noisy.

Zoom part of a displayed image (*): interactively selects a portion of a previously displayed screen window (which corresponds to the portion of a data matrix stored in a file) and zoom it on another window of the current format. The display system keeping track of its contents, the zoom takes place on the initial data and not only on the ones which are contained within the display memory. It makes it possible to take advantage of the true data resolution and not only the one of their representation on the screen.

Erase a window (*): interactively select a window of the current format to clear.

Clear the screen.

Restricted set:

Addressing mode selection: absolute, windowed, indexed, relative:

the absolute mode forces the program to use the full screen as the only window. The windowed one forces it to use the windows defined by the current format. Indexed and relative are some special modes (see Figure 2), which may be used by some functions (plottings, ...)

Scan sequence selection: one among eight: it allows matching the display representation to the actual data structure (image, matrix data, ...).

Zoom control: this parameter is used whenever an image dimension do not match the size of the window where the image data must be displayed. Several modes are available: integer (pixel duplication, which only works for expansion from data to screen window) or real (maximum space occupancy) factors, forced or not to square (i.e. square for no deformation of the image shape).

Mask on the bit planes of the display memory to access when displaying. It restricts or extends the bit planes (16 of them) which are going to be affected by the following display operations. A display may be done on just one bit plane or on any combination of the 16.

Range of color levels to consider within the current color scale when displaying: it sets the portion of the color scale which is going to be used for data display. This can be used for multiple color scale display when the current color scale has been loaded with a corresponding scale.

DISCINE : This routine allows simulation of a movie mode (cartoon-like) display by roaming all or part of the current format windows on the screen. It uses another feature of our hardware: roaming through the display memory (origin of the display memory) and hardware zoom (integer factors 1, 2, 4).

Screen format handling: chains on the screen formatting utility (DISFORM) to set the current format whose windows are going to be roamed.

Image display, deletion: chains on the image display utility (DISDISP) to display the images that will further be roamed.

Range of the windows to roam and basic increment (algebraic): it selects the windows that are going to be roamed and in which direction (increasing or decreasing index within the current format description) when in automatic mode (see below).

Interactive (trackball) or automatic (time interval) roaming: selects whether the roaming will be done interactively by using the trackball, or automatically according to the window increment and a time interval, previously chosen, between each display.

Size of the screen window where to display a roamed one: it can be set to the full screen or reduced to a quarter of it, independently from the initial window size.

DISLABL : This routine deals with every kind of drawing and text writing on the screen. It is particularly useful to write a text or to edit and draw pictures (slide preparation, ...).

Clear part of the screen (display memory), and on which bit planes.

Draw basic figures: squares, circles, rectangles, ellipses (*): plot a figure with the current plotting parameters (see below) on an interactively chosen location.

Fill basic figures, frames (2 concentric figures), two-figure combinations (*): after 2 basic figures (see above) (1 + width for frames) have been selected, the space belonging to only one of them is filled with the current filling parameters.

Select filling, plotting parameters: color, type of overlay (additive, for color of the combination, or priority for color of the most significant one), type of plotting (additive which does not erase the other bit planes, destructive which does), solid, dotted lines, hash lines (spaced lines for filling) and filling directions combination. The filling (shadowing) can occur with any of the plotting parameters and with any combination of 4 directions (--, |, /, \).

Draw pointers: arrow, circle, square (*): after selecting the type and size of the pointer pattern, interactively selects its origin and direction.

Text writing: this option does everything for text writing/erasing.

Mask on the bit planes to access within the display device memory:
this mask is used whenever a text writing/erasing is requested. As it accesses the data and overlays bit planes, it sets the color of the text to be written on the screen.

Blinking for the device overlay which is exclusively used for text writing (bits 12 to 15 of the display memory). This can be used to enhance the visual impact of some text (warning, ...).

Writing directions: 8 of them covering rotation and reflexion of the current text.

Writing mode: normal (colored text), box (black text, colored box), italic or not for each of the previous choices.

Expansion factors along each character dimension: the basic size of the text characters along each one of their dimensions can be independently set.

Select text location (*): once the text to write has been selected, the location of the writing on the screen is interactively selected on the screen according to the current text writing parameters (string size, expansion, direction).

DISSAVE : This routine handles the saving and restoration of all or part of the screen.

After having been compressed (run-length), the least and most significant parts (bytes) of the display data are stored within a file.

The restoration will pick these data and restore them on the screen whenever necessary.

Saving:

Select the bit mask on the planes of the display device memory to save: it restricts the saving only to the display memory bit planes we are interested in (image data, plotting, text, ...).

Select the window to save (*): interactively selects the screen portion to save.

Restoration:

Erase part of the screen (*): interactively selects the screen portion and the display memory bit planes to erase.

Select the mask on the bit planes of the display device memory to restore: sets the bit planes to be affected by the restoration. It makes it possible to only restore part of the saved data (image data, plotting, text) and/or to keep some planes previously filled in the display memory.

Select the restoration scan sequence (1 among 8): allows rotation/reflexion of a previously saved screen portion.

Select the location where to restore (*): interactively selects the screen portion where to restore a previous saving.

Set of subroutines:

Besides the main display options available through the routines previously described, several basic display operations are also accessible through a set of high-level subroutines.

Some of them deal with the following functions:

Initialization: all the options of DISINIT

Display parameters setting and reading (format description, addressing mode, number of color levels, etc...)

Plottings: any kind of pattern (point, cross, star, circle, square, lozenge, triangle, text, ...) and line type (solid, dotted with variable cyclic ratio) to draw line, polygon, rectangle, ellipse, curve (with automatic matching to the display window and axis plotting).

Text writing: any color, direction (8), size (2 independent expansions), mode (normal, box, + italic)

Interactive media access: several enabling modes, both for cursors (display, shape and tracking), and switch management.

Display of data matrices with the current parameters.

CONCLUSION

This package is now currently in use in our division and in other facilities around the world for all the studies currently being carried out in either clinical examinations or medical research.

The set of subroutines has been used extensively to develop the display aspects of regions of interest computation and several display-related utilities: contour drawings, edge detection, profile and histogram computation, automated roi drawings, etc...

It is used for developing new means of looking at medical data, with more interactive/display related modalities than the ones actually available, both as a time saver and as an aide for more accurate interpretations and/or diagnosis.

This system will also be a main component of the software developed for a new concept in medical imaging: the PET clinical integrated center. This will be a fully computerized medical center whose mainframe will link acquisition devices, database, remote workstations (almost entirely dedicated to display processing), and handle all the communication and data processing management (Figure 4).

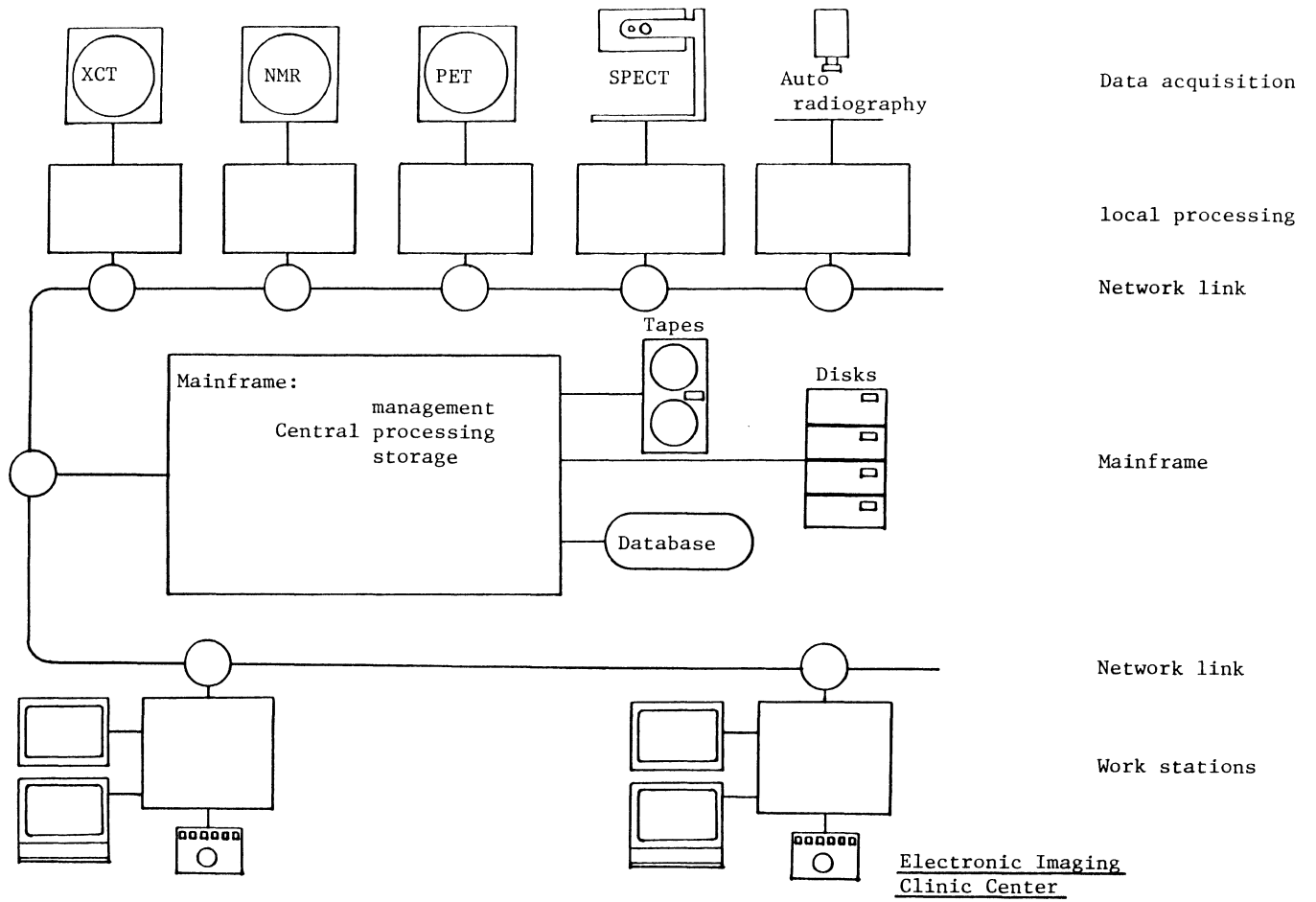


FIGURE 4.



Randolph M. Pacetti
AT&T Technologies, Inc.
Lisle, IL

ABSTRACT

PCs can be a valuable tool for DECsystem-10/20 users. By moving selected tasks to a personal computer, system administrators can reduce the load on the mainframe and extend the variety of software available to the end-user. But, the administrator is also faced with the problems posed by communication methods, security, and user retraining.

This session will review those and other PC issues and suggest an approach for integrating PCs with a DECsystem-10/20 running 1022. We will review a new System 1022 feature which allows users to extract 1022 data and create WKS (LOTUS) and DIF formatted files. We will also show how MOBIUS (a PC interface product from FEL Industries) can be combined with System 1022 to simplify access to the extracted data and to other 1022 features.

Agenda

1. Introduction
Barbara Bersack, Software House
2. 1022 PC extract feature
John Duesenberry, Software House
3. MOBIUS
Chris Kayes, FEL Industries

Introduction

Barbara Bersack, Software House

Personal computers are assuming an increasingly important role in the workplace. At Software House, we have been examining the ways in which our users can combine System 1022* with their PCs. This session focuses on our approach and how it was selected. Version 117B includes a PC extract feature that allows users to create WKS and DIF formatted files from within 1022. MOBIUS* can then be used to simplify access to the data and to other System 1022 features.

Why are PCs so popular? According to one survey**, which examined 83 users of PCs and 34 information system managers in ten major U.S. firms, PCs are attractive because they give end users flexibility at a low cost. Users can choose the type of software they want as well as when and how they use it. Also, they do not need to rely on MIS staff for all of their application development needs.

Although PC users may free themselves from dependence on central MIS support, they still need the corporate data that the MIS staff controls. Communication software and terminal emulators for the PC provide access to that data but also

introduce problems of security, redundant or inaccurate data, and user retraining. Our aim is to find the best way to access corporate data and facilities from a PC while maintaining all the flexibility that these devices offer the user.

To consider how PCs could be combined with 1022, we examined the needs of our 1022 customers. We spoke to a number of our customers who are currently using PCs or who want to integrate them with their DECSystem-10s or DECSYSTEM-20s. We also discussed the 1022/PC issue with many of our customers at our users' conference. This customer information became the basis for the 1022 PC extract feature.

Primarily, our users want to be able to offload tasks from mainframes to PCs in order to increase mainframe performance and user productivity. They want to extract data from 1022 and to upload data from the PC to the mainframe data base. Currently, we have implemented an extract function, and we are looking at ways to handle data uploading.

Most of the customers we contacted use 1-2-3 or Symphony* but also want support for DIF* file format. There is some interest in interfacing with dBASE II*. Since comma-delimited output can already be produced with formatted PRINT commands in 1022, increased support for that feature was considered unnecessary.

*System 1022 is a registered trademark of Software House. MOBIUS is a trademark of FEL computing, a division of FEL Industries, Inc. 1-2-3 and Symphony are registered trademarks of Lotus Development Corporation. DIF is a trademark of Software Arts. dBASE II is a trademark of Ashton-Tate, Inc.

**Quillard, J.A., et al., "A Study of the Corporate Use of Personal Computers," Cambridge, 1983.

Totalling was important to a few users. We feel that subtotalling will become increasingly important. Other requests from our users include labeling capability, control over placement of data in cells, and reasonable defaults.

As a result of user requests and suggestions, we have developed the PC extract feature. This feature involves enhancing the 1022 INIT and PRINT commands in order to create WKS or DIF files.

The extract feature gives users the ability to produce a file directly readable by Lotus software. DIF is directly readable by other spreadsheet programs or importable by existing conversion utilities. Users can take advantage of the sophisticated totalling capabilities already available in the System 1022 report writer. Our defaults in this mode allow the casual user to do a simple INIT/FIND/PRINT ALL/RELEASE sequence. Conveniences for users of 1-2-3 include control over display of individual data items, arbitrary positioning of cells in the spreadsheet, and use of the Range Name feature.

We then needed an easy way to access that data from the PC. Our goal was to find a versatile product with a solid record of performance. We looked for a way that would minimize user retraining. We wanted to build on the existing operating system and 1022 security provisions, thus avoiding the need for new security conventions.

MOBIUS meets these requirements. The 1022 DBA, in conjunction with the MOBIUS user, can quickly enable naive users to extract current information from the data base as often as required. Users can then load the data directly into their spreadsheets or other PC applications.

Extraction of 1022 Data to PC Files:
New INIT and PRINT Features in V117B

John Duesenberry
Software House

1.0 OVERVIEW

This paper previews new features in V117B for the extraction of 1022 data to spreadsheet and other applications on personal computers. Enhancements to the System 1022 PRINT and INIT commands will provide users of Lotus 1-2-3 and Symphony with data in Lotus Worksheet file format from within 1022. 1022 will also provide data in Data Interchange Format (DIF). These features are currently in testing and may be modified before final release.

The following examples illustrate command sequences such as a user might employ to obtain data at his or her PC. We will assume that the user's link between the PC and the host computer is MOBIUS.

2.0 CREATING LOTUS PRN FILES WITH FORMATTED PRINTS

Before considering the new 117B features, let us briefly look at what is probably the most direct method currently available for extracting 1022 data into a file that Lotus 1-2-3 can translate.

FIGURE 1 is an example program; MAKPRN.DMC creates a file in Lotus 'PRN' format. In this format, all data items are separated by commas, text strings are delimited by quotes, and each CRLF delimits a spreadsheet row.

```

!                                     MAKPRN.DMC
!      A DMC to write out 1022 data in Lotus PRN format.
!
OPEN MOBDEM RO.
F SYSD BET 1 10.
SORT LN FN.
INIT 2 DEMO2.PRN.
PRINT ON 2 "From 1022 Dataset: "+$TRIM(SYSDSNAME)+" in " -
+$TRIM(SYSDSFILE) FMT "'',1x,'','',','" 'A'" END.
PRINT ON 2 SYSDATE FMT -
"'',1x,'','',','" "Extracted on: ",','',','",D2,'" END.
!skip a row
PRINT ON 2 FMT "'','" END.
PRINT ON 2 "FIRST NAME" "LAST NAME" "# CHILDREN" "CITY" -
"STATE" "ZIP" FMT 5('"'A,'"') "'',A,'" END.
!skip a row
PRINT ON 2 FMT "'','" END.
PRINT ON 2 FN, LN, NCH, CITY, STATE, ZIP FMT -
2('"'A,'"',''), I, 2('"'A,'"','') "'',A,'" END.
!skip a row
PRINT ON 2 FMT "'','" END.
PRINT ON 2 MEAN(NCH) FMT -
"'',1x,'','',','" "Average # Children:",','',','', F2.1 END.
RELEASE 2.
TYPE "DEMO2.PRN has been created on host. -
It can be FILE IMPORTed to 1-2-3."

```

Fig. 1

FIGURE 2 is the actual output of MAKPRN.DMC. Note the use of quote-delimited spaces in order to 'indent' 1 cell at the start of some rows. The null string is printed to skip a row.

```

" ", "From 1022 Dataset: NEW in MOBDEM.DMS"
" ", "Extracted on: ", "Jul-29-1985"
""
"FIRST NAME", "LAST NAME", "# CHILDREN", "CITY", "STATE", "ZIP"
""
"CHARLES", "CARAGIANES", 3, "DEDHAM", "NY", "02138"
"RICH", "GARLAND", 4, "BRISTOL", "CT", "02138"
"CHARLES", "GOTT", 2, "BRISTOL", "CT", "22209"
"KATHY", "HOUSMAN", 0, "AUGUSTA", "GA", "43220"
"MARK", "JONES", 0, "DEDHAM", "NY", "60064"
"ROGER", "LEVINSON", 3, "BRISTOL", "CT", "11729"
"LOUIS", "MERZ", 0, "ROXBURY", "TX", "77056"
"OLGA", "PONG", 3, "STONEHAM", "MA", "02238"
"ALFRED", "SAVIO", 1, "ROXBURY", "IN", "46225"
"ALFRED", "STEVENS", 0, "BRISTOL", "CT", "02238"
""
" ", "Average # Children:", 1.6

```

Fig. 2

FIGURE 3 shows the sequence of commands the user would give in order to create the .PRN file on the host and load it into 1-2-3 after having defined the device D: as the host area where the datasets and output files reside and having created the PC command "1022" with the MOBIUS MAKE feature:

```

A>1022
*USE MAKPRN
DEMO2.PRN has been created on host.
It can be FILE IMPORTed by 1-2-3.
*QUIT
A>123
/File Import Numbers D:DEMO2

```

Fig. 3

Having imported the file and changed the width of a few spreadsheet columns, the user sees the 1-2-3 screen of FIGURE 4.

```

From 1022 Dataset: NEW in MOBDEM.DMS
Extracted on: Jul-29-1985

FIRST NAME LAST NAME # CHILDREN CITY STATE ZIP
CHARLES CARAGIANES 3 DEDHAM NY 02138
RICH GARLAND 4 BRISTOL CT 02138
CHARLES GOTT 2 BRISTOL CT 22209
KATHY HOUSMAN 0 AUGUSTA GA 43220
MARK JONES 0 DEDHAM NY 60064
ROGER LEVINSON 3 BRISTOL CT 11729
LOUIS MERZ 0 ROXBURY TX 77056
OLGA PONG 3 STONEHAM MA 02238
ALFRED SAVIO 1 ROXBURY IN 46225
ALFRED STEVENS 0 BRISTOL CT 02238

Average # Children: 1.6

```

Fig. 4

To the end user, this is quite straightforward, thanks to MOBIUS. However, if we return to the DMC of FIGURE 2, various deficiencies become apparent, from the viewpoint of the programmer who must write such DMC's:

- * Coding the format statements is tedious and error-prone. The code becomes virtually unreadable, and making a simple change in the program, such as adding a new item to a PRINT list, is more difficult than one would wish.
- * Kludges (the aforementioned blanks and null strings) must be used if empty cells or rows are desired.
- * Ad hoc queries are difficult, especially for end users who are likely to know little of 1022 PRINT formats.
- * The resulting file is not native to 1-2-3, in the sense that the PRN file must be translated and loaded into the spreadsheet and the result saved to a WKS file.

3.0 CREATING A LOTUS WORKSHEET FILE DIRECTLY FROM 1022

By way of contrast, consider FIGURE 5. MAKWKS.DMC is a working program that uses the V117B enhanced INIT and PRINT commands to produce a 1-2-3 worksheet file directly from 1022. A cursory glance shows that the FORMAT statements have been eliminated or greatly simplified.

```

! MAKWKS.DMC
! A program to create a Lotus WKS file
! using V117B INIT/PRINT features
OPEN MOBDEM RO.
F SYSD BET 1 10.
SORT LN FN.
INIT 123 2 DEMO3.
PR ON 2 "From 1022 Dataset: "+$TRIM(SYSDSNAME)+" in " -
+$TRIM(SYSDSFILE) FMT 1X,A END.
PRINT ON 2 SYSDATE FMT 1X,"Extracted on: ",L8 / END.
PRINT ON 2 "FIRST NAME" "LAST NAME" "# CHILDREN" "CITY" -
"STATE" "ZIP" .
PRINT ON 2 ALL.
PRINT ON 2 MEAN(NCH) FMT /,1X,"Average # Children: ",L1.1 END.
RELEASE 2.
TYPE "demo3.WKS has been created on host. -
It can be FILE RETRIEVED by 1-2-3."

```

Fig. 5

Before walking through the code, let's look at how the user would produce the file with MOBIUS and 1022 (FIGURE 6):

```

A>1022
*USE MAKWKS
DEMO3.WKS has been created on host.
It can be FILE RETRIEVED by 1-2-3.
*QUIT
A>123
/File Retrieve D:DEMO3

```

Fig. 6

FIGURE 7 is a screen dump from 1-2-3 after loading the resulting file. The results are practically identical to those obtained with the PRN file, with the exception of the formatting of SYSDATE. (This will be explained below).

```

From 1022 Dataset: NEW in MOBDEM.DMS
Extracted on: 30-Jul-85

FIRST NAME LAST NAME # CHILDREN CITY STATE ZIP
CHARLES CARAGIANES 3 DEDHAM NY 02138
RICH GARLAND 4 BRISTOL CT 02138
CHARLES GOTT 2 BRISTOL CT 22209
KATHY HOUSMAN 0 AUGUSTA GA 43220
MARK JONES 0 DEDHAM NY 60064
ROGER LEVINSON 3 BRISTOL CT 11729
LOUIS MERZ 0 ROXBURY TX 77056
OLGA PONG 3 STONEHAM MA 02238
ALFRED SAVIO 1 ROXBURY IN 46225
ALFRED STEVENS 0 BRISTOL CT 02238

Average # Children: 1.6

```

Fig. 7

Returning to FIGURE 5, notice the following features of the DMC:

- * The new '123' keyword in the INIT command informs 1022 that any output directed to PRINT channel 2 must be formatted as 1-2-3 data cells, rather than the usual ASCII strings. The extension for the output filespec is defaulted to .WKS, and several other actions are taken in the background at INIT-time. The most significant action is the initialization of internal counters which, in effect, always point to the spreadsheet cell to which 1022 will next PRINT data. These counters are automatically updated in the course of printing, and are also accessible to user programs in the form of system variables. They will be discussed in more detail below. For now, it suffices to say that in our example, after the INIT command the counters will point to cell A1, by default.
- * There are two instances in the example of a new format spec - "L" format. (The L stands for Lotus). This format serves a dual purpose:
 1. It enables a 1022-to-1-2-3 data transformation which maps 1022 datatypes (integer,real,date,double integer,or text) into Lotus datatypes (integer, real, or text [label]).
 2. It allows the user to specify a Lotus display format to be used with the item. In our example, a date(SYSDATE) is printed under L8 format. This causes a Lotus binary date to be written, with its format code set to (D1) - 1-2-3's day-mon-yr format. (This

is why SYSDATE shows up differently in FIGURE 7). In the last PRINT of the example, a 1022 function result (type real) is printed under L1.1 format. The 1022 real is converted to a Lotus real, with a format code set to Fixed format, 1 decimal place. "L1" format signifies Fixed format, with the argument after the "." indicating the desired number of decimal places.

* Unformatted PRINT statements also occur in the DMC. Unformatted PRINTS simply default to "L" format by virtue of the fact that they are directed to a PRINT channel that has been INITED to a 1-2-3 file. Thus, in the PRINT ALL statement, 1022 looks at each attribute it is to write out, and produces a 1-2-3 label, integer, or real record on the basis of the attribute's datatype.

The data transformations made under L format are summarized in FIGURE 8:

1022 TYPE	LOTUS TYPE
INTEGER	INTEGER or REAL *
DOUBLE INTEGER	INTEGER or REAL *
REAL	REAL (8087 DP floating point)
DATE	INTEGER or REAL **
TEXT	LABEL***

* Integers or double integers that exceed the range +/- 32767 will be converted to 1022 reals and then to Lotus floating point.
 ** The stored binary date is offset such that Jan.1,1900 = 1. The Lotus format byte is set for (D1) format.
 ***The maximum length of a LABEL is 240 characters, including a Label-Prefix ('') and a terminating null, both of which are automatically added to the string by 1022. Any text expression or literal longer than 238 characters will be truncated to 238 characters.

Fig. 8

FIGURE 9 summarizes the possible L format specs and the resultant Lotus format types:

Form of an L-format spec: rLm.n

where r = repeat count
 m = integer signifying Lotus format type
 n = number of decimal places (0<=n<=15)

Default for m = 12

Default for n = 2, in accordance with 1-2-3 default (n is applicable only to types 1-5 below)

Values of m	Lotus type
1	fixed
2	scientific
3	currency (\$)
4	percent
5	comma (xxx,xxx.xx)
6	+/- horiz. bar graph
7	general
8	day/mon/yr
9	day/mon
10	mon/yr
11	text
12	default

Fig. 9

* Notice also in FIGURE 5 the presence of conventional 1022 format specs: "A" format, "/" format, "X" format and quoted literals. Conventional 1022 formats, in the context of PRINTing to 1-2-3 files, work differently than in normal printing.

"X" format is a means to skip some number of cells within a row. Thus, in our example the 1X format in the first PRINT command repositions the internal cell counter such that the first actual data item in the worksheet is in cell B1.

"/" format is a means to skip some number of rows. The "/" spec at the end of the 2nd PRINT command in the example repositions the internal cell counter such that the next item will be written to cell A3, instead of to the next consecutive row.

Most of the other conventional 1022 formats (such as the "A" format in the first PRINT command or the literal in the second) will cause a LABEL to be written to the worksheet, whether the type of the 1022 expression being printed was text or not. The content of the label will be an ASCII string identical to that which 1022 would have produced while printing 'normally'. For example, consider the following command sequence:

```
*INIT 123 3 FOO.WKS.
*DEFINE INTEGER Q. LET Q 9999.
*PRINT ON 3 Q Q+1 FMT L I.
*RELEASE 3.
```

When loaded into 1-2-3, the spreadsheet will contain the binary integer 9999 in cell A1, while cell B1 will contain a label consisting of the DIGIT STRING '10000'. Since, presumably, most of the data that 1022 users will want to extract will be for computational purposes, they will therefore wish in most instances to use L format, which is the default.

3.1 Summary Of 1-2-3-Related INIT/PRINT Features

Our example has covered the basics of PRINTing to Lotus 1-2-3 files. Let us now summarize the points covered so far, and explore further options available under this file format:

3.1.1 L Format - is the default format used when PRINTing to a channel that has been INITed to a 1-2-3 file. The optional specification m.n following the L selects the Lotus format type and (if applicable) number of decimal places. L format MUST be used to derive computational spreadsheet data from 1022 numeric data.

3.1.2 Conventional Formats - such as I, F, E, A, D, etc. are used to produce labels. /, X, and \$ are among the means available for controlling the cell location of data. H format and literals also produce labels.

3.1.3 Data-Positioning Options - One requirement of Lotus worksheet format is that the cell coordinates of every data item be included in the data record. In order to do this, 1022 maintains three counters for each channel that has been initialized to a 1-2-3 file. These counters are available to user programs in the form of three system variables, indexed on channel number N.

3.1.3.1 Cell Location Counters SYSPCROW, SYSPCCOL, SYSPCICOL

* SYSPCROW(N) points to the spreadsheet row to which 1022 is currently printing or is about to

print. By default, SYSPCROW points to row 1 at INIT-time and is incremented upon completion of every PRINT command. "/" format may be used to increment SYSPCROW at any time. SYSPCROW is user-settable.

- * SYSPCCOL(N) points to the spreadsheet column to which 1022 is currently printing or is about to print. By default, SYSPCCOL points to column A at INIT-time. It is then incremented once for every cell produced in a given PRINT command. "nX" format adds n to SYSPCCOL, effectively skipping cells. Whenever a new-row action is triggered (as at the conclusion of a PRINT command or the execution of a "/" format), SYSPCCOL is reset to point to the column whose value is stored in a third PC-related variable, SYSPCICOL.
- * SYSPCICOL(N) (ICOL stands for Initial COLumn) points to the column at which each new row is to start. By default, SYSPCICOL points to column A at INIT-time.

SYSPCCOL and SYSPCICOL, like SYSPCROW, are user-settable.

NOTE that all the SYSPC variables are "zero - origin": that is, when the column and row counters are pointing to A1, they are both set to 0. To advance SYSPCROW to row 5 and SYSPCCOL to column D, therefore:

```
LET SYSPCROW(N) 4 SYSPCCOL(N) 3.
```

3.1.3.2 \$\$SYSPCPOS Function - As a convenience, the string function \$\$SYSPCPOS(N) has been added. \$\$SYSPCCOS(N) returns the ASCII representation of the cell coordinates to which SYSPCROW(N) and SYSPCCOL(N) currently point. An example:

```
*LET SYSPCROW(2) 5 SYSPCCOL(2) 4
*TYPE $$SYSPCPOS(2)
*E6
```

3.1.3.3 Controlling Data Position Via The SYSPC Variables

- FIGURE 10 shows an example of manipulating the position of data within a target spreadsheet by changing the values of the SYSPC variables.:

```
!program fragment showing placement of data in spreadsheet
!via alteration of SYSPCxxx variables.
!attributes AT1...AT5 are from GOO.DMS
!attributes VV...ZZ are from POO.DMS
!
OPEN GOO.DMS POO.DMS. FIND SYSID BET 1 10.
INIT 123 2 FOO.WKS.
!start at A1
!SYSPCROW(2),SYSPCICOL(2),SYSPCCOL(2) all=0
!fill COLS A-E of ROWS 1-10
PRINT ON 2 AT1 AT2 AT3 AT4 AT5.
DBS POO.DMS. FIND SYSID BET 1 10. !go to another ds
! right now, SYSPCROW=10,SYSPCCOL=0
LET SYSPCROW(2) 0. !reset to row 1
LET SYSPCICOL(2) 6. !slide over to col.#G
!NOTE!! SYSPCICOL is now pointing to COL. G. but if we
!do not also set SYSPCCOL, SYSPCCOL will be set to COL. A
!when the first record is printed below.
LET SYSPCCOL(2) SYSPCICOL(2).
PRINT ON 2 VV WW XX YY ZZ !fill cols G-K on rows 1-10
:
```

Fig. 10

3.1.3.4 Setting The SYSPC Variables At INIT-Time - Two new arguments have been provided in order to allow the user to set initial values of SYSPCROW, SYSPCCOL, SYSPCICOL prior to PRINTing data. The syntax is as follows:

```
INIT 123 [COL c] [ROW r]...
```

If COL is present, SYSPCICOL and SYSPCCOL are set to c. If ROW is present SYSPCROW is set to r. The defaults for c and r are A1, as mentioned previously.

3.1.3.5 Columnwise Vs. Rowwise Data Formatting - In all our examples thus far, data cells have been written in left-to-right fashion within each PRINT command, with the row counter advancing down the spreadsheet upon each new PRINT. This is termed "columnwise" representation in spreadsheet parlance, and is the default action. However, a "rowwise" representation is also possible when printing to a 1-2-3 file. We include an optional CWISE/RWISE clause in the INIT command for this purpose:

```
INIT 123 [COL c] [ROW r] { CWISE
                          } .....
                          RWISE
```

FIGURE 11 shows a DMC using these options. FIGURES 12A-12B show the spreadsheet results:

```
!this DMC prints the same data to two Lotus WKS files.
!the first files is INITed CWISE (by default)
!while the second is INITed RWISE.
INIT 123 2 EELS1.WKS.
INIT 123 RWISE 3 EELS2.WKS.
DEF TEXT 10 A B C D E F.
LET A "My" B "Hovercraft" C "is" D "full" E "of" F "eels." .
PRINT ON 2 A B C D E F.
PRINT ON 3 A B C D E F.
RELEASE.
```

Fig. 11

B1: 'Hovercraft' READY							
	A	B	C	D	E	F	G
1	My	Hovercraft	is	full	of	eels.	
2							
3							
4							
5							
6							
7							

Fig. 12A - CWISE

A1: 'My' READY							
	A	B	C	D	E	F	G
1	My						
2	Hovercraft						
3	is						
4	full						
5	of						
6	eels.						
7							

Fig. 12B - RWISE

As one might imagine, the underlying effect of the RWISE option is simply to switch counters; SYSPCROW is auto-incremented as individual items are printed, "X" formats executed etc. SYSPCCOL is auto-incremented on the end of each PRINT command, by "/" formats, etc. Programmers who want to change the counters would do well to remember this.

3.1.3.6 "\$" Format - "\$" format behaves as an analog to "\$"-format in normal printing: it disables automatic incrementing of SYSPCROW and

resetting of SYSPCCOL at the end of a PRINT command. This enables your program to (for example) print some data to a given row, "save its place" and do more calculation, and resume printing in the same row.

3.1.4 NAMED RANGE (NRANGE) Option - The final INIT option for 1-2-3 files is the ability to designate a block of cells within a worksheet as a Named Range:

```
INIT 123 NRANGE FRED B3 D10
```

uses the defaults for COL,ROW, and CWISE, and creates the Named Range FRED in the worksheet. One could use this with the File Combine feature, for example, to extract the subset FRED of the 1022 data into 1-2-3.

The default is no NRANGE present.

4.0 CREATING DIF FILES DIRECTLY FROM 1022

DIF is an ASCII format; therefore DIF files can be written from 1022 using normal formatted PRINTS. FIGURE 13 is an example:

```
!This DMC extracts 1022 records and fields and produces a DIF
!file. Note that the number of 'VECTORS' = 7 (6 attr's and one
!blank cell) and IWIDTH is set accordingly. The number of 'TUPLES'
!equals the number of selected records, + 1 for the 'tuple' of
!labels, + 1 for the blank 'tuple' and NTUPLES is set accordingly.
CLEAR.
OPEN MOBDEM.DMS RO.
F SYSID BET 1 10. SORT LN FN.
INIT 2 DEMDIF.DIF.
DEF TEXT 9 SKIPCELL TEXT 11 BOT EOD TEXT 2 CRLF TEXT 63 SKIPROW.
LET CRLF $CHAR(13)+$CHAR(10).
LET SKIPCELL "1,0"+CRLF+""+CRLF.
LET SKIPROW SKIPCELL+SKIPCELL+SKIPCELL+SKIPCELL+SKIPCELL -
+SKIPCELL+SKIPCELL.
LET BOT "-1,0"+CRLF+"BOT"+CRLF.
LET EOD $REPLACE("BOT","EOD",BOT).
LET IWIDTH 7. ! kludge to setup correct VECTORS item
LET NTUPLES SYSNREC+2. !kludge works as long as we print IWIDTH
!cells for SYSNREC+ (# of label rows and
!blank rows) records
!print the header section, note vectors and tuples counts.
PR ON 2 "TABLE" "0,1" "" "VECTORS" "0," IWIDTH -
FMT 4(G / ) 2G / "" END.
PR ON 2 "TUPLES" "0," NTUPLES FMT G / 2G / "" END.
PR ON 2 "DATA" "0,0" "" FMT 2(G / ) G END.
!print the data section. start with a row of text labels, with a
!blank cell in col. A (all rows will be like this)
PR ON 2 BOT "FIRST NAME" "LAST NAME" "# CHILDREN" "CITY" -
"STATE" SKIPCELL "ZIP" FMT G 5("1,0" / "" G "" / ) G "1,0" / -
"" G "" END.
!now print a row of blank cells
PR ON 2 BOT SKIPROW FMT G $ END.
!now print the stuff from the records
PR ON 2 BOT FN LN NCH CITY STATE SKIPCELL ZIP -
FMT G 2("1,0" / "" G "" / ) "0," G / -
"v" /2("1,0" / "" G "" / ) G "1,0" / "" G "" END.
PR ON 2 EOD FMT G $ END.
PR ON 2 $CHAR(26) FMT G $ END. !ctrl-Z eof mark
RELEASE 2.
TYPE "DEMDIF.DIF has been created on host and may be File".
TYPE "Translated into 1-2-3 format."
```

Fig. 13

Like our previous example (MAKPRN), this one suffers from intractably complicated FORMATS. Furthermore, there are two requirements imposed by DIF which our DMC does not really address:

1. The file must consist of a known number of "TUPLES" (we can consider them spreadsheet rows) and this number must be recorded in the file header;
2. Each "TUPLE" must be of equal length - i.e. each tuple must consist of the same number of "VECTORS" (we can consider them spreadsheet cells) and this number must be recorded in the

file header.

Our example meets these requirements only because it was constructed knowing the correct counts in advance. Obviously this is not usually the case.

A more general way to handle requirement (1) would be to have the program keep track of the number of lines(TUPLES) printed, and to write this count to the DIF file header when done. This is in fact what the 1022 DIF printing option does. The means of counting "TUPLES" is SYSPCROW, whose value is used for the count at RELEASE-time.

Requirement (2) is handled by assuming that each "TUPLE" will contain 100 cells ("VECTORS") unless the user says otherwise at INIT-time. The user specifies this via an optional NCOLS (Number of COLUMNS) clause in the INIT command. Given this NCOLS parameter, 1022 ensures that each row is of equal width.

Given a large enough NCOLS, then, a program need not worry about uniform length of its PRINT commands. If the correct NCOLS value can be known in advance at INIT-time, however, it can be used to advantage by avoiding padding and therefore saving file space.

4.1 INIT Syntax For Printing DIF

The full INIT syntax for DIF is:

```
INIT DIF [COL c] [ROW r] NCOLS n chan filespec
```

defaults: c=A, r=1, NCOLS=100, filespec extension=.DIF

4.2 DIF Example DMC

FIGURE 14 shows a DMC using the new INIT/PRINT features to produce a DIF file equivalent to the previous example. Note the use of "X" and "/" formats.

```
!This DMC extracts 1022 records and fields and produces
!a DIF file using V117B DIF printing features.
OPEN MOBDEM.DMS RO.
F SYSID BET 1 10. SORT LN FN.
INIT DIF 2 DEMDIF.
!INIT wrote the header... no fuss, no muss.
!Note use of X and / fmts in next command
PRINT ON 2 "FIRST NAME" "LAST NAME" "# CHILDREN" "CITY" -
"STATE" "ZIP" FMT G 1X G / END.
!now print the data from the records
PR ON 2 FN LN NCH CITY STATE ZIP FMT 5G 1X G END.
RELEASE 2.
TYPE "DEMDIF.DIF has been created on host nd may be File".
TYPE "Translated into 1-2-3 format."
```

Fig. 14

"X" and "/" (as well as the COL and ROW options) are functionally identical to their usage in printing 1-2-3 files. However, they actually work by writing out "padding" (blank cells).

FIGURE 15 is the result of using the Lotus File Translate utility to derive a WKS file from the DIF file produced by the program of FIG. 14, and loading the WKS file into 1-2-3:

FIRST NAME	LAST NAME	# CHILDREN	CITY	STATE	ZIP
CHARLES	CARAGIANES	3	DEDHAM	NY	02138
RICH	GARLAND	4	BRISTOL	CT	02138
CHARLES	GOTT	2	BRISTOL	CT	22209
KATHY	HOUSMAN	0	AUGUSTA	GA	43220
MARK	JONES	0	DEDHAM	NY	60064
ROGER	LEVINSON	3	BRISTOL	CT	11729
LOUIS	MERZ	0	ROXBURY	TX	77056
OLGA	PONG	3	STONEHAM	MA	02238
ALFRED	SAVIO	1	ROXBURY	IN	46225
ALFRED	STEVENS	0	BRISTOL	CT	02238

Fig. 15

4.3 1022 --> DIF Data Transformations

DIF format recognizes only two datatypes: text and numeric. Numeric data is represented by ASCII digit strings. The rules for data transformation and formatting are quite simple:

1. The type of the 1022 item being printed determines the DIF datatype:

1022 type	DIF type
Integer	Numeric
Double Integer	Numeric
Real	Numeric
Date	Text
Text	Text

(Note the date --> text transformation. If a computational date is desired, \$INT(date-item) should be printed. This does not guarantee that the resultant number will be correct when read into the target spreadsheet or other program, which is likely to represent dates differently than 1022. The \$INT result may have to be offset by an amount that will cover the difference.)

2. Conventional 1022 formats are employed when printing to DIF files; there is no DIF equivalent to "L" format. The actual text or digit string that results is the same as that which would normally be produced under a given format spec. It is the programmer's responsibility to ensure that such a result is appropriate for the spreadsheet or other program which is to receive the DIF file. To mention a fairly obvious example, PRINTing an integer under "O" (octal) format would produce a numeric DIF item which would be interpreted as a string of decimal digits by any program that adheres to the DIF standard.

Using Mobius to Extend 1022 and 1032 Capabilities to Personal Computers

Chris Kayes, FEL Computing

Mobius is a micro/host integration package that extends the capabilities of Software House's 1022 or 1032 systems to personal computers. The personal computer user can now extract data from a large central data base and process it using the vast array of readily available personal computer software. This can all be accomplished without ever leaving the familiar environment of the microcomputer. In addition, host system capabilities, such as its mail system, are now directly available to the microcomputer user.

This paper shows how the 1022 system interacts with Mobius to provide a smooth interface between the host and micro computers. While the examples given are for 1022, the principles and most of the details are identical for 1032. How Mobius meets the varying needs of the microcomputer end user, host computer user, programmer, and information manager will also be addressed. More complete information and technical details about Mobius may be obtained by contacting the author.

1.0 MOBIUS, 1022, AND THE MICROCOMPUTER END USER

Mobius allows the microcomputer user to access host programs, data, and other resources (such as printers) in exactly the same way that the micro's own programs, data, and resources are accessed. Thus, the end user only needs to master one computing environment - that of the microcomputer. Mobius handles access to host resources completely transparently, so that the user can be totally unaware of where the programs and/or data actually reside. Therefore, the user is left to concentrate on the task to be accomplished, unencumbered by difficult and error-prone file transfer and communication tasks. This results in more efficient use of the person's time, both because no additional training is required and because the operations being performed are handled smoothly and easily.

1.1 An Example

An example will illustrate how easily the micro-to-host interaction becomes to the end user. Here, the host 1022 data base management system is used to extract information from a central data base, and then that information is loaded into a Lotus 1-2-3 spreadsheet on the micro. With Mobius, this task is performed completely on the user's microcomputer with the following sequence of commands:

```
(1)  A>1022
(2)  *USE MAKWKS
(3)  *QUIT
(4)  A>123
(5)  /File Retrieve D:DEMO3
```

Line (1) of this example shows the "A>" prompt displayed on the user's personal computer (IBM-PC, Rainbow, etc.). The user now wants to run the 1022 system which is written for and runs on the host machine, so he enters the "1022" command to the microcomputer's prompt. Notice that this is exactly how the user would start a program that was written for and runs on the microcomputer. As far as this user is concerned, he is simply running a program; he does not know or need to know where it actually resides.

Between lines (1) and (2) of the example, Mobius operates invisibly so as to make the host program access completely transparent to the user. First, Mobius starts the host 1022 program and then it automatically causes the microcomputer to operate as a VT-100 terminal. Thus, when 1022 outputs its "*" prompt, it appears on the screen just as would the prompt from any microcomputer program.

Lines (2) and (3) are commands which the user enters into the 1022 system. These commands can be as simple or elaborate as the application requires, and all features of the host 1022 system can be utilized. In this example, MAKWKS is a program that

extracts data from a 1022 data base file and outputs a file in a format that can be read by the Lotus 1-2-3 spreadsheet program which runs on the user's microcomputer. The file itself is stored in a directory on the host computer, but the user need not be concerned about this. All the user in this example needs to know is that when the MAKWKS program is run, it produces a file called "DEMO3" on the microcomputer's "D" drive which is internal to his machine and which he can't actually see. In fact, later on, Mobius will perform the appropriate tasks, invisibly to the user, which cause this file to be retrieved when the "D" drive is referenced.

Between lines (3) and (4), Mobius again operates invisibly. First, it detects that the host 1022 program has terminated; then it causes the microcomputer to operate as it normally does, instead of as a VT-100 terminal; and finally, it causes the micro's "A>" prompt to again appear.

Now, when the Lotus 1-2-3 program is started (line 4), all that the user needs to do is to retrieve the file D:DEMO3 that was created by the 1022 system, just as any other file would be retrieved with 1-2-3 (line 5). Again, Mobius operates invisibly to retrieve the file from the host system and to make it available to the 1-2-3 program.

1.2 An Even Simpler Example

The above example illustrates how Mobius operates to provide truly integrated micro/host interaction. The entire process can be even further simplified by using still other features of Mobius. For example, for users who do not know how to use 1022, but still have a need to access its data, Mobius provides a facility where lines (1) through (3) of the example can be combined into what appears to the user simply as a microcomputer program. If we call this program "GETWKS", then the following user commands to the micro perform the same function as the previous example:

```
(6)      A>GETWKS
(7)      A>123
(8)      /File Retrieve D:DEMO3
```

In this example, the user is freed from needing to know anything about the host 1022 system. This is particularly useful in the somewhat common situation where the user wishes the same type of updated data on a regular basis.

Since Mobius is completely integrated into the microcomputer's operating system, its "batch" facility can be used to even further simplify the action required by the user. For example, if lines (6) and (7) are combined into a batch file called "START123", then the entire process of accessing the host, starting the 1022 system, extracting data from the data base, outputting the extracted data into 1-2-3 file format, and loading that data into a 1-2-3 spreadsheet can be performed with only two microcomputer commands:

```
(9)      A>START123
(10)     /File Retrieve D:DEMO3
```

Note that in these examples, Mobius has worked completely invisibly and has not required the user to deviate from normal microcomputer procedures in any way.

1.3 Additional Versatility

The above is only one illustration of how Mobius allows end users to access host resources without needing to know any of the details of the host system. While the examples used the 1022 and 1-2-3 programs, they are equally valid for any host and/or micro program or combination of them. For instance, the host MAIL program can be run just as conveniently as 1022 was run in the example, thus providing the micro user with access to all of the features of the host mail facility as if that facility resided on the micro.

As another example, a microcomputer text editor, such as WordStar, can be used to edit files that have been created by a host program. In this case, Mobius allows the host file to be read directly into WordStar, eliminating the need to perform any complex file transfer tasks.

As can be seen by all of these examples, the integrated applications which Mobius makes available are virtually unlimited, since every host program can now be run as if it were on the micro, and every micro program can directly access host data and other resources. Mobius imposes no constraints on these whatsoever, thus eliminating user retraining and preserving current investments in software.

2.0 HOW MOBIUS SUPPORTS THE HOST USER WHO HAS A MICROCOMPUTER

While Mobius allows the microcomputer end user complete transparency when accessing a host machine, such transparency may not always be desired by a person who is familiar with the use of the host. Also, this type of person is most likely to be setting up applications for end users, and therefore needs a mechanism to accomplish this quickly and conveniently.

2.1 Switching Between the Micro and Host

Switching directly between the microcomputer and host environments can be accomplished in a variety of ways with Mobius. The way most familiar to most host users is simply to type the following microcomputer commands:

```
A>PUSH      (if the host is TOPS-20)
A>SPAWN     (if the host is VMS)
```

When this is done, Mobius invisibly starts a new host process and causes the microcomputer to operate as a VT-100 terminal. At this point, any host program or function can be performed, such as editing a file, reading mail, running a data base system, etc. When Mobius detects that the process is terminated (ie: the user enters "POP" on TOPS-20, "LOGO" on VMS), it causes the microcomputer to operate as it normally does, instead of as a VT-100 terminal, and then to display the "A>" prompt again.

2.2 Configuring the Micro/Host Environment

Rather than requiring direct access to the host operating system functions as above, the user may wish instead to access that portion of the Mobius system itself which resides on the host machine. It is this portion of Mobius that contains an easy-to-use set of commands which allow the user to configure the Mobius environment, as was necessary

for the 1022 example given above. To do this, the user simply enters a single keyboard character (initially defined as "CONTROL-A", but resettable by the user). Then, the current activity taking place on the microcomputer is instantly suspended (so that it can be resumed later), the host Mobius system is activated, and its "MOBIUS>>" prompt is displayed. At this point, Mobius is waiting for a command to be entered by the user.

The host Mobius commands provide a tremendous amount of convenience and capability for setting up applications as well as for performing useful host functions. In the 1022 example above, a file called "DEMO3" was written to a host directory, and that file was seen by the microcomputer to reside on its disk drive "D:". This relationship between host and micro resources is established using the Mobius "DEFINE" command. For instance, the command

```
MOBIUS>>DEFINE (micro device) D: (to be) Host  
(resource) *.*
```

tells Mobius that whenever the microcomputer's "D:" device is referenced (such as was done with Lotus 1-2-3 in the example), the files of the currently accessed directory (as specified by the "*.*") are to be accessed. Thus, as an additional example, the microcomputer command

```
A>DIR D:
```

will display all of the files on the user's currently accessed host directory. If the list of files specified to the DEFINE command had been "*.DOC,*.MEM", then only those files with "DOC" and "MEM" extensions would be listed. Similarly, if "SYS:" had been specified, then all of the files associated with that logical name would be displayed, no matter how many directories that represents.

The DEFINE command can also be used to specify that output normally destined for the microcomputer's printer will instead be routed to a host device. For example,

```
MOBIUS>>DEFINE (micro device) PRN: (to be) HOST  
(resource) PRINTR.OUT
```

would route all microcomputer printer output to the host file "PRINTR.OUT". This output could have just as easily been routed to a host line printer or other device.

The "DEFINE" command is only one of about thirty commands that the host Mobius system provides. Some of the other commands replicate host system commands such as "COPY", "DELETE", "RENAME", etc., so that these functions can be performed easily and without leaving the Mobius system. Others allow for the tailoring of the Mobius environment for individual user's needs, such as changing the "CONTROL-A" character mentioned above, specifying the amount and type of information given when help is requested, and outputting specific application-oriented information. Still other commands allow setting the parameters of the communication channel or showing the status of the Mobius environment.

Once it has been determined how the Mobius environment is to be configured, all of the necessary host commands can be put into a data file. This file is then read when host Mobius is started and each command is executed, just as if it had been entered from the keyboard. Thus, the entire micro/host environment can be set up automatically and invisibly to the user.

3.0 MOBIUS AND THE PROGRAMMER

For most organizations, Mobius provides all necessary micro-to-host integration functions without requiring any special programming whatsoever. However, for those organizations which wish to create specialized distributed applications, Mobius simplifies the process by providing an Advanced Programmer's Interface (API). The API is designed to give programmers direct access to the Mobius features that are available to the user at the microcomputer keyboard. For example, the user activates the VT-100 terminal emulator by typing a special character. Similarly, a program can activate the terminal emulator by using a Mobius API "system call".

The Advanced Programmer's Interface appears to the programmer as an extension of the micro's operating system. As such, it gives the programmer access to several new system calls which are utilized in exactly the same way as normal system calls are utilized. Any programming language which can make calls to the microcomputer's operating system (which is virtually all of them) can call upon Mobius to perform its micro-to-host integration tasks. Thus, the API allows end-user organizations and OEMs to create sophisticated distributed applications without requiring systems programmers or communication specialists.

4.0 MOBIUS AND THE INFORMATION MANAGER

The previous sections have shown some features of Mobius as they related to particular types of host and/or personal computer users. To the Information Manager, though, Mobius is more than a set of technical features and capabilities. Rather, it is a single unified solution to the problems created by a diverse set of micro/host users, using a variety of programs and machine types. The inherent versatility of Mobius is illustrated in the previous section by the ease of use for the microcomputer end user, host system user, and programmer alike. Mobius provides each class of user with the same environment they are already used to, thus increasing their productivity and minimizing (even eliminating) the need for retraining. Each class of user is also provided with easy access to the rich set of features that are available to the other classes of users, should they wish to take advantage of them.

This versatility is complemented by close attention to the needs of managing host data and security. Mobius provides this through a combination of host file access mechanisms and special Host Mobius features. A key element of Mobius is that first-level access security is not controlled at the microcomputer, which is the most vulnerable part of a micro-to-host system, or even by Mobius itself; but rather it is controlled through the host operating system.

5.0 HOST-BASED ACCESS PROTECTION

The Host Mobius program operates as a normal user program running under the host operating system. Therefore, Mobius can provide the microcomputer user with no more file access than that user would have if accessing the host from a normal computer terminal. This design was chosen over a "server" or "privileged program" concept because it allows easy custom tailoring to individual users without introducing security problems.

Some of the advantages of this design are:

1. The host system manager needs to establish directory and file access privileges only once. There is no additional mechanism needed to provide protection for microcomputer users.
2. No passwords can be entered by the user when running a program from the microcomputer, nor can any passwords be accidentally displayed.
3. Because of (2), it is useless to enter passwords into data files stored on the microcomputer since they can not be functional from such files. Storing passwords in such files is one of the most common areas of security breach.
4. The microcomputer user has full access to those host files normally available to that user. No additional procedures must be learned to access them.
5. The host system manager remains in full control of the access and integrity of the host system files.

6.0 MOBIUS-BASED ACCESS PROTECTION

By design, Mobius can not allow access to host files beyond what is allowed by the host operating system. However, it can further restrict such access. For example, if the host system allows a user to read and write all files in a particular directory, Mobius can be set to allow reading only those files written by the user during the current Mobius session.

Mobius also provides the ability to mark sets of host files as "read only". This is accomplished with the Host Mobius "LOCK" command, which not only prevents writing to files that already exist, but also prevents new host files from being created.

It is also possible to prevent the user from accessing the host except through Mobius and/or to issue any Host Mobius commands. Thus, the micro-to-host environment can be set up so that the user will never be able to change it, but all required host resources will still be available to the microcomputer user.

7.0 INTEGRATING PCS AND HOSTS

Mobius is a system which fully integrates personal computers with host machines. While traditional file transfer and terminal emulation capabilities are built into Mobius, these only scratch the surface of the tremendous versatility available to the user and/or system integrator.

The example illustrating the use of the 1022 data base system with the 1-2-3 spreadsheet shows that Mobius supplies direct access to an organization's data and facilities from a personal computer, while maintaining all of the flexibility that these machines offer the user. Also, by offloading tasks to the personal computer, host performance and user productivity is increased.

The versatility of Mobius is further enhanced by the wide variety of machines on which it is implemented, including VAX, DECsystem-10, and DECsystem-20 host computers and PC-DOS (IBM-PC and compatibles), MS-DOS, and CP/M microcomputers. This range of machines allows integration to take place not just between PCs and hosts, but between dissimilar hosts and microcomputers as well.

All of this adds up to an unusually flexible system for the 1022/1032 user. First, Mobius allows the capabilities of these systems to be immediately extended to the microcomputer user. Then, extracted host data can be used in 1-2-3, dBase, and other microcomputer programs. As the user's needs grow, additional host systems can also be extended to the micro. What may initially be viewed as an adjunct to the 1022 or 1032 system, in fact provides general-purpose capabilities that can be used to integrate virtually any micro/host application.

Kathy Rosenbluh
 Digital Equipment Corporation
 Marlboro, Massachusetts

ABSTRACT

This session covered VMS user interfaces and concentrated on the functionality that end users need as they move from TOPS to VMS. Areas discussed included definition of a user, logging on, directories, processes, queues, batch and spooling systems, file access protection, file manipulation, device allocation and mounting, system services, program development, mail, editors, networks, command procedures, symbols and logicals, lexical functions, error handling, DCL, inter process communication, linker images and debuggers.

Kathy indicated that both systems managers and programmers could use the information in this presentation. She said that the session could be used as a guide for a short course as you move to VMS from TOPS. Following are the slides presented during this presentation. More information has been interspersed in the original slides to cover additional items discussed during the session.

LOGGING ON VMS

- . When logging on, you get a username prompt, unlike on TOPS-10 and TOPS-20
- . You can implement a system password feature
 - Requires a user to enter a system password before getting announcement of the system you are on
 - : Especially useful for dial-up lines when people are trying to break into a system
- . Initially prompt for a username if not using a system password

USER

- . An entry in the SYSUAF file creates a User. It contains such things as: account strings, UIC, login name, password, quotes, privileges
- . Entry also specifies the login directory

Username	UIC
-----	---
Text string	Octal number
Used for login only	For everything else
	Unique for each user

- . The login directory can be a subdirectory (defined by the system manager), but the username does not therefore become a sub-username
- . EXAMPLE:
 UAF> add SMITH /DIRECTORY=1 [PROJECT1.SMITH]
 /QUOTA=4000 /OVERDRAFT=1000 /CLI=xxx

DIRECTORIES

- . QUOTA and OVERDRAFT QUOTA, on a per-structure basis

- . File is charged to File Owner (can be different from Directory Owner)
- . Directory owner may create subdirectories (unlike TOPS)
 - Example:
 CREATE/DIRECTORY DISKA:[PROJECT1.SMITH]
 /OWNER_UIC:xxx/PROTECTION=(S:RWED,O:RWED,G:RE,W:)

DIRECTORIES

- . \$ SET DEFAULT device:[directory]
 - Changes "connected" directory
 - No limitation to number of directories to which a user can be connected
- . No access checking is done for SET DEFAULT
 - File access checking is done when file operation is attempted
- . Search lists:
 - Implemented through logical name definitions
 - \$ DEFINE DSK [dir1],[dir2],[dir3]
 - \$ SET DEFAULT DSK

A PROCESS

- . Process = Context + Executable Image
- . Has one 32-bit physical address space
- . Has 4 30-bit virtual address spaces
- . Contains current image in P0
- . Contains stacks, I/O database, quota and privilege information, logical name tables, PSL, etc. in P1
- . Contains system space, shared by all processes in S0
- . The last space is not yet used but it is reserved for Digital

A PROCESS CAN

- . Execute images or run programs
- . Execute DCL commands
- . Execute procedures
- . Spawn another process

GETTING PROCESS/SYSTEM INFO

- . SHOW STATUS

- Show, e.g., working set size, quota used, clock and system time used for current process
- . SHOW SYSTEM
 - Show version of VMS being run
 - One line message for every process running
- . SHOW PROCESS
 - More detail than SHOW SYSTEM
 - Describes a users current process in full
 - Privileges enabled, resource information on process are shown
- . SHOW NETWORK
 - IF DECNET is running, show physical connection, whether end or routing node, and a list of possible connections if it is a routing node

PRINTING & BATCH QUEUES

- . Getting Information: (Information is clusterwide)
- . SHOW QUEUE/DEVICE
 - Shows status of all queues on the system
 - Shows all characteristics of the queues if you use the /FULL switch
 - Default is to show only queue entries matching the user name of the logged in process. To see all entries in the queue, use the /ALL switch
- . SHOW QUEUE
 - Shows status of output queues
- . SHOW QUEUE BATCH
 - Shows status of batch queues

QUEUES

- . Queue manipulation commands are available to users
- . SUBMIT filename(s)
 - <==== to batch queue
- . PRINT filename(s)
 - <===== to output print queue
- . MODIFY jobname(s)
 - <===== to modify a queue entry once submitted
- . SYNCHRONIZE entry
 - allows two batch jobs to be interdependent; This command is included in the second batch job
- . DELETE / entry
 - To cancel/delete a queue entry

BATCH & SPOOLING SYSTEMS

- . Queue entries can be moved between queues by operator
- . Typical queue characteristics; not user settable
 - JOB_LIMIT, BASE_PRIORITY, RESTART_VALUE, SPOOLED, FORM, CHARACTERISTIC, BLOCK_LIMIT
- . Default values for flag, trailer & separation pages and bursting are set by operator, but may be overridden by user queue entry
- . Operator chooses position-in-file when print queue is restarted
- . Batch jobs can be restarted from point of completion

BATCH & SPOOLING SYSTEMS

- . Print and batch queues can be cluster-wide
- . Users can load forms for programmable printers

- . Generic queues
 - Output: printer and terminal queues can be assigned to a generic queue
 - jobs in generic queues go to first available device assigned to the generic queue
- Batch: used in clustered systems

FILE ACCESS

- . This is a description of the UIC based protection available for files on VMS
 - For files, directories, queues, etc.
- . Divide world into 4 groups:
 - Group defined as members of the same UIC group number
 - Owner, System, Group, World
- . Each group can have any/none/all of 4 kinds of access:
 - Read, Write, Execute, Delete
 - The user sets these up for each file
 - Defaults are set up for directories so these do not have to be set up for each file written

FILE ACCESS LISTS

- . New to VMS V.4
- . Can include/exclude access by identifier e.g., UIC, , type-of-connection
- . Utilities exist to create/modify access lists and identifiers
- . List is stored in file header
- . Fine-tuned control of file access is possible at cost of slight cpu overhead

OTHER FILE FUNCTIONS

- . A full file specification looks like this
 - node"username password account":device [directory]filename.ext;n
- . Standard format for APPEND, COPY, TYPE
 - \$ copy oldfile-spec newfile-spec
 - \$ copy
 - From file: oldfile-spec
 - To file: newfile-spec
 - Can be done on same line or can prompt for it
- . Wildcards can be used

FILE MANIPULATION

- . No UNDELETE command is available
- . \$ DELETE file.ext:n
- .
- \$ DELETE /BY_OWNER can look for matching UIC anywhere on a disk
- . \$ PURGE file.ext
 - Deletes all but the last generation (by default) or a specified number of generations
- . \$ SET file/ENTER = alternate name allows multiple directory entries for one file
 - Be careful with this. Deleting one name deletes all alternate names in a directory

MORE FILE MANIPULATION

- . File attributes are stored in file header
- . Include information about data format, recording

- mode, etc. (Can see these with the DIRECTORY/
FULL command.)
- . ANALYZE/RMS_FILE shows internal structure of file
 - . CONVERT command used to set file attributes and structure
 - . SET FILE/ERASE_ON_DELETE for better security
 - Costs in extra overhead so you may not want this for every file in the system

DEVICE ALLOCATION -- DISK

- . To see detailed information about device characteristics:
 - \$ SHOW DEVICES /FULL DMAO
- . \$ ALLOCATE device-name logical-name
[DM] [MYDISK]
[MYDISK] [PROJECT1]
- . \$ INITIALIZE device-name volume-label
 - Done once only. Wipes out current data on disk
 - On non-blank disk requires VOLPRO, or same UIC as disk-owner
- . \$ MOUNT device-name, volumn-label, logical-name
[MYDISK] [PROJECT1] [P1]
- . Can use generic define name

MOUNTING TAPES

- . \$ ALLOCATE device-name logical-name
[MTA1:] [MYTAPE]
[MYTAPE] [PROJECT1]
- . \$ INITIALIZE logical-name label
 - This writes a volume label, headers, BOT, EOT, EOF and EOV
- . \$ MOUNT device-name label logical-name
[MTA:1] [PROJECT1] [P1]

MOUNTING TAPES

- . Switches to MOUNT
 - BLOCKSIZE, LABEL (type)
 - OVERRIDE (access checks)
 - RECORDSIZE can be specified
 - FOREIGN
- . User must DISMOUNT and DEALLOCATE device

PROGRAM DEVELOPMENT

- . Native-Mode VMS Languages:
 - FORTRAN, COBOL, BASIC, PL/1, RPG, Pascal, MACRO, ADA
 - Can all call one another
- . No explicit Compile command (as in TOPS) or Save command
 - \$ FORTRAN FILE-1.FOR, FILE-2.FOR, ...
 - \$ COBOL FILE-3.COB, FILE-4.COB, ...
 - \$ LINK FILE-1,FILE-2,FILE-3,FILE-4, ...

SYSTEM SERVICES

- . From two basic sources
 - Common Run-Time Procedure Library
 - : Used for file activity for RMS, e.g.
 - Base System Services - Information, e.g., on the state of the world, state of the process or outstanding asynchronous interrupts
- . Analogous to UUO's and JSYS's
- . Can be used directly for native-mode VMS languages

LINKER

- . Invoked with "\$ LINK ..." command
- . Creates executable and shareable images
- . Linker can find and include library information
 - object modules, macros, help text, record descriptors

MAIL

- . VMSMail
 - Available by default to all users with VMS
 - Subset of TOPS MS features
 - Can call any editor
 - Can store mail in folders

MAIL

- . DECMail
 - Has its own editor
 - Users must be explicitly made known to DECMail
 - Two-step send
 - Can store mailing lists
 - Has storage folder management functions
 - Menu or command mode

EDITORS

- . SOS --line oriented editor
- . EDT --line mode
 - screen mode with keypad functionality
 - screen mode with typed-in commands (now also available on TOPS)
- . EMACS-32 --available from third-party
- . TECO/TV } Both available on Integration
- . SED } Tools Tape, unsupported
- . TPU --- Programmable editing utility

NETWORK ACCESS

- . Same network access as on TOPS
- . Virtual terminal support through
 - \$ SET HOST hostname
- . Proxy logins allow remote file access without specifying password/username
- . Network file operations are transparent -- just include nodename in file specification
- . Batch jobs can be executed on remote DECnetted nodes at user request

SPAWNED PROCESSES

- . Can execute single line of DCL without attaching
- . Can DISCONNECT virtual terminal from process
 - CONNECT to relink to physical terminal
- . Can ATTACH to move among processes
- . For speedier spawn, specify /NOLOGICAL /NOKEYPAD /NOSYMBOL
- . Can run images detached

COMMAND PROCEDURES

- . Procedures can be executed interactively, can be nested up to 16 levels deep, or can be submitted as a batch job; or a sub process can be spawned, and a procedure can be submitted from that sub process
- . Interactive: \$ @procedure-file
OR define a command-name synonym
 - \$ task1 = "@procedure-file"
 - \$ task1

- . Batch: \$ submit procedure-file

PROCEDURE DEBUGGING

- . SET VERIFY will display lines of DCL and/or image i/o while procedure executes
- . Reset SET MESSAGE to show full error strings
- . Look for contradictory logical and symbol definitions
- . Do SET PROCESS /DUMP and ANALYZE /PROCESS_DUMP

COMMAND PROCEDURES

- . After each login (Batch or interactive) 3 commands files are automatically executed (if they exist):
- . System "GROUP" LOGIN.COM in user's directory
- . Procedures can be nested up to 16 levels deep

PROCEDURE I/O

- . Procedures can contain DCL commands and data lines. Can have data and flow control commands
- . Procedures are interpreted and not compiled
- . INQUIRE statement will ask for input from terminal
- . Up to 8 parameters can be passed to a procedure
- . You can redirect input and output to files
- . Can specify number of seconds to wait for input with READ/TIME_OUT = n

EXAMPLE 1 --- PROCEDURE I/O

```
$ if 'p1' . nes. "" then goto do_it
$ if f$mode () .nes. "INTERACTIVE" then exit
$ inquire p1 "Directory name"
$ do_it
$ set default 'p1'
$ directory /size /date
$ run proggy.exe
option 3
$ define/user_mode sys$input message.txt
$ mail
$ exit
```

EXAMPLE 1 EXECUTION

- . Interactive submission
 - \$ @exam1 -OR-
- . Interactive define symbol
 - \$ exam1 rosenbluh.project1 /output dirfil.txt -OR-
- . Batch submission
 - \$ submit exam1 /parameters=(rosenbluh.project1)

SYMBOLS & LOGICALS

- . Symbols can stand for CHARACTER STRINGS or INTEGER VALUES
- . Both kinds of symbols can be used in expressions
- . Logical names are equated to a character string, usually representing file specifications, directories and devices

SYMBOLS & LOGICALS

- . Logical names are expected by VMS in file manipulating commands

- . Symbol names are expected by VMS
 - On the right side of an = assignment statement
 - In arguments to lexical functions
 - In IF, WRITE, DEPOSIT and EXAMINE commands
 - Wherever else symbol substitution is requested with an operator

SYMBOLS EXAMPLES

PLAIN SYMBOL ASSIGNMENTS

- . Can assign integers to the symbol name count
 - \$ count = 123 \$ octal count = %123
- . 2 ways to specify character streams
 - \$ infile = "magic.dat" \$prompt := Hello all
- . Can define a file specification to a symbol
 - \$ symdat = "disk2:[develop.data]"

EXPRESSIONS USING SYMBOLS

- . Can do arithmetic manipulation with symbols
 - \$ count = count + 1
- . Can do character manipulation
 - \$ infile .eqs. infile - ".dat"
- . \$ sum = f\$length(infile) + 1

LOGICALS EXAMPLES

- . Define is used to set up a logical; can concatenate files
 - \$ define logdat disk2:[devel.data],user1:[prod.data]
- . \$ directory logdat:*.dat
- . \$ directory 'symdat':*.dat
- . \$ if symdat .nes. "" then filspec = symdat + infile

LEXICAL FUNCTIONS

- . Can be used interactively or inside command procedure
- . Return system and process environment information (interactive or batch, network or local, e.g.)
- . Do string manipulation
- . Do data type manipulation
- . Parse file specs, get file attributes and device information
- . Translate logical names

LEXICAL FUNCTIONS EXAMPLES

- . f\$environment will tell you what the current directory is
 - \$ dirnam = f\$environment("default")
- \$ set default user1:[project1]
- \$...
- \$ set default 'dirname'

LEXICAL FUNCTIONS EXAMPLES

- . \$ filspec = f\$parse(filspec,"*.*;*")
- . \$ filspec = f\$search(filspec)
- . \$ dirspeg = f\$parse(filspec,,,"DEVICE")- +f\$parse(filspec,,,"DIRECTORY")

. \$ filnam = f\$parse(filspec,,,"NAME")-
+f\$parse(filspec,,,"TYPE")

. \$ write sys\$output dirspec+filnam

CONTROL STRUCTURES

- . Loops are implemented with:
 - IF true (expression) THEN [\$] command
 - GOTO label
 - GOSUB for subroutines
- . Case statements implemented with:
 - \$ command list = "EXIT/HELP/OPTION1/OPTION2"
 - \$ inquire command(command_list+>)
 - \$ if f\$locate(command+"/",command_list) .eq.-
f\$length(command_list) then goto error
 - goto 'command'
 - option1:

ERROR HANDLING

- . Error handling is up to the person writing the procedure
- . Global symbol \$STATUS contains last-error information:
 - severity level, message number, who generated, flags
- . Global symbol \$SEVERITY - serverity level only
- . By default, error or severe error result in an EXIT command
- . Turn on error handling with \$ SET ON, disable with \$ SET NOON

ERROR HANDLING

- . Control Commands:
 - \$ ON WARNING THEN command
 - \$ ON ERROR THEN command
 - \$ ON SEVERE_ERROR THEN command
 - \$ ON CONTROL_Y THEN command
 - Can SET VERIFY to do command procedure debugging

SOME DCL LIMITS

- . Limits increased in Version 4
- . Command buffer size is 2048 characters
- . Command line limit is 255 characters
- . File name and extension can each be 39 characters
- . Version numbers go up to 32767

DCL -- KEYPAD DEFINITIONS

- . You can define certain terminal keyboard keys to a character string
- . Example: \$ SET TERMINAL/APPLICATION_KEYPAD
\$ DEFINE/KEY PF1 "SHOW"/SET_STATE=DUBBLE/
NOTERM/NOECHO
\$ DEFINE/KEY PF1 "DEFAULT" /TERMINATE/IF_State
=DUBBLE/ECHO
- . Then pressing PF1 twice will do:
\$ SHOW DEFAULT
- . Key states can be explicitly defined,
e.g: \$ SET KEY /STATE=DUBBLE
- . \$ SHOW KEY displays settings and states

COMMAND LINE EDITTING

- . Up to 20 previous lines of DCL commands can be recalled to correct lines

- . Each line can be edited, using control characters, and then re-executed

COMMAND LINE EDITTING

- . Example: \$ show default
\$ directory *.mem
\$ set default [project1.review]
\$ recall/all
1 set default [project1.review]
2 directory *.mem
3 show default
\$ recall 2
\$ directory *.mem

COMMAND LANGUAGE INTERPRETER

- . Ways to tailor the command language:
 - SET CLI allows you to define which command-line-interpreter will be used by your process
 - : Possibilities:
 - DCL
 - DECShell
 - User-written
- . You can write a program to do a function, and define a foreign command which runs that program
- . With the Command Definition Utility, you can add commands to standard DCL

TERMINAL ENVIRONMENT

- . SET HOST 0 /LOG to create terminal log
- . SET MESSAGE controls amount of feedback
- . SET BROADCAST controls reception of information from others (e.g., Mail, line detects from other terminals)
- . SET PROMPT (to . or @ or anything)

COMMUNICATION WITH USERS

- . MAIL
- . PHONE -- split screen tty linker allows logging of session and can show directory of users on remote node
- . REQUEST sends line of text to operator terminal

INTERPROCESS COMMUNICATION

- . Via shared files
- . Via decnet
- . Via mailboxes
- . Via common event flags
- . Via resource locks
- . Via asynchronous software trap

IMAGES

- . Shareable images
 - Must be INSTALLED in memory
 - Use only one copy of page in physical memory at run time so reduce physical memory requirements
- . Executable image is saved in a disk file by the Linker
 - Can call Shareable image
 - Can be INSTALLED



Kathy Rosenbluh
 Digital Equipment Corporation
 Marlboro, Massachusetts

ABSTRACT

The purpose of this session was to cover the basics of programming under VMS so that TOPS users could understand what is involved in the transition to VMS. The areas covered included processes, inter-process communication, system services, program development, compilers, object libraries, runtime libraries, the Linker, global symbols, shareable images, privileges, monitoring running programs, debugging, breakpoints, editors, system routines, intra-process communication and the file system.

Presented below are the slides used during this presentation. Comments brought up during the session are sometimes interspersed in the slides.

A PROCESS

- . Process = Context + Executable Image.
- . A process is what is established for a user when he logs in.
- . Has one 32-bit physical address space.
- . Has 4 30-bit virtual address spaces.
- . Contains current image in P0.
- . Contains stacks, I/O database, quota and privilege information, logical name tables, PSL, etc. in P1.
- . Contains system space, shared by all processes in S0.

SYSTEM SERVICES

- . From
 - Common Run-Time Procedure Library.
 - Base System Services.
- . Analogous to UUO's and JSYS's.
- . Can be used directly from native-mode VMS languages.

PROGRAM DEVELOPMENT

- . Native-Mode VMS Languages:
 - FORTRAN, COBOL, BASIC, PL/1, ADA, C, RPG, Pascal, MACRO, BLISS-32.
 - Can all call one another.
- . No explicit Compile command or Save command
 - \$ FORTRAN FILE_1.FOR, FILE_2.FOR, ...
 - \$ COBOL FILE_3.COB, FILE_4.COB, ...
 - \$ LINK FILE_1,FILE_2,FILE_3,FILE_4,...

COMPILERS

- . Create object modules from source code.
- . Source code can have multiple program units.
 - AND- object file can have multiple object modules.

- . Some switches that you can use on the compiler command.
 - /DEBUG adds symbols, entry point, line # info.
 - /CHECK for out-of-bound subscripts, arithmetic overflows and underflows.
 - /NOOPTIMIZE - Speeds up compile time; especially useful during early compiles.
 - /LIST (can use in conjunction with /NOOBJECT) line #'s, variable datatypes & addresses.

OBJECT LIBRARIES

- . Created with \$ LIBRARY/CREATE lib_name.
- . Add entries with:
 - \$ LIBRARY/REPLACE lib_name object module.
- . Entries can be extracted and deleted.
- . Libraries contain compiled object modules.
 - and- other objects (not compiled):
 - command language descriptions.
 - error descriptions.
 - symbol definitions.
 - system-defined procedures.

LINKER

- . Invoked with "\$ LINK ..." command.
- . Gets object modules from object files and/or library files.
- . Creates executable and shareable images.
 - A shareable image cannot be run by itself; useful for routines called from multiple programs.
- . /DEBUG appends symbol, line # info to image.
 - causes image to run under debugger by default.
 - override default at runtime with \$ RUN/NODEBUG (and still enter debugger after CTRL/Y).
- . /TRACEBACK dumps image/process state after error.
- Very useful during program development.
- . /MAP/FULL provides virtual memory map, global symbols, crossreference, module synopses...

GLOBAL SYMBOLS, LIBRARIES

- . Linker resolves global symbols by searches of:
 - Explicitly named modules & libraries.

- System default libraries.
- User default libraries.
- . User default libraries: (mylib is an object library).

\$ DEFINE LNK\$LIBRARY dev:[dir]MYLIB

SHAREABLE IMAGES

- . Shareable images = nonexecutable.
 - Saves disk space.
 - Executable images link to it without physically including it.
 - Linkage is setup when image is activated.
 - Use transfer vector macro to save having to relink executing image when shareable changes.
 - Use CLUSTER in options file to bind macro+image.

SHAREABLE IMAGES

- . Use /GSMATCH in options file to indicate whether executable image must relink when shareable changes.
- . Create shareable image libraries with.
 - \$ LIBRARY /CREATE /SHAREABLE image_name.
 - Default file type is .OLB.
- . ADV: Save Disk Space, Maintainability.
- . DISADV: Image execution is slower, depending on number and size of shareable images.

PRIVILEGED PROGRAMS

- . Some programs execute privileged system services, or obtain access and resources through enabled privs.
 - . Instead of giving all users the privileges:
 - Install image with privs.
 - \$ INSTALL
- INSTALL> CREATE dev:[dir]image /PRIV=(privname)

SHARED IMAGES

- . Shared image: only one copy in memory.
- . Use Install Utility to make image shared.

CHECKING UP ON PROGRAMS

- . SHOW PROCESS /CONTINUOUS /ID=xxx
 - Information on program counters, amount of CPU, e.g.
- . CTRL/T
- . RUN /PROCESS image /DUMP
 - examine dump with System Dump Analyzer.
- . RUN /PROCESS image /ERROR.
- . Relink image with /DEBUG and/or /TRACEBACK.

DEBUGGER

- . Can be invoked at compile, link or execution time.
- . If invoked only at execution time, won't have access to symbol table.

DEBUGGING COMFORT

- . In window mode, shows 3 default displays: Debugger output, Source code, Register contents.
- . You can save a snapshot of a display.
- . You can define other displays.
- . Has keypad mode.

- . Has HELP and SPAWN commands.
- . There are language sensitive editors for some languages that can interact with the debugger.

WHERE TO BREAK, WHAT TO DO

- . Can set breakpoint at routine start, at exception break, at any location, on a type of instruction,...
- . Can set tracepoints at same places, to just display execution of interesting instruction and continue.
- . Can activate breakpoint /AFTER n iterations.
- . Can conditionally execute list of commands at break.
- . Set watchpoint to display modifications to location.

OTHER INFORMATION

- . SHOW MODULES
- . SHOW REGISTERS
- . SHOW CALLS

SYMBOLS

- . SET SCOPE to define program region to use in interpreting symbols; done at link time.
- . Make symbol uniquely identifiable with pathname prefix.

module\routine\block\section\line\symbolname

EDITORS AVAILABLE ON VMS

- . SOS --line oriented editor.
- . EDT --line mode.
 - screen mode with keypad functionality.
 - screen mode with typed-in commands.
 - Now on TOPS-10 and TOPS-20 also.
- . EMACS-32 --available from third-party/
- . TECO/TV | Both available on Integration
- . SED | Tools Tape, unsupported.
- . TPU

SYSTEM ROUTINES

- . Languages which can call system routines:
 - MACRO, BASIC, BLISS-32, C, COBOL(-74), ADA, CORAL, DIBOL, FORTRAN, Pascal, PL/1.
- . Arguments are passed by
 - value, reference or descriptor.
- . Condition value always returned.
- . Kinds of system routines:
 - System services.
 - Run time library routines, e.g., RMS file handling.
 - Utility routines.

SYSTEM SERVICES

- . Functions:
 - Security - check protections, ACE's, identifiers, disk erase.
 - Event Flag Services.
 - ASTs - set and deliver.
 - Logical Names - create, delete, translate.
 - I/O - assign channels, pass messages to QIO, device, volume mailbox, brkthru, message to job

- controller, operator, etc.
- Process Control - creation, state, priority, privs.

SYSTEM SERVICES 2

- . Timer and Time Conversion (between various date and time formats).
- . Condition Handling Set Up.
- . Memory Management.
working set, global section, lock page swap mode, stack limits, map section...
- . Lock Management - enqueue/dequeue, get lock information.

INTERPROCESS COMMUNICATION

- . List is ordered, more or less, by size.
 - . Common Event Flags (fast, but only bits).
 - . Logical Name Tables
 - . Mailboxes
 - . Global Sections (fastest).
 - . Lock Management (fast, but only bytes).
 - . Shared Files (slowest, unlimited data).
 - . Decnet task-to-task - any length across Decnet.
- \ / limited amount of data accepted.

RUN TIME LIBRARY

- . Same calling and return standards as System Routines.
- . Use RMS for file I/O.
- . Execute in same access mode as caller.
- . Major Subsets such as the following utility libraries:
 - Mathematics
 - Resource Allocation
 - Condition Handling
 - Screen Management
 - Image/Process Handling

OTHER SYSTEM ROUTINES

- . Command Line Interpreter (CLI) Parsing.
- . RMS Services.
- . File Definition Language Routines.
- . Sort/Merge Routines.
- . File Conversion Services.
- . Data Compression/Expansion.
- . EDT Access as well as access to other editors.
- . Librarian Routines.
- . Print Symbiont, Job Controller Interface.

INTRA PROCESS COMMUNICATION

- . (Between different images executed by same process.)
- . Local Event Flags.
- . Per-process Common Blocks.
- . ASTs.
- . Symbol Table.

FILE SYSTEM

- . File ORGANIZATION can be:
 - Sequential
 - Relative
 - Indexed

FILE SYSTEM

- . Record Formats:
 - Fixed length (all organizations).
 - Variable length (all organizations).
 - Variable with fixed-length control (Sequential and Relative).
 - Stream (terminator delimited)(Disk Sequential only).

FILE SYSTEM

- . File ACCESS modes:
 - Sequential (works with all organizations).
 - Random Access by Key Value (for Indexed only).
 - Random by relative record number (Sequential and Relative).
 - Random by Record File Address (works with all).
 - Block I/O.

TOPS TO VMS BUSINESS APPLICATION
TOPS-10/VAX PERFORMANCE COMPARISON

Frank Francois and Ralph Bender
Federal Home Loan Bank Board
Washington, D.C.
(202) 377-6115

ABSTRACT

This session was a presentation of a COBOL applications benchmark conducted by the Federal Home Loan Bank Board. The benchmark was run on a DECsystem-10, a VAX-11/780, a VAX-11/785 and a VAX-8600; and the results of all runs were presented and compared.

Ralph Bender presented background on the Federal Home Loan Bank Board (FHLBB) and on the business benchmark. The configuration for the FHLBB's Computer Center is in Exhibit I; it shows the TRI-SMP DECsystem-10, the VAX-11/780, the VAX-11/785 and all peripherals. The FHLBB is the federal government agency that regulates the savings and loan industry. Users of the Computer Center are 500 people in Washington, the 12 Federal Home Loan Banks throughout the United States and the 27 Examination Offices throughout the United States. Remote access is done through Telenet.

This session was planned at DECUS in New Orleans in May 1985. None of the benchmarks presented previously at DECUS showed DECsystem-10 to VAX comparisons, nor did they show COBOL, business applications. The need for a DECUS session coincided with the Bank Board's study of future data processing needs. The results of that study were to migrate to the VAX-8600 from the DECsystem-10. However, the FHLBB wanted to do benchmarks before making the final decision.

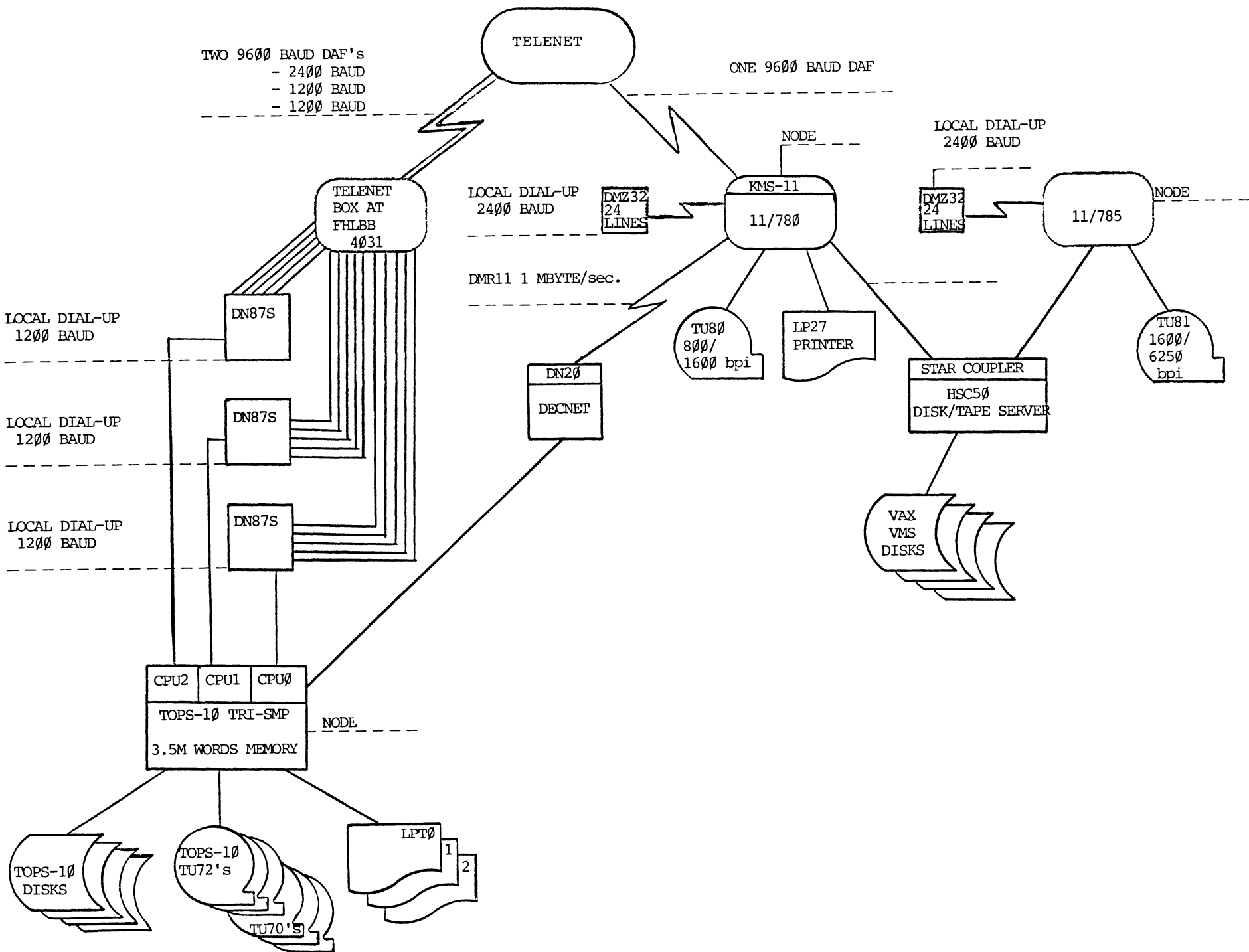
The FHLBB had experience in benchmarks dating back to the purchase of their first KA1050 in 1973. The FHLBB's DP division uses COBOL for 90% of its production, and they also have applications running in IFPS, DPL and FORTRAN. The benchmark was designed to reflect the normal, COBOL business workload. Operator jobs plus 18 application jobs were run concurrently for 90 minutes to heavily load the four systems that were benchmarked. Ralph indicated that the participants in the FHLBB benchmark project were Tom Wood (Computer Center Manager), Frank Francois (Technical Support Manager), Donn Lindsey (Programming Manager), Carl Spellacy (Computer Operations Manager) and Jean Nowak and Ron Leisey of the Technical Support staff. Ralph then introduced Frank Francois, who did much of the management of the benchmark and who presented the results.

Frank used the enclosed exhibits to present the results of the benchmark.

- . Exhibit 2 shows in detail the purpose and content of the 18 jobs in the benchmark mix. The jobs consisted of 8 types of jobs: 3 COBOL compiles, 3 COBOL jobs doing disk input and terminal output, a "normal" COBOL application program, 3 COBOL programs doing intensive computations, 3 COBOL programs that exercise memory, 3 COBOL programs doing terminal input, 1 backup utility doing backup, and 1 COBOL sort program.
- . Exhibit 3 - The DECsystem-10 (running TOPS-10/7.02) benchmark configuration.
- . Exhibit 4 - The VAX-11/780 (running VMS 4.2) benchmark configuration.
- . Exhibit 5 - The VAX-11/785 (running VMS 4.2) benchmark configuration.
- . Exhibit 6 - The VAX-8600 (running VMS 4.2) benchmark configuration. The benchmark was run on the VAX-8600 with 6MB of memory and again with 8MB of memory.
- . Exhibit 7 - For each of the 8 types of jobs, this shows transactions processed on the KL-1090, VAX-780 and VAX-785. Transactions equals number of lines compiled for compiles, transactions input/output for terminal input/output, number of computations for computational programs, number of records backed up for backup, and number of records sorted for sort.
- . Exhibit 8 - For each of the 8 types of jobs, this shows transactions processed on the KL-1090, VAX-780, VAX-785 and VAX-8600 with 6MB of memory.
- . Exhibit 9 - For each of the 8 types of jobs, this shows transactions processed on all five of the benchmark configurations - the KL-1090, VAX-780, VAX-785, VAX-8600 with 6MB and VAX-8600 with 8MB.
- . Exhibit 10 - For the 8 types of jobs, this shows the CPU time allotted on the KL-1090, VAX-780, VAX-785, VAX-8600 with 6MB and VAX-8600 with 8MB.

- . Exhibit 11 - For each of the five benchmark configurations, these graphs show performance on the compiles.
- . Exhibit 12 - These graphs show performance on the BMKBIG program that exercised memory paging.
- . Exhibit 13 - These graphs show performance on terminal input.
- . Exhibit 14 - These graphs show performance on terminal output.
- . Exhibit 15 - These graphs show performance on BMK31, the program that has a heavy computational load.
- . Exhibit 16 - These graphs show performance on BMK32, the second program that has a heavy computational load.
- . Exhibit 17 - These graphs show performance on BMK21, the "normal" COBOL program.
- . Exhibit 18 - These graphs show performance on the Backup utility.
- . Exhibit 19 - These graphs show performance on the sort.
- . Exhibit 20 - For the 8 types of jobs, this graph shows the relative processing power of the KL-1090, VAX-780, VAX-785, VAX-8600 with 6MB and VAX-8600 with 8MB.

Conclusion: The overall results of these benchmarks indicated that the VAX-8600 performed very well on COBOL applications and processing power appeared to be consistent with FORTRAN benchmarks previously presented at DECUS. The FHLBB was satisfied that the VAX-8600 would give us the processing power needed for our follow-on computer to the DECsystem-10.



EXPLANATION OF BENCHMARK PROGRAMS

<u>Job #</u>	<u>Program</u>	<u>Benchmark *</u> <u>Repetitions</u>	<u>Job #</u>	<u>Program</u>	<u>Benchmark *</u> <u>Repetitions</u>
1	Compiles	90	10	BMK 32	90
2	Compiles	90	11	BMKBIG	90
3	Compiles	90	12	BMKBIG	90
4	BMK 01	90	13	BMKBIG	90
5	BMK 01	90	14	BMKACC	C
6	BMK 01	90	15	BMKACC	C
7	BMK 21	90	16	BMKACC	C
8	BMK 31	90	17	BACKUP	C
9	BMK 31	90	18	SORT	90

* 90 = Ran for 90 wall clock minutes; C = Ran once to completion.

Jobs 1,2,3: COBOL Compiles - Each of these jobs consists of the same six COBOL programs, compiled with the cross reference and compile list options, the latter to keep the line printer busy during the benchmark. The number of statements for each program is shown below:

	<u>Non</u> <u>Procedure</u>	<u>Procedure</u> <u>Division</u>	<u>Total</u>
Program 1	174	465	639
Program 2	674	1180	1854
Program 3	2155	1101	3256
Program 4	1870	797	2667
Program 5	166	355	521
Program 6	1498	894	<u>2392</u>
			<u>11,329</u>

These programs are not executed, only clean-compiled in a round-robin fashion by job; e.g., after program 6 in job 1 is finished, program 1 in job 1 begins compiling again.

Jobs 4,5,6: Program BMK01 - The purpose of this COBOL program is to put a disk input and 1200 baud CRT terminal output load on the system. Upon loading, the program first (and one-time only) creates a fifty (50) 132 character record random file. During the normal processing loop, a random file record is read and a 72 character message is displayed on a 1200 baud CRT. The message contains a sequential number, the calculated response time in seconds since the last message was displayed, and the time of day (HH MM SS) of this current message. For a subsequent hard copy review, every 200 displays, the same message is sent to a report file on disk.

Job 7: Program BMK21 - This COBOL program simulates a typical program at our installation. It is neither I/O nor compute bound. A normal processing loop consists of twenty (20) internal Working Storage moves followed by 200 calculations, a record written to random file 1, then this sequence repeated except that a record is written to random file 2. There is no terminal display. For a hard copy audit of system response and thru-put for this program, for every 200 complete processing loops a record is sent to a report file showing time of day, the response time since the last timing, and the calculated average response time of the last 10 timings.

Jobs 8,9: Program BMK31 - This COBOL program puts a computational load (as opposed to an I/O load) on the system. A 500 element array is initialized with the same number throughout. During each normal processing loop, the program takes the square root of successive elements in the array, from 1 to 500 (without replacing the computed values back into the array). At the end of each loop, a message is sent to a report file showing time of day and the time necessary to complete this loop.

Job 10: Program BMK32 - This COBOL program is identical to BMK31.

Jobs 11,12,13: Program BMKBIG - This COBOL program exercises memory paging through successive references to individual fields in non-contiguous memory locations. Six Working Storage tables, work-1 thru work-6, each contain 7000 numeric [S9(12)] Comp fields. In each normal processing loop the program increments a subscript, then moves a numeric constant to a single subscripted field in each of the six tables. Every 100 loops, the program displays a message showing the time-of-day and the number of loops completed.

Jobs 14,15,16: Program BMKACC - This COBOL program uses a micro computer with one file on a single floppy disk to upload data to the mainframe at 1200 baud, for the purpose of putting a terminal input load on the system. The floppy disk file contains 2,344 records of 128 characters each. COBOL program BMKACC reads a record, checks position 1-4 for the next sequential tally, checks position 80 for an *, and position 128 (the last character) for a # sign to insure that the entire record is read and available for processing. After this is done, the program reads the next record and repeats the checking. The program terminates upon completion of the file upload.

Job 17: Backup Utility - This job uses the backup system utility (DEC10 or VAX) to backup a file to a 1600 BPI tape. A file of one hundred thousand (100,000) records of 128 characters each is backed up to tape once during the benchmark. The file is not accessed by any other program or utility during the benchmark.

Job 18: Sort Utility - This job uses a COBOL Sort Program to sort a file of twenty thousand (20,000) 128 character records. The sort is in ascending sequence on a single 11 position numeric field. When the sort is finished, it cycles around to again sort the unsorted file.

EXHIBIT 3

TOPS 10 BENCHMARK CONFIGURATION

1.5 M Words Memory
KL1090B CPU
TX01/DX10/TU72 Tape System
DN87 (1200 Baud)
3 RP07's - Just used for Swapping
1 RP06 - System pack
1 RP07 - User Disk
1 LP07C - Line Printer
Monitor - 702
 With Galaxy 4.1

EXHIBIT 4

VAX-11/780 BENCHMARK CONFIGURATION

6 M characters memory
780 CPU
TU80 Tape Drive
DMZ32 Communication Device
HSC50/RA81 Disk (System Disk, Swapper Disk and User Disk)
Monitor VMS V4.1
LP27 Line Printer

EXHIBIT 5

VAX-11/785 BENCHMARK CONFIGURATION

6 M characters memory
785 CPU
TU80 Tape Drive
DMZ32 Communication Device
HSC50/RA81 Disk (System Disk, Swapper Disk and User Disk)
Monitor VMS V4.1

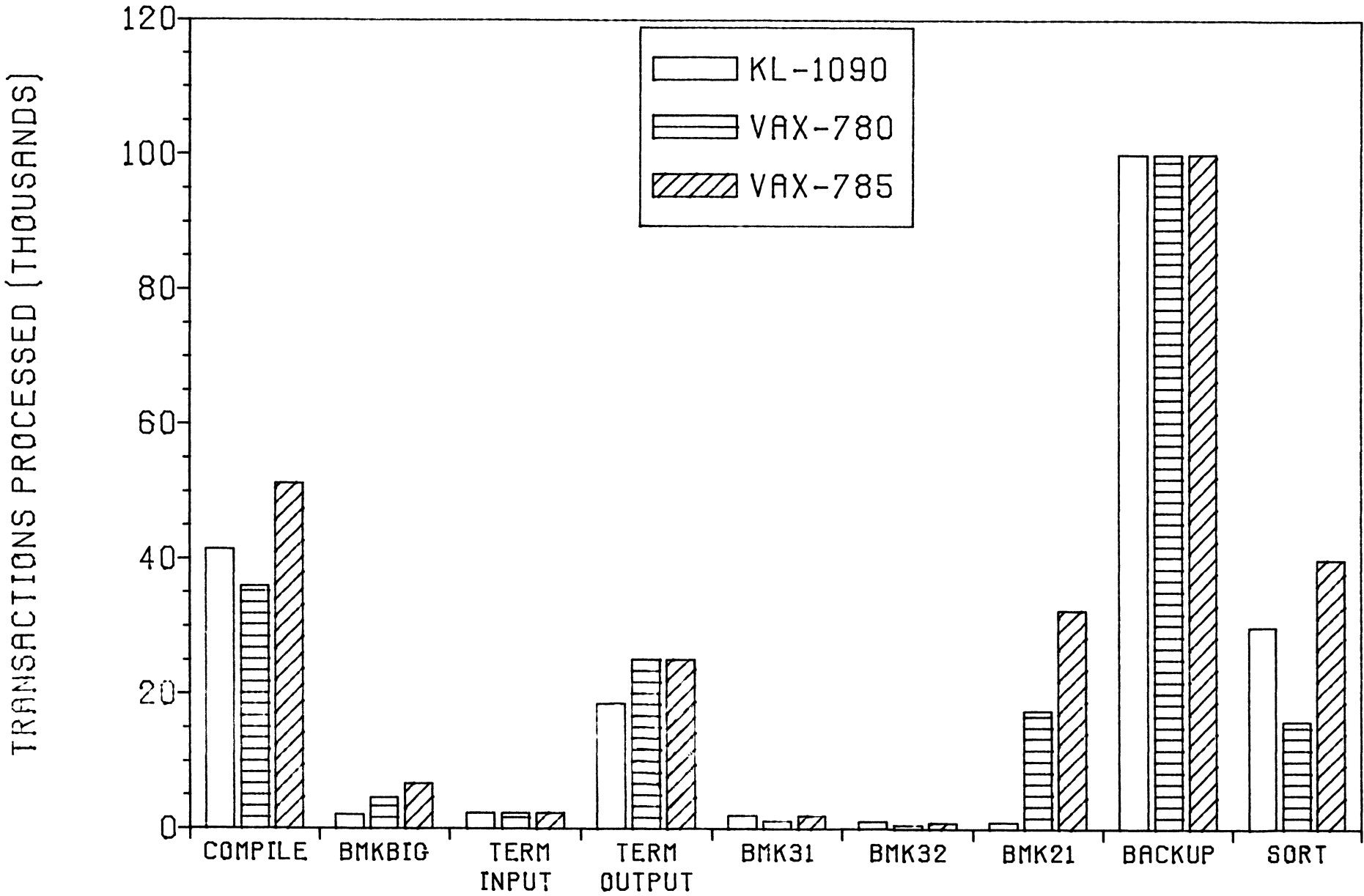
EXHIBIT 6

VAX 8600 BENCHMARK CONFIGURATION

6 M characters memory
8600 CPU (with floating point)
Tape T478
UDS50/RA81 (2) (System Disk, Swapping Disk and User Disk)
Monitor VMS V4.1
LP27 Line Printer
DMZ32 Communication Device

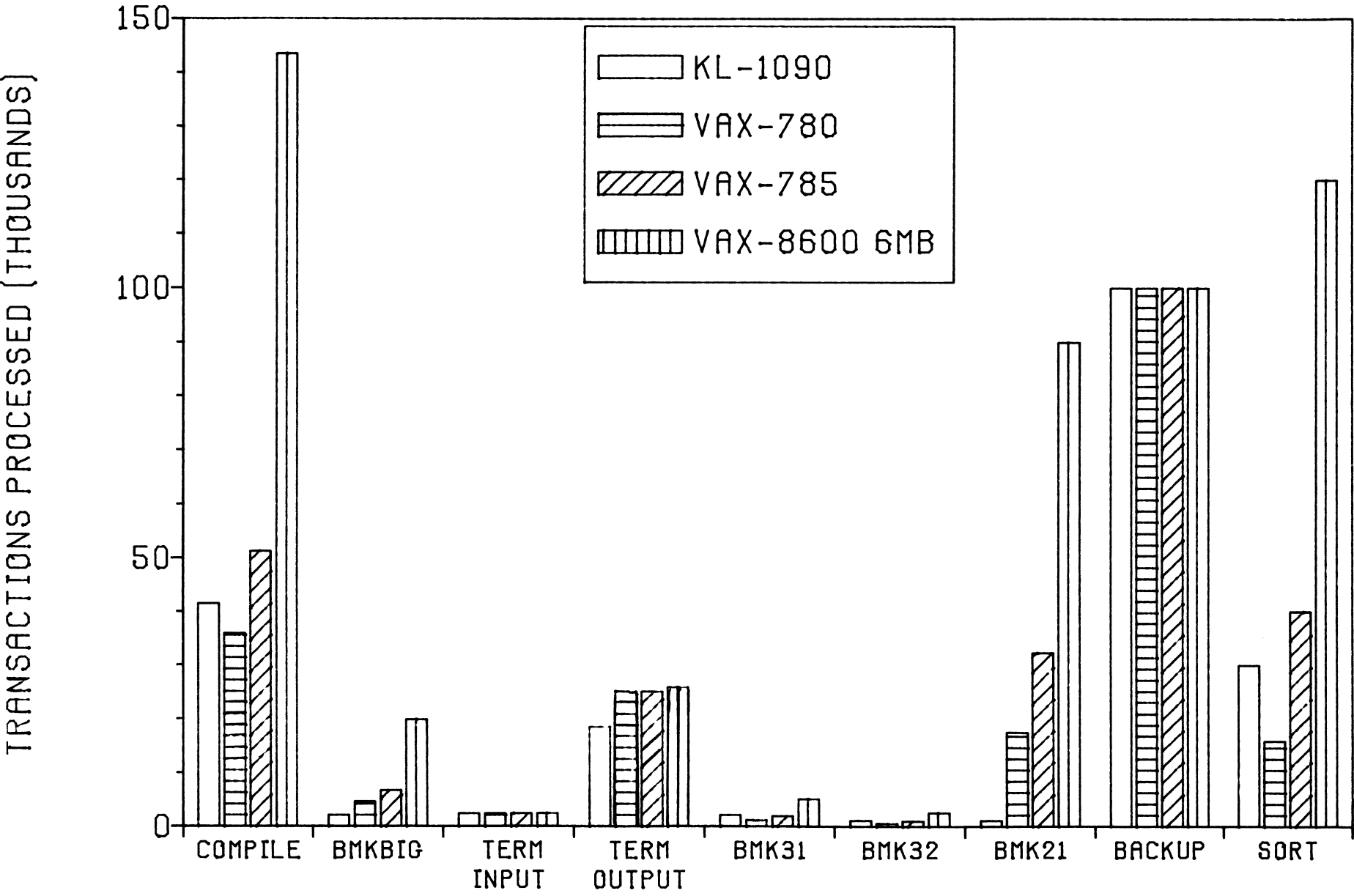
FHLBB BENCHMARK

Transactions Processed



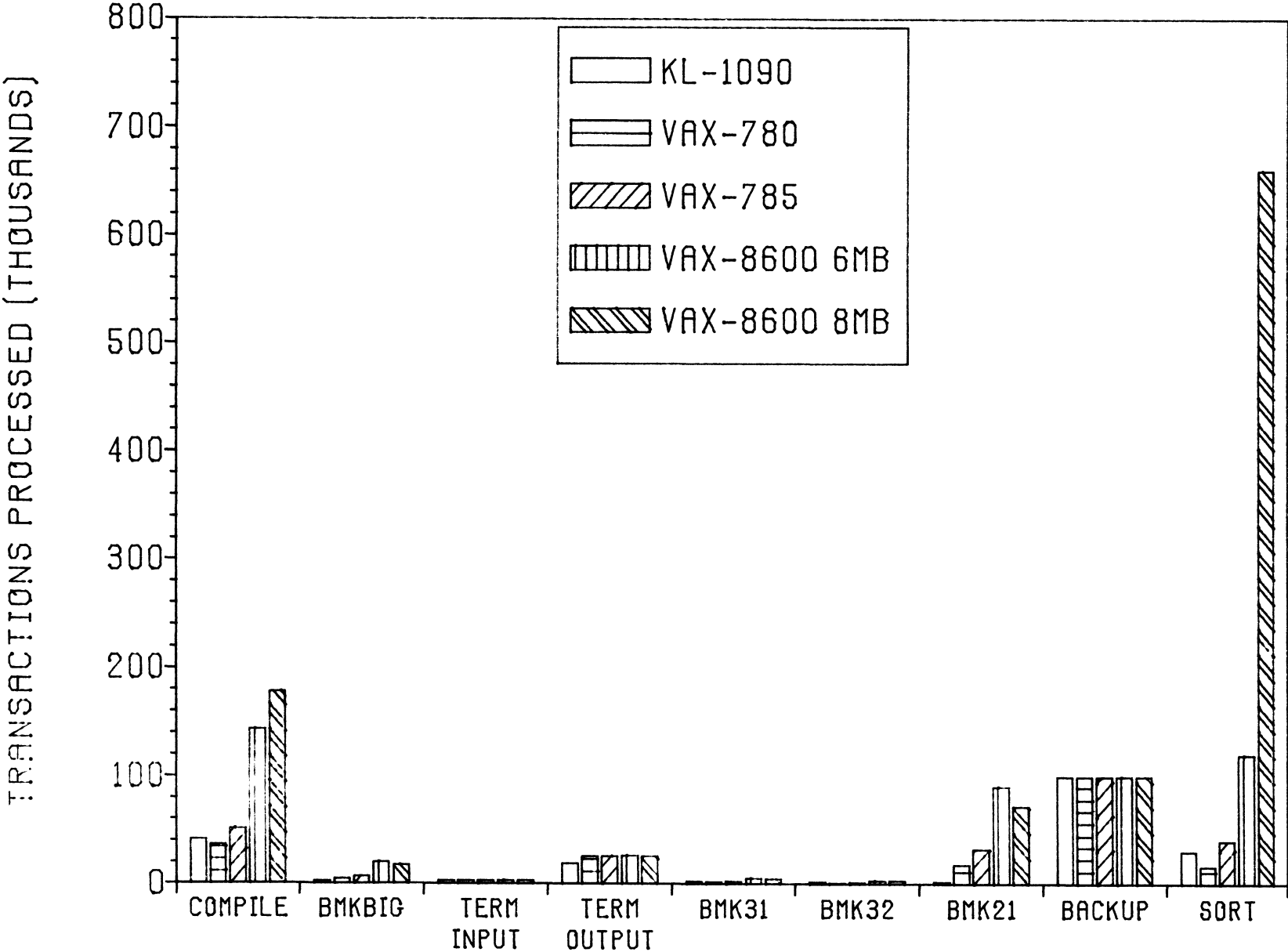
FHLBB BENCHMARK

Transactions Processed



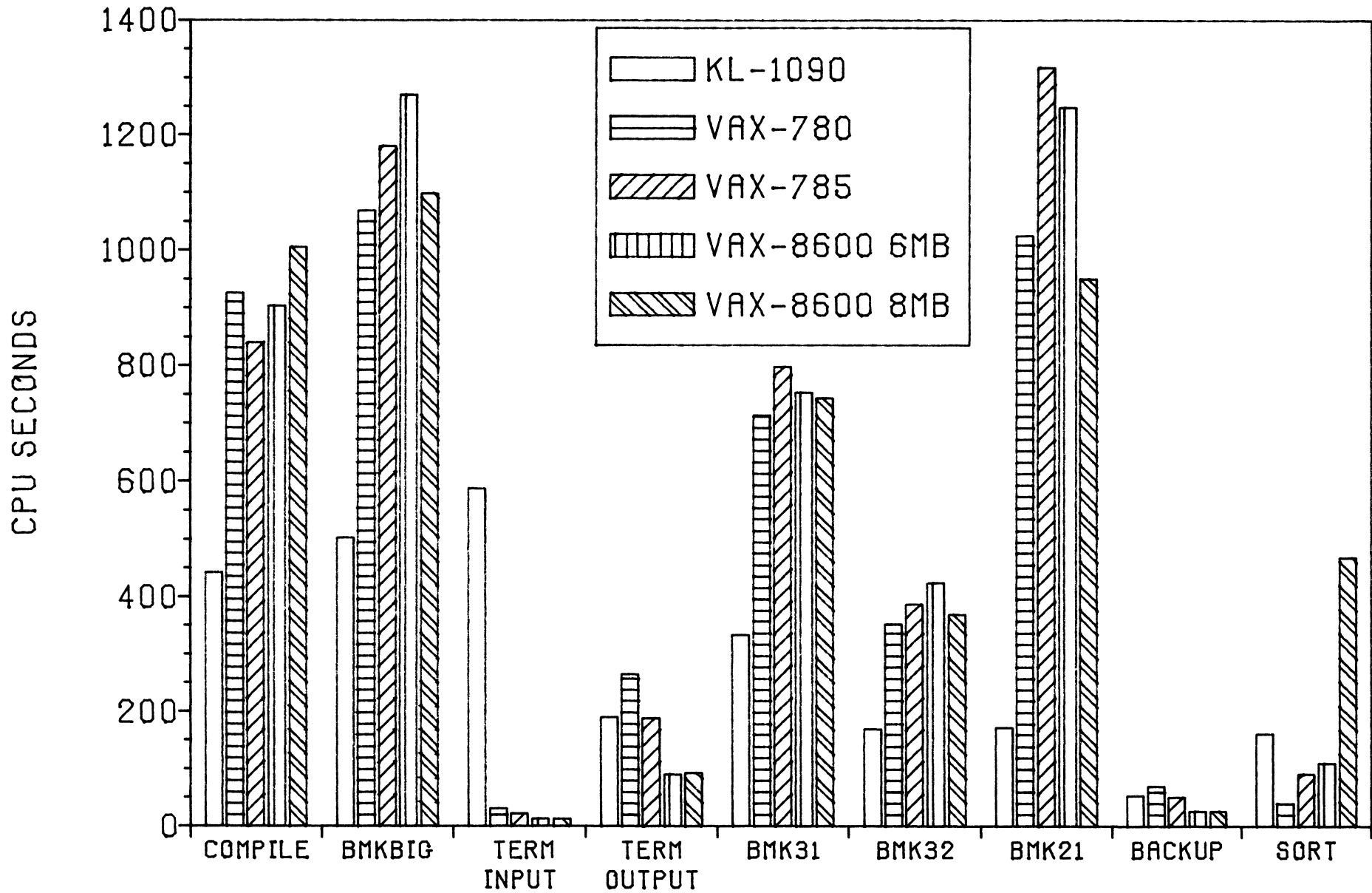
FHLBB BENCHMARK

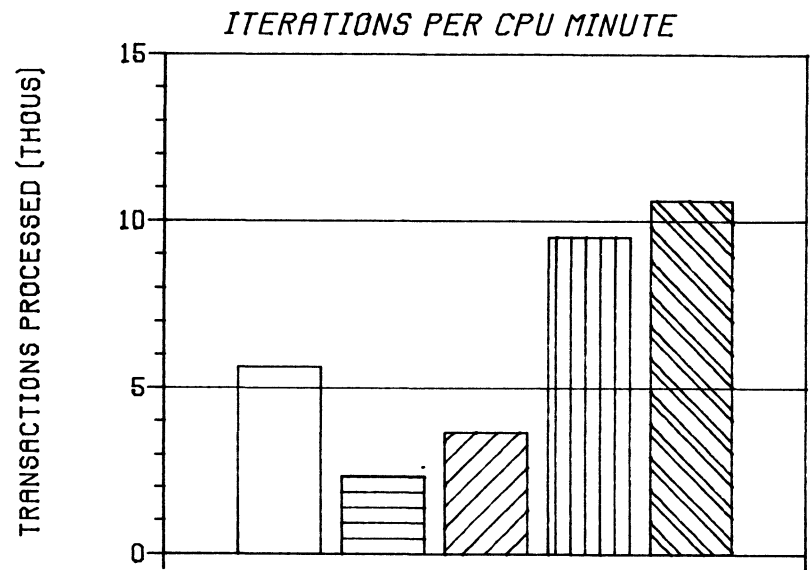
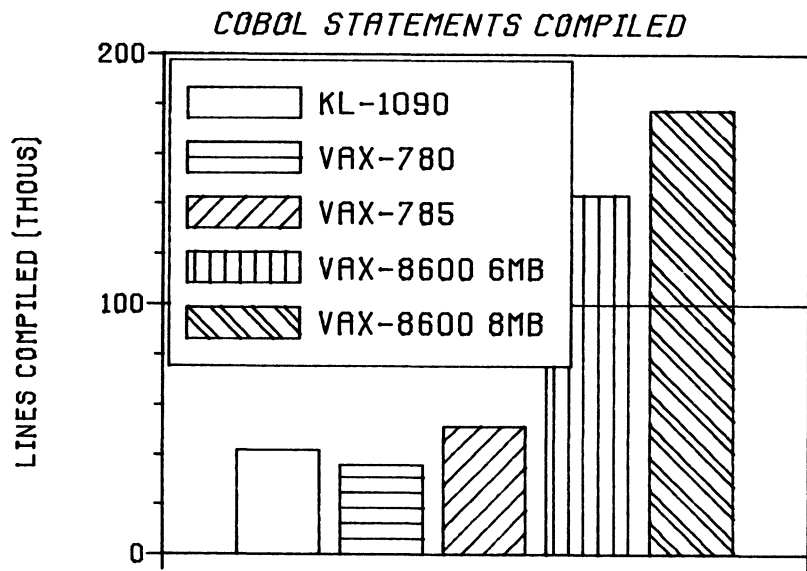
Transactions Processed



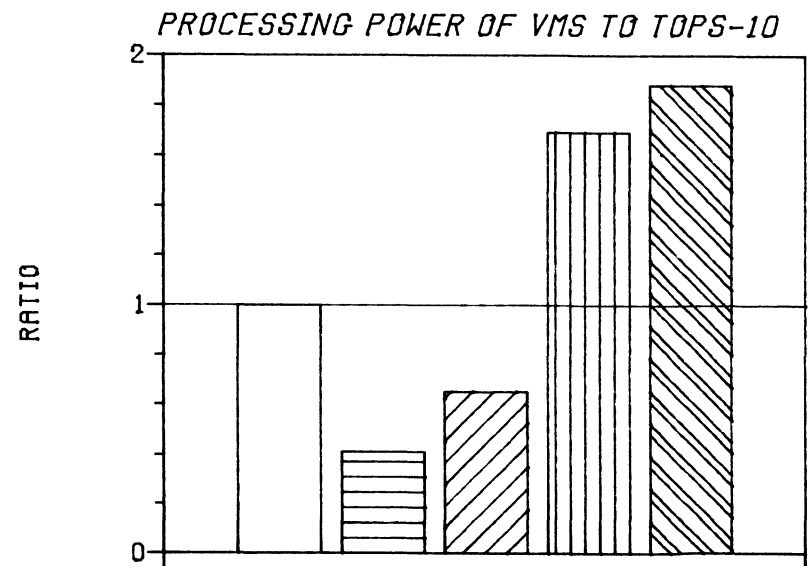
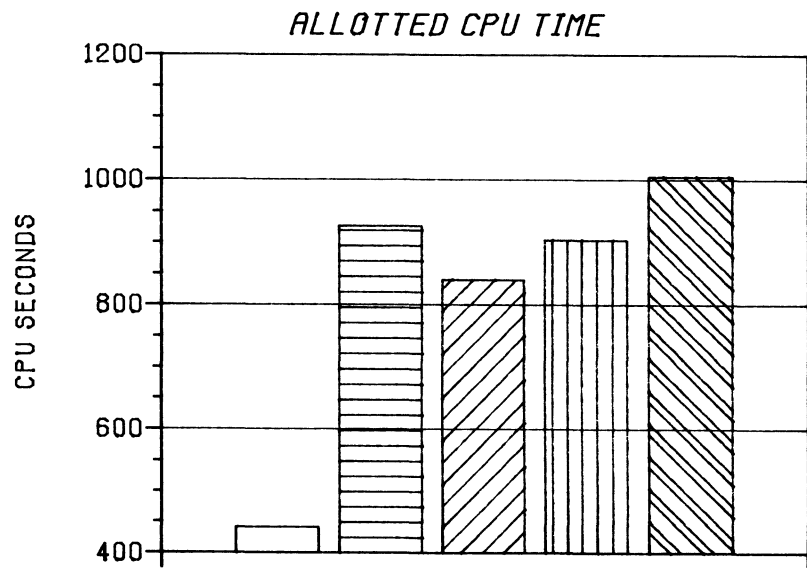
FHLBB BENCHMARK

Allotted CPU Time

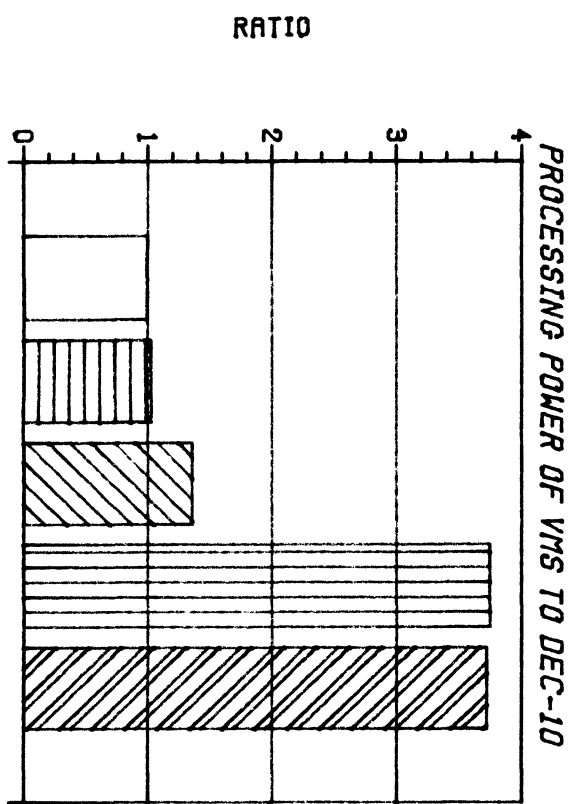
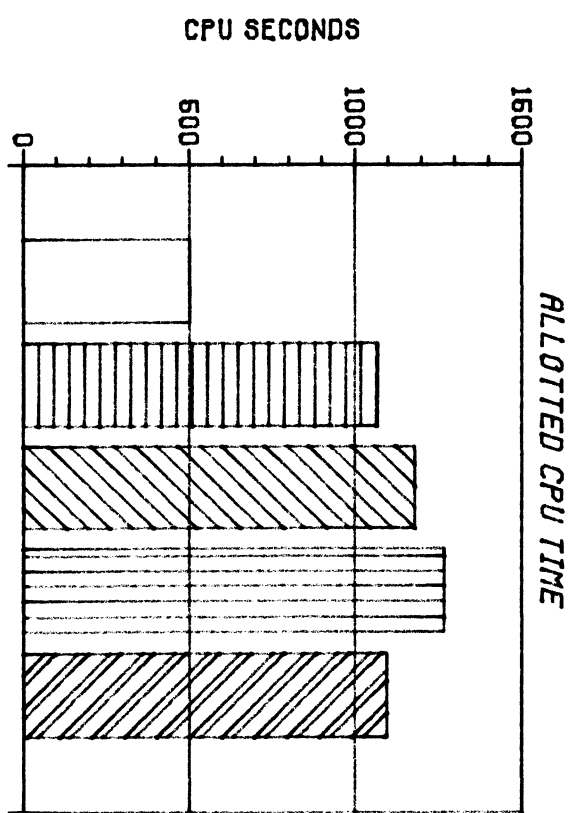
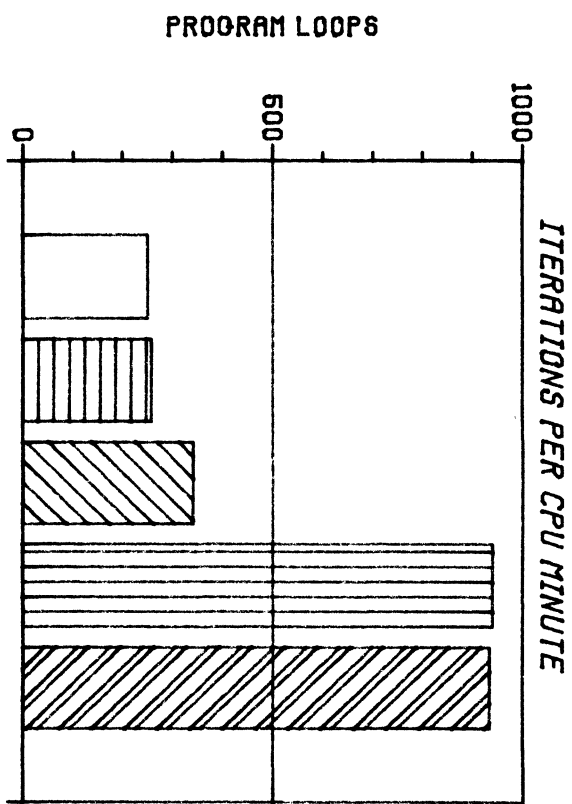
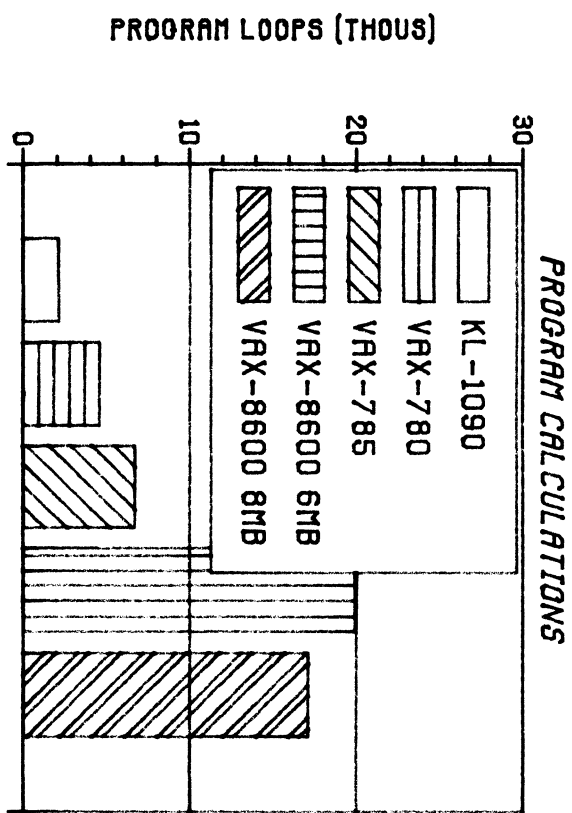


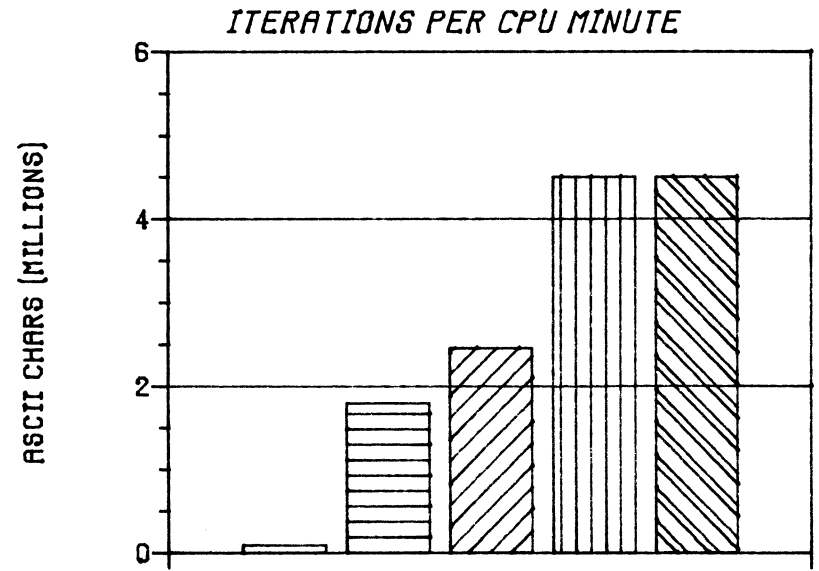
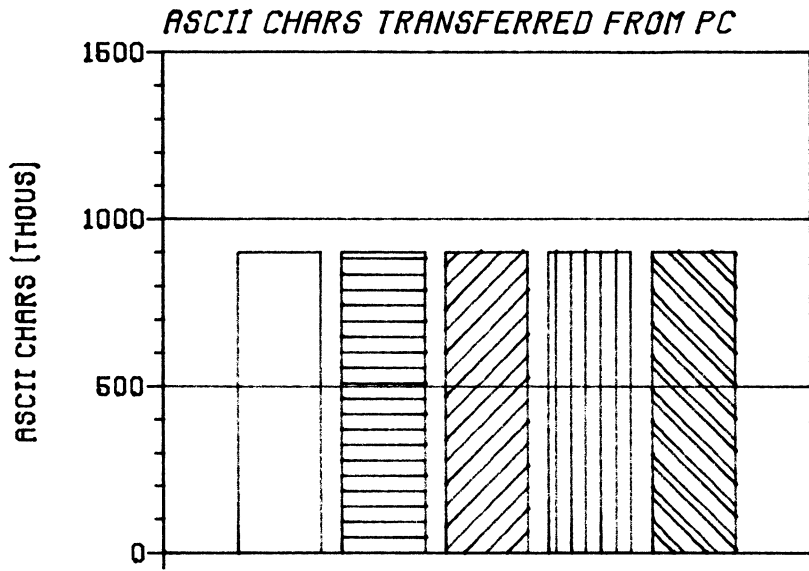


BENCHMARK - 90 Minutes (COMPILE)

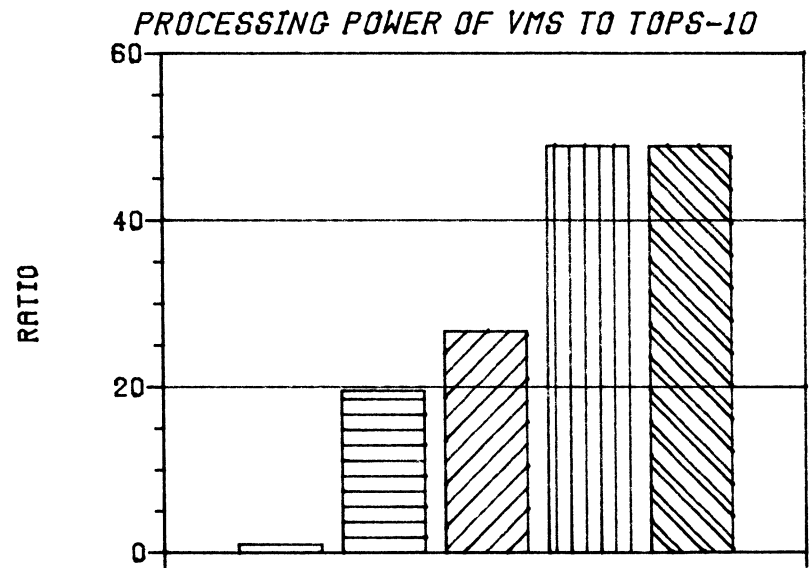
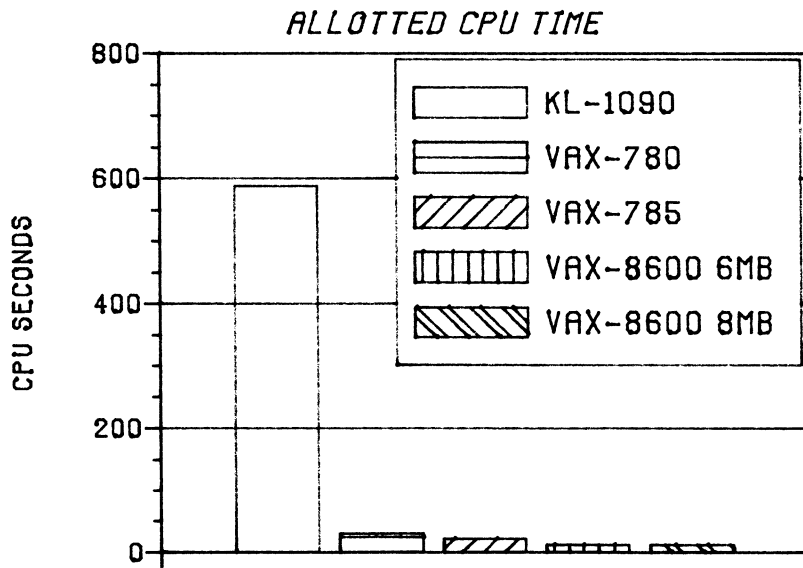


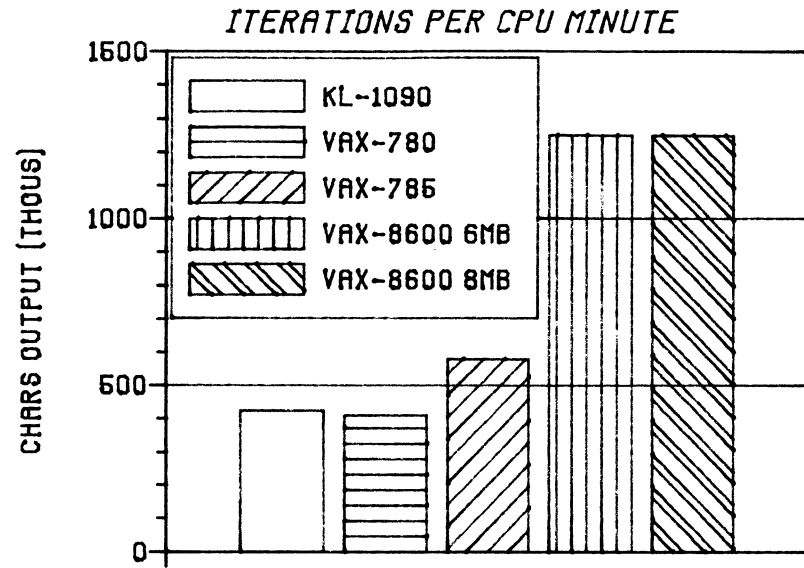
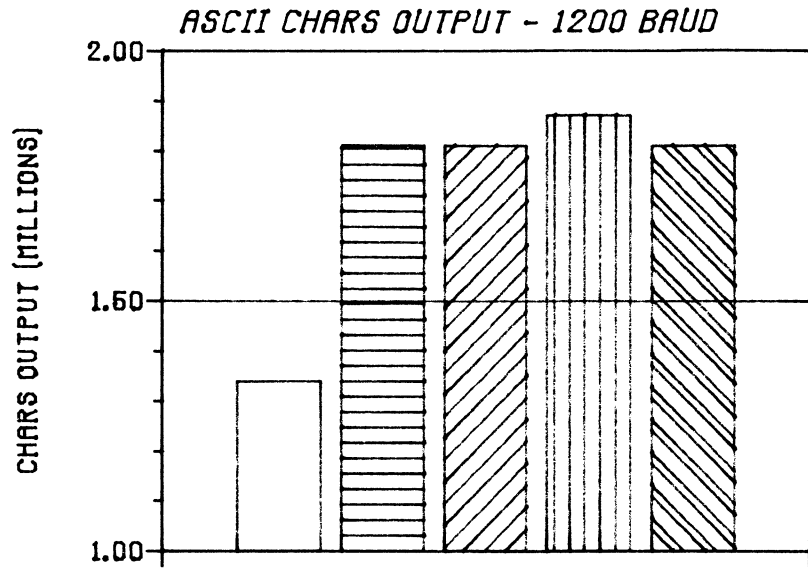
BENCHMARK - 90 Minutes (BMKBIG)



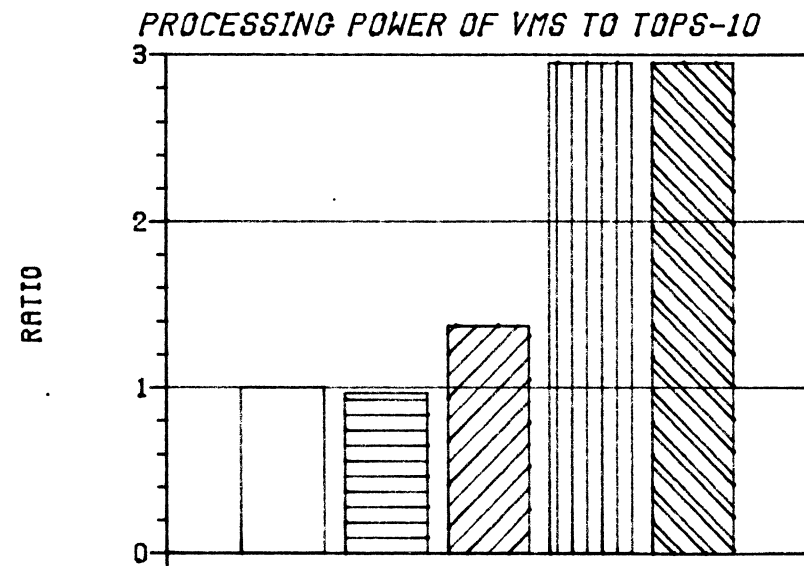
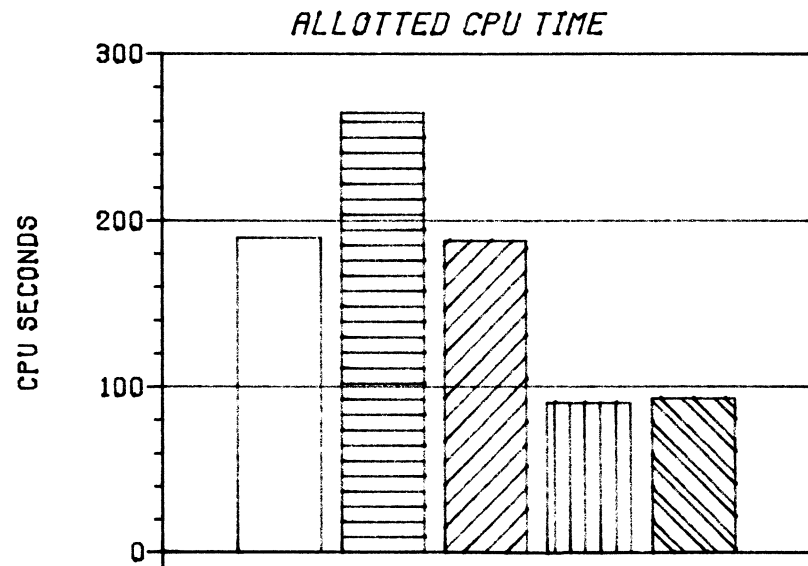


BENCHMARK - 90 Minutes (*TERM INPUT*)

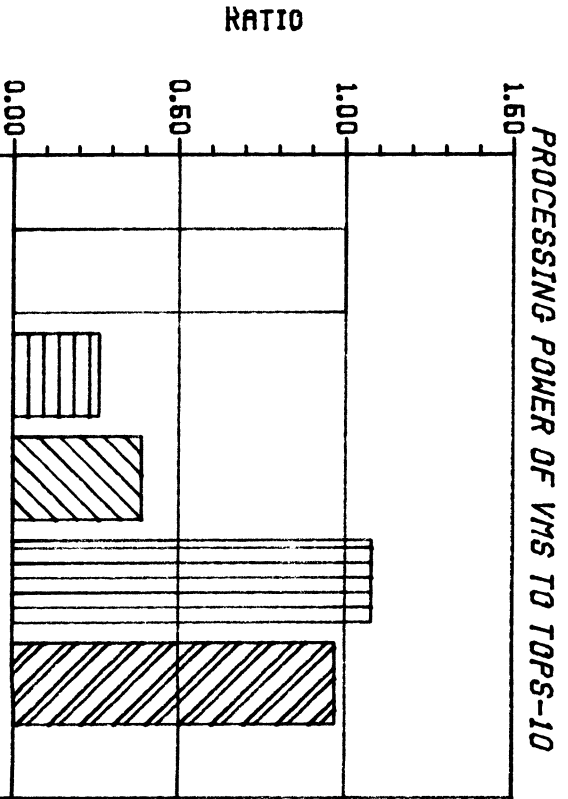
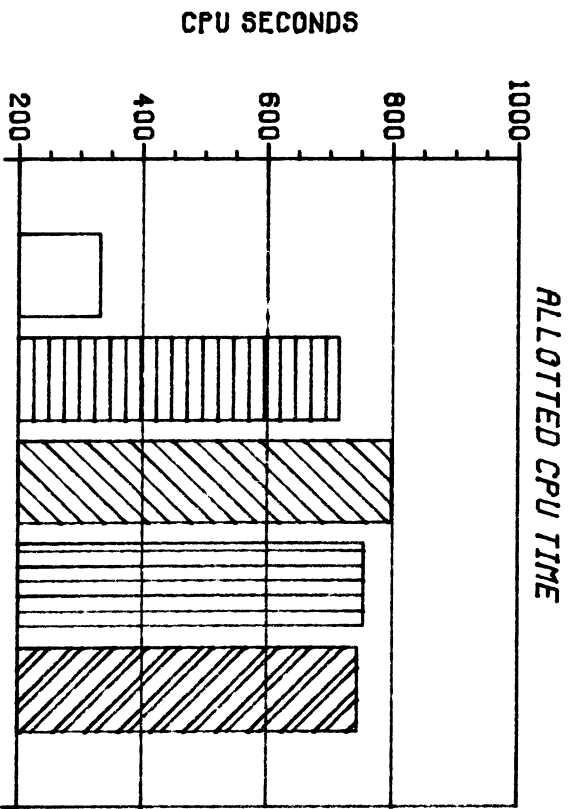
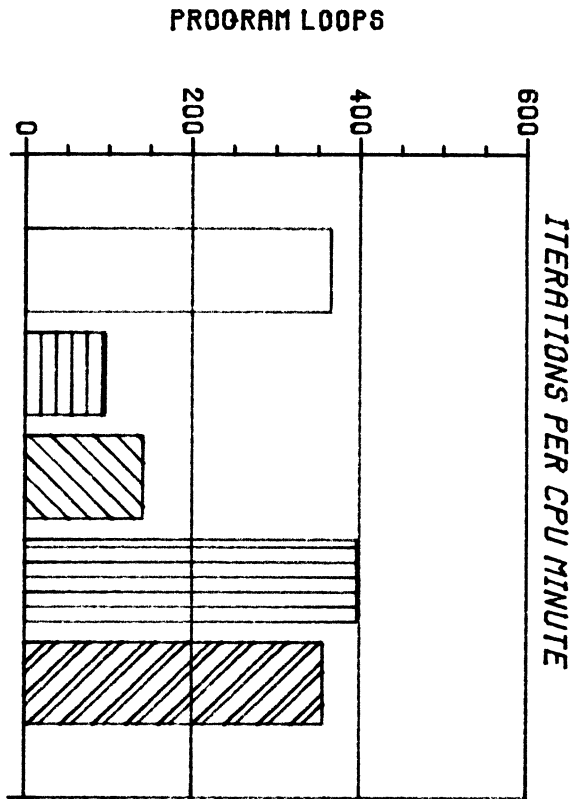
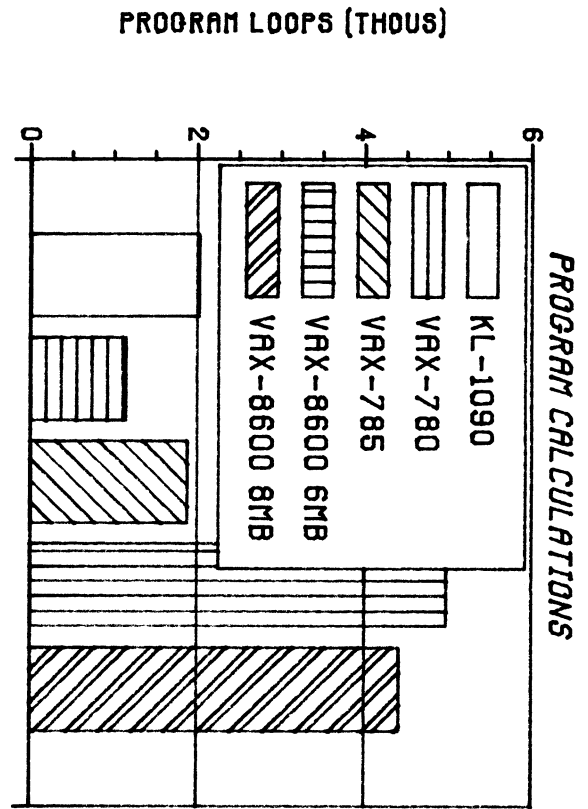


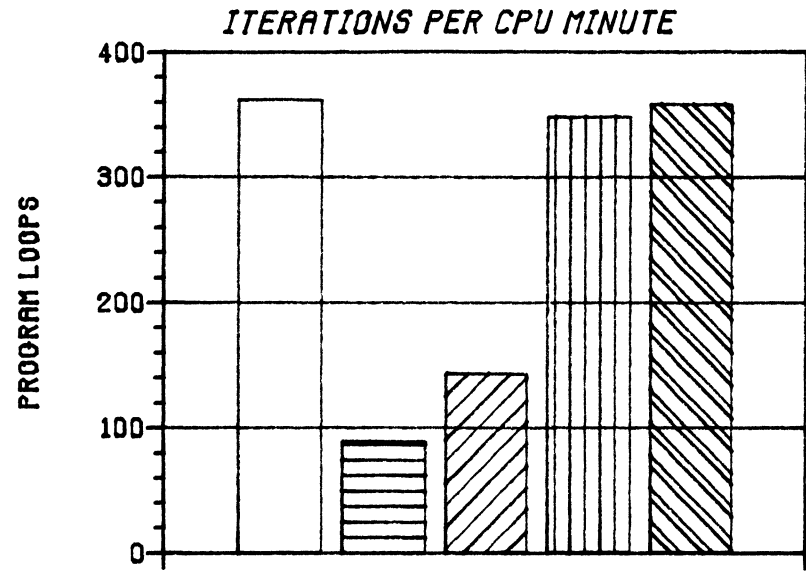
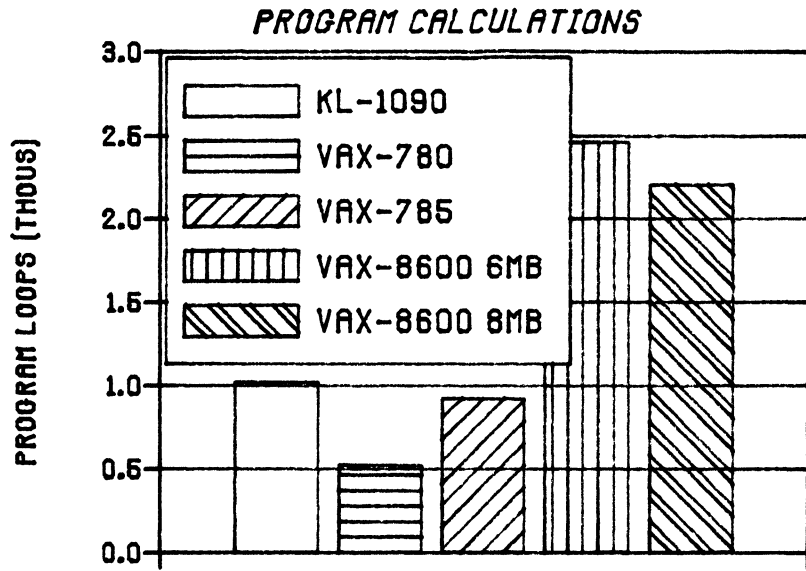


BENCHMARK - 90 Minutes (*TERM OUTPUT*)

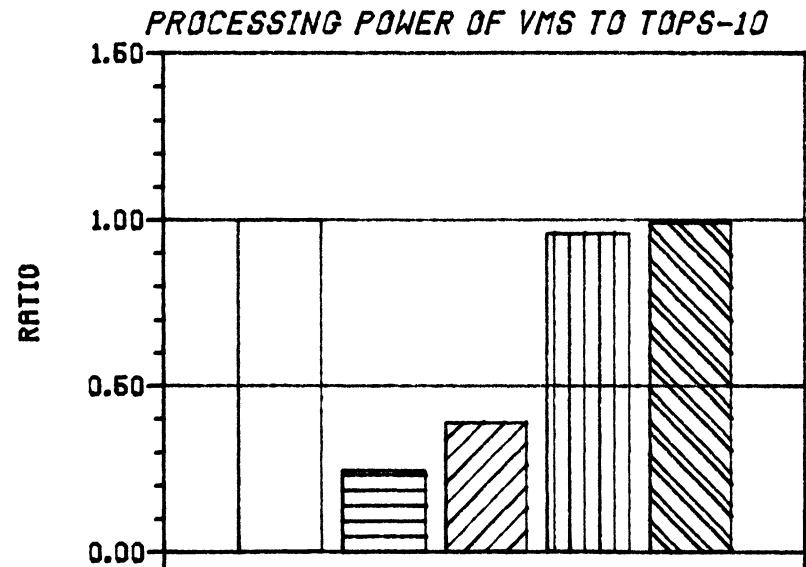
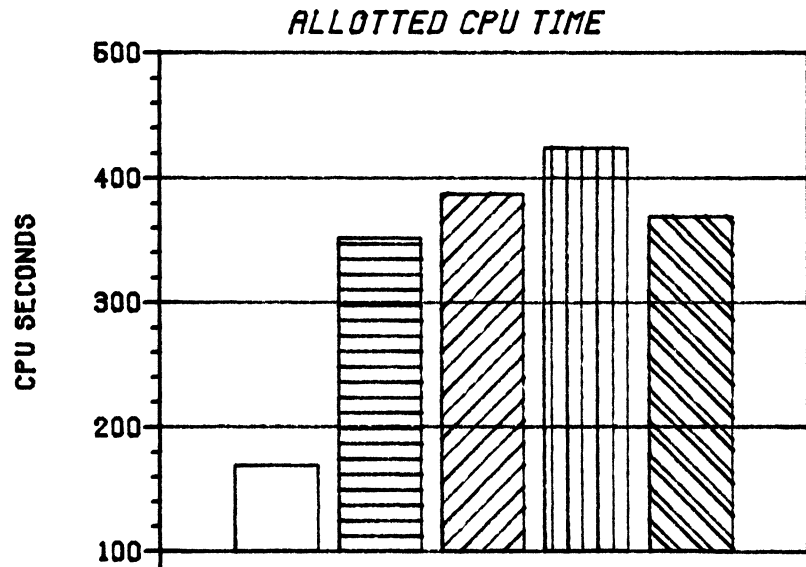


BENCHMARK - 90 Minutes (BMK31)

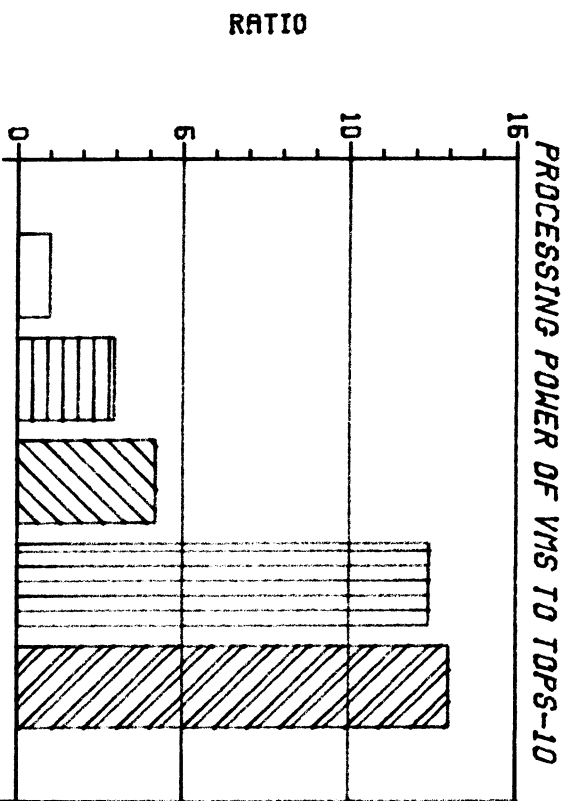
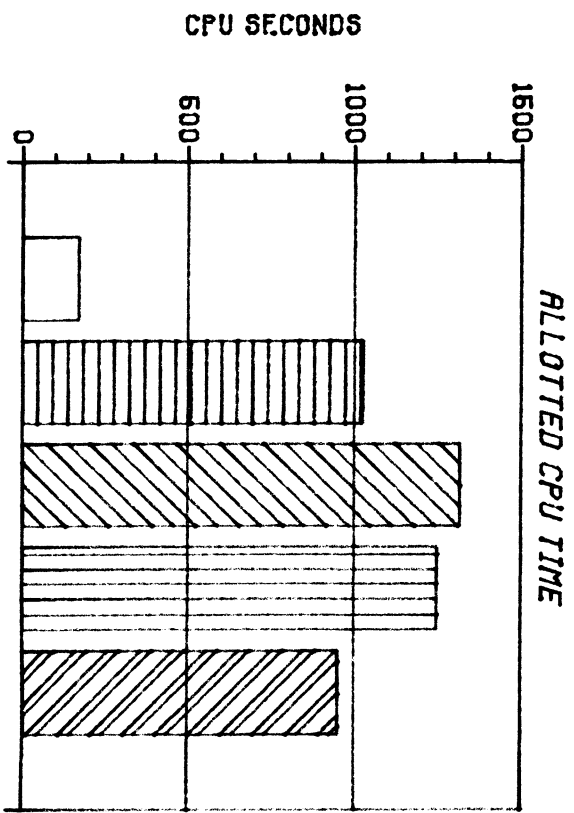
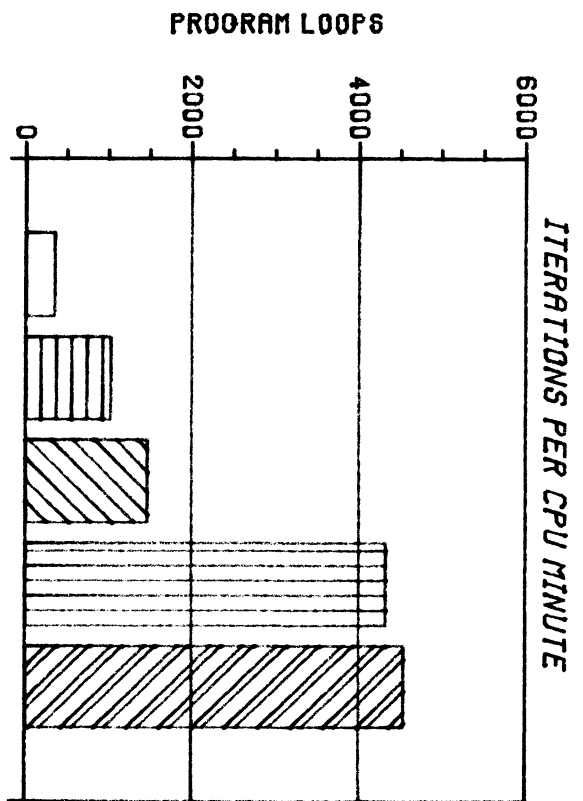
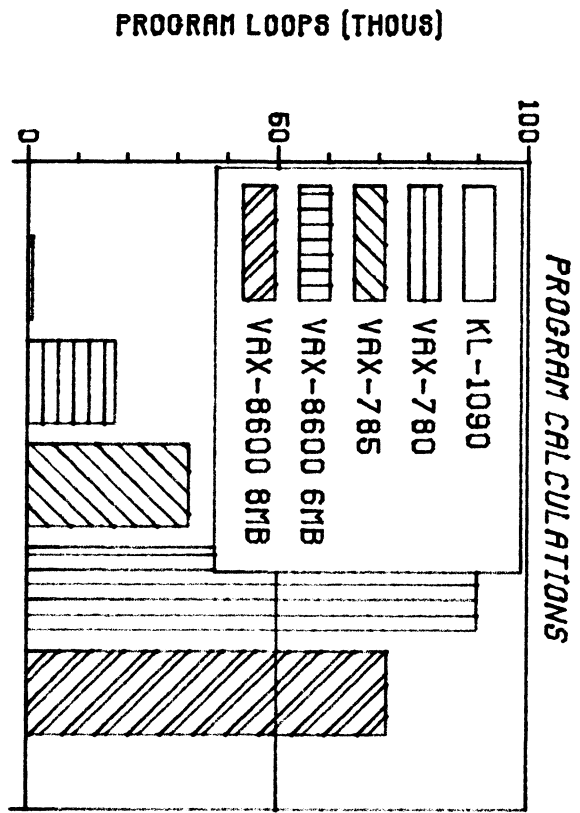


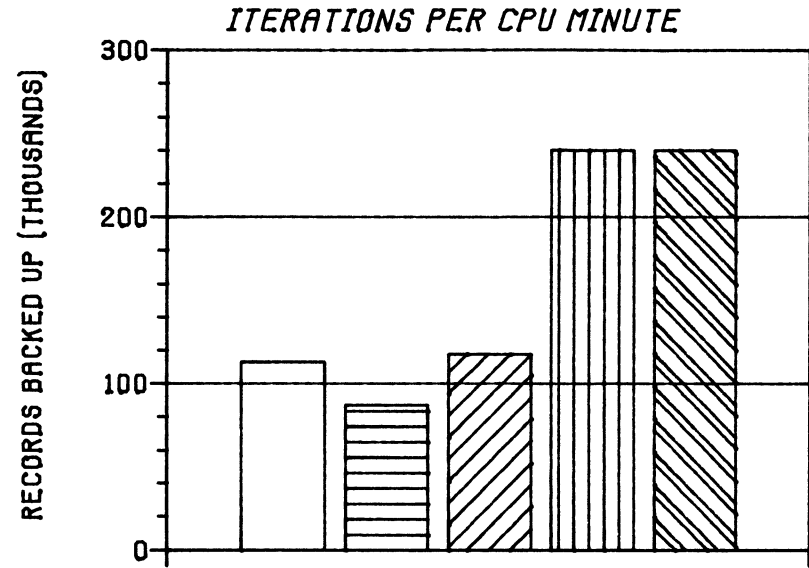
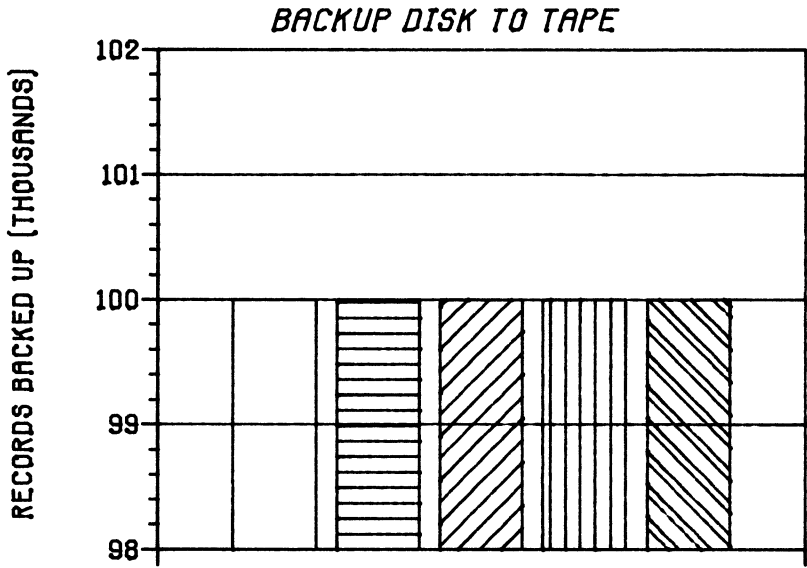


BENCHMARK - 90 Minutes (BMK32)

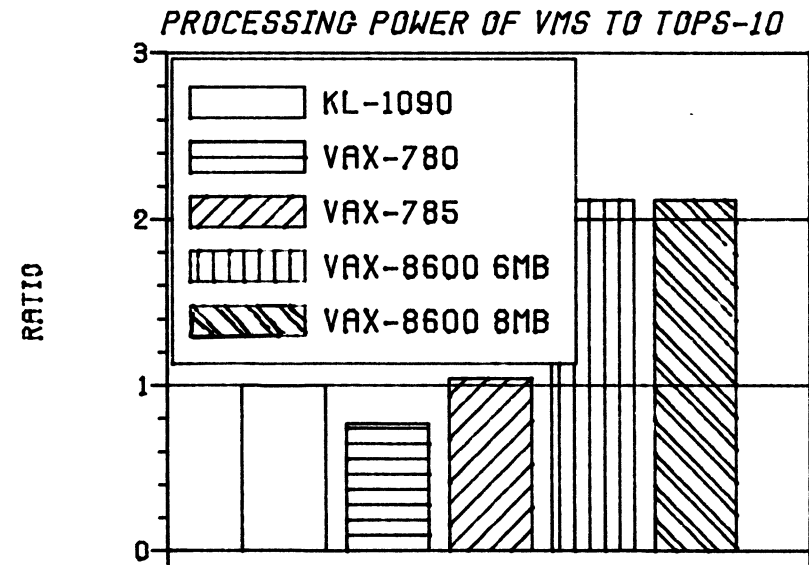
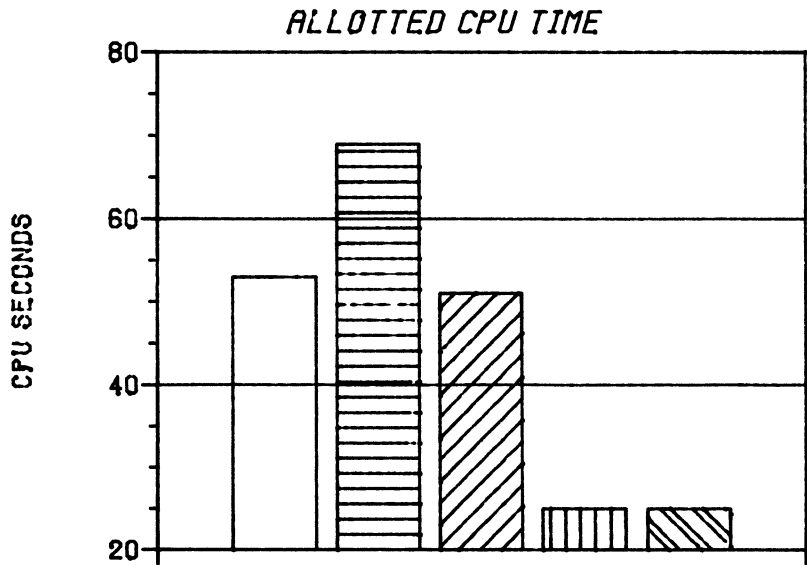


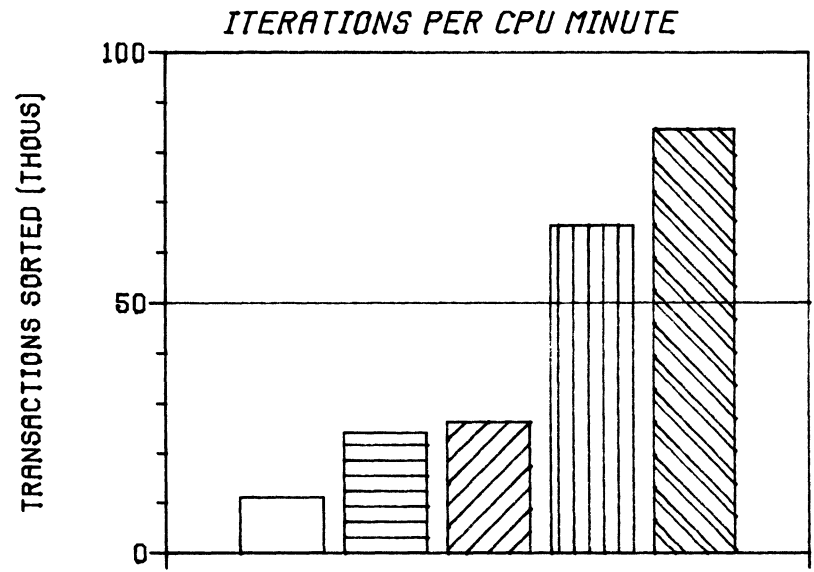
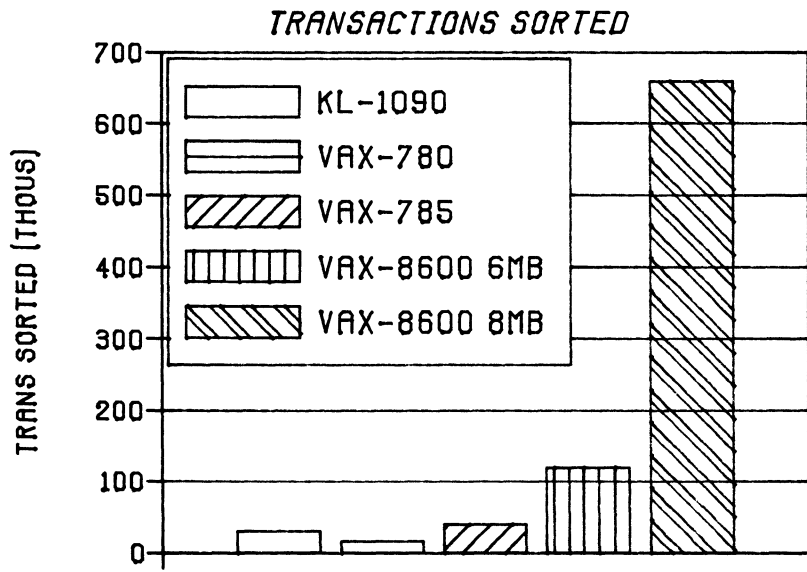
BENCHMARK - 90 Minutes (BMK21)



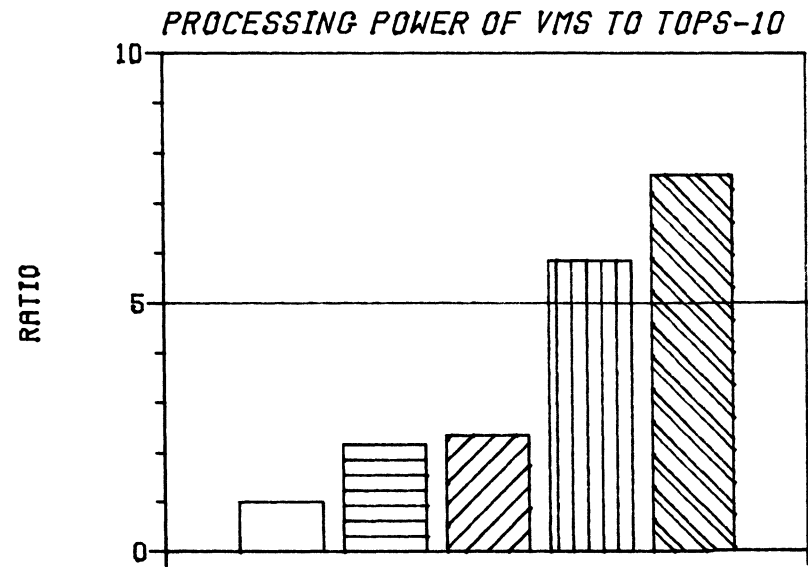
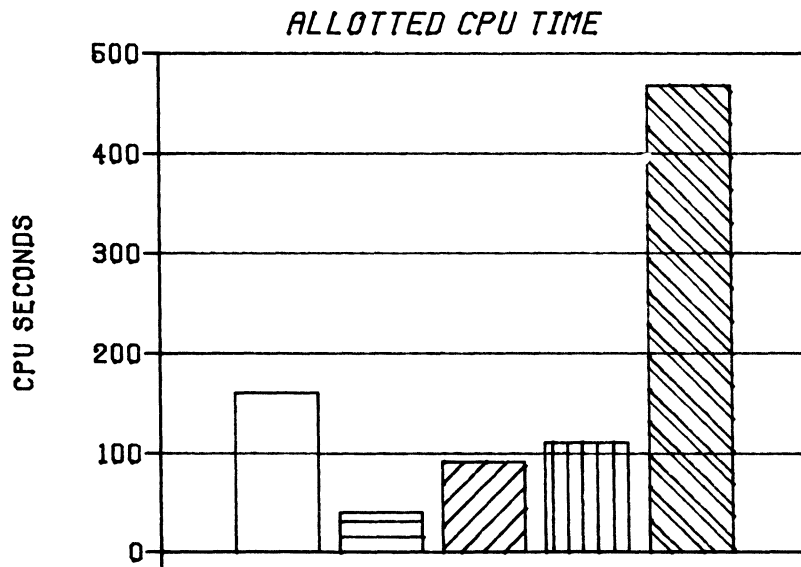


BENCHMARK - 90 Minutes (BACKUP)



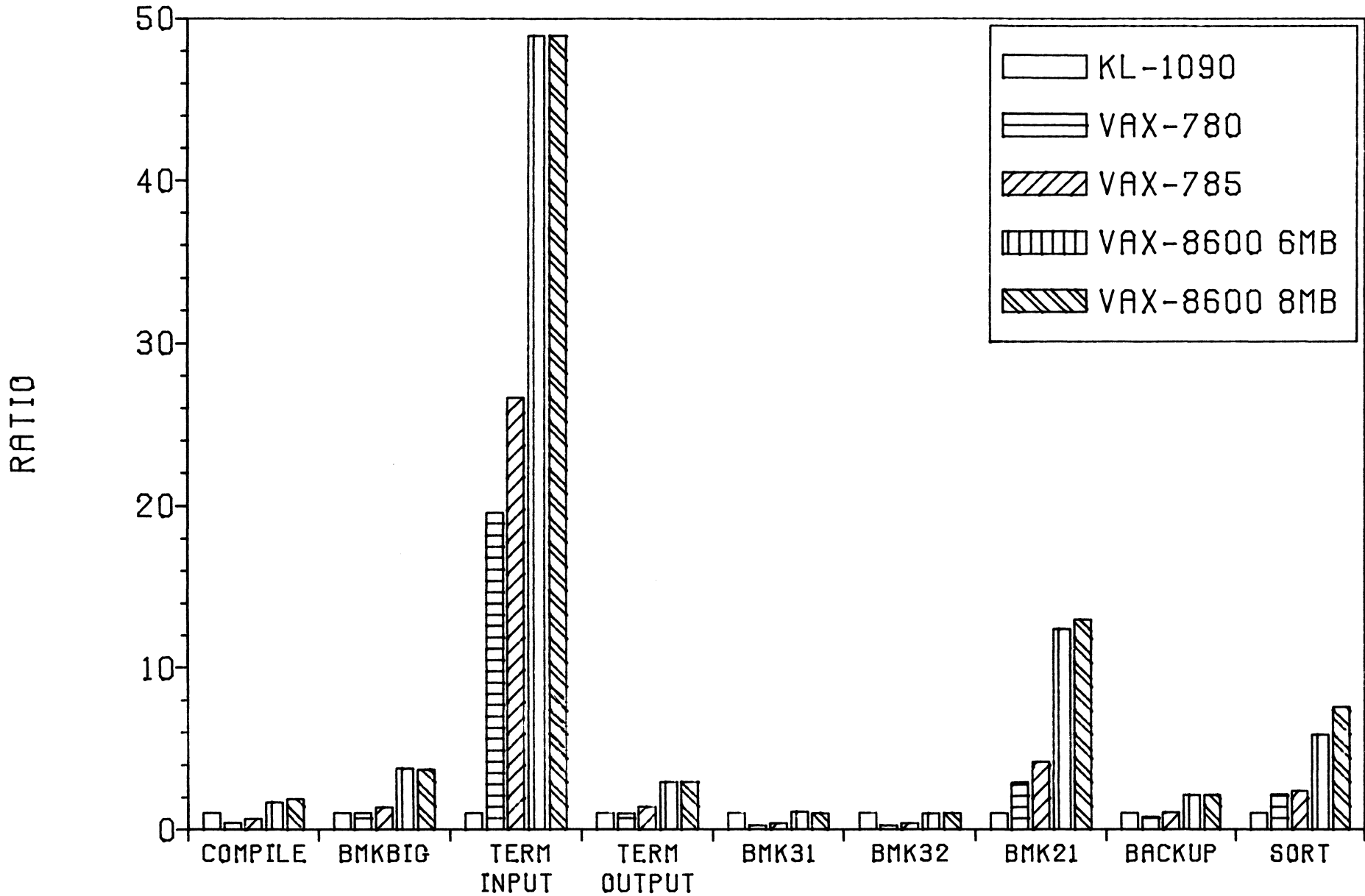


BENCHMARK - 90 Minutes (SORT)



FHLBB BENCHMARK

Processing Power of VMS to TOPS-10



Randolph M. Pacetti
 AT&T Technologies, Inc.
 Lisle, IL

ABSTRACT

Peter Samson of Systems Concepts, Inc. provided an overview of the history of the LISP programming language on 36-bit systems and new developments in the field.

The first publicly-available LISP was on a 36-bit machine, an IBM 704, and was called LISP 1.5. This was developed at MIT in 1958. LISP 1.0 lacked a great many features, and was never released to anyone outside the development group. LISP 1.5 set the standard for LISP for a long time. The manual is still in print and available in bookstores today. Many of the syntactic peculiarities of LISP were handed down from this original program. Users interacted with LISP 1.5 through the O26 card punch, which had a very limited character set: uppercase letters, numerals, and about a dozen punctuation marks.

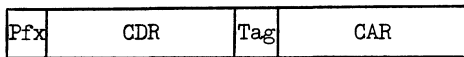


Figure 1

The data format illustrated in Figure 1 is that of a CONS cell, or LISP node. LISP 1.5 used a 36-bit word as a CONS cell. The IBM 704 divided its 36-bit word into four fields; from left to right these were called the prefix (3 bits), the decrement (15 bits), the tag (3 bits), and the address (15 bits). The two 15-bit fields were used in the implementation of LISP 1.5 as two data pointers known as the CDR and the CAR. They were 15 bits because that was all the machine could address.

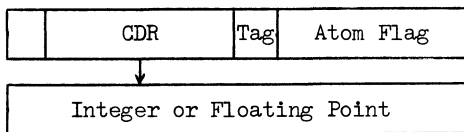


Figure 2

In certain cases the tag field was used by the LISP interpreter, as in Figure 2, which shows how a number was stored. In this case the CDR points to a memory location containing the number, the CAR field contains a flag indicating that the data is an atom, and the tag field distinguishes between various types of atoms (literal, integer, floating point).

At MIT, someone wrote a LISP interpreter for the PDP-1, which was an 18-bit machine with 12 bits of address. It worked, but it was essentially useless because of the very small address space available and the fact that it took two words per node. Several efforts were made to have virtual memory going off to a disk, but this was unspeakably slow. The idea of having an interactive LISP where you could type in

definitions or function calls and get responses immediately served as an inspiration for the first LISP on DEC's first 36-bit machine, the PDP-6.

The PDP-6 was designed, in part, with LISP in mind. The 36-bit word is processed in half words by a large class of instructions. Because the machine could address 18 bits of physical memory, having CAR and CDR occupy one 36-bit word was the intention right from the start. PDP-6 LISP was written in large part next door to the Tech Model Railroad Club at MIT in 1965-66. The 256K addressability of the PDP-6 was used up by LISP right away. This is a situation which hasn't changed since.

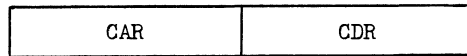


Figure 3

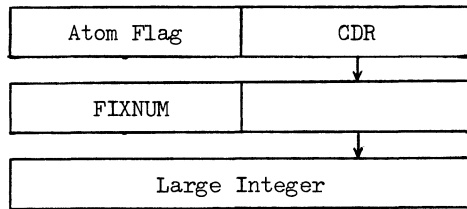


Figure 4

The data format of PDP-6 LISP is shown in Figure 3, with the CAR having moved to the left half of the word. Figure 4 shows how a large integer is expressed in the machine. This is not as compact as the representation on the IBM 704, where the tag bits were available. Another representation of numbers, the INUMs or small integers, could be encoded right in the pointer so that numbers of plus-or-minus a couple of thousand could be represented without further storage.

PDP-6 LISP has a great many offspring. An early copy was sent to Stanford, where they got many bugs out and released a cleaned-up version with the name LISP 1.6 in 1968. The University of California/Irvine made some additions to LISP 1.6 and released UCI LISP in 1973. Rutgers took over maintenance of UCI LISP and released R/UCI LISP in 1978. Another version was released as T/UCI LISP from the University of Texas.

36-bit LISP development also stayed at MIT, continuing on the PDP-10 and now given the name MAC LISP.

MAC LISP is different from PDP-6 LISP in that it was tuned for speed while PDP-6 LISP was tuned for space. MAC LISP got a great many enhancements in performance out of user demand, particularly in numeric operations, making it in many ways a non-standard LISP. On PDP-10-compatible machines it is one of the fastest LISPs available.

A totally independent line of work producing LISPs on PDP-10-like systems was started at a company called Bolt, Beranek, and Newman in Cambridge. MIT's PDP-1 LISP was copied to BBN's PDP-1, where they made some improvements but didn't fully succeed in making a reasonable system. This was copied to an SDS 940, and was then rewritten from scratch for the PDP-10 as BBN LISP around 1970. In 1972, BBN entered into a joint agreement with Xerox which took BBN LISP, added a great deal to it, and called the result INTERLISP. One way to tell this line of development from that stemming from PDP-6 LISP is that BBN put the CDR in the left half of the word, as was done on the IBM 704. One nice feature internal to BBN LISP is that large integers and floating point numbers didn't require a special atom header.

Type	CAR
Type	CDR
Type	Integer

Figure 5

In 1982, LISP was developed on the S-1 36-bit multi-processing supercomputer at Lawrence Livermore National Laboratories. If you have your choice of hardware, this is one of the fastest LISPs around. As shown in Figure 5, the data format consisted of 5-bit type fields and 31-bit address fields. Several types are reserved for integer, so you can get a 32-bit integer in the space of one pointer. It also has a tremendous number of other kinds of objects.

Type	CAR
Type	CDR
Type	Integer or Floating Point

Figure 6

A descendent of the Rutgers version of UCI LISP is ELISP, or Extended Addressing LISP, which uses the full theoretical 30-bit addressing of the extended addressing KL-10. The high order 6 bits are reserved for type information, as shown in Figure 6. Some types are classed together to get 32-bit integers and 32-bit immediate floating point in the space of one pointer. ELISP is a fully multi-sectioned program.

While all this proliferation of LISPs was going on, an effort was being made under the sponsorship of the Department of Defense to converge the various LISPs into one version that would run without conversion on a number of different machines. Common LISP is considered to be more the union of all possible LISPs than the intersection, and is therefore a rather large language. It derives more from MAC LISP than

from BBN/INTERLISP or Rutgers/UCI LISP. There is a version of Common LISP for PDP-10-like machines, called CLISP, from Rutgers. It uses the same data formats as ELISP, but is a bit slower because of what Common LISP requires of the interpreter and function call mechanism.

The next line of development is specific hardware for LISP. Xerox at Palo Alto made a PDP-10-compatible machine called Maxc and wrote LISP microcode for it. They wrote a few LISP-specific for it which sped up INTERLISP by 25%. They also had something called BYTE LISP, in which the LISP compiled code was in 9-bit bytes, which were opcodes that they interpreted for an additional 15% performance improvement.

In 1978, MIT built 32-bit LISP machines which interpreted LISP in the hardware. The first machine was called simply "the LISP machine" and the second was called "CADR." Two companies spun off from the MIT lab: LISP Machines, Inc. and Symbolics. LMI came out with the 32-bit LM-2. The Symbolics 3600 is a 36-bit machine which uses an 8-bit type field and a 28-bit address.

Systems Concepts is currently developing SC LISP for their PDP-10-compatible SC-30M computer. SC LISP is CLISP with hardware interpretation of the type codes.

TOPS-20 Directions

Donald A Kassebaum
University of Texas at Austin Computation Center
Austin, TX 78712

Abstract

Diane Lorion, Tops-20 Product Manager, and David Braithwaite from Digital Equipment Corporation presented an update on TOPS-20 directions.

Engineering Topics

- o Message from TOPS-20 Developers
- o Engineering's View of V6.1 Impact
- o Performance Expectations For V6.1
- o Near Future (Autopatch)
- o Maintenance Goals and Progress
- o Planning for V7.0

Message from TOPS-20 Developers

- THANK YOU
 - o Patience
 - o Quality of SPRs
 - Helpful Suggestions
 - Amount of Analysis
 - o Feedback From UPR Response Cards
 - o Quality of Field Test Relationships
 - o Positive Attitude During Critical Problems
- TOPS-20 V6.1 Availability
 - o IT'S ABOUT TIME !!!!
- LATE KITS (SORRY)
 - o Available January
 - o QT029 — RSX20F Source Pack
 - o QT102 — Combined Sources (Monitor, Exec, RSX20F)

Engineering's View of V6.1 Impact

- Based on Customers Information:
 - o V5.4 Customers
 - o V6.0 Customers
 - o V6.1 Field Test Customers
 - o V6.1 Pre-Release Customers
- HSC50 Disks
 - o Trend to Replace Massbus disks
 - o Multiple HSC50's for failure
 - o Increased disk space - Mostly RA81's (not RA60's)
- Common File System
 - o Customers Move to CFS
 - Quickly
 - Fully (painful fallback)
 - o As many systems (up to 4) as possible

- o All structures visible from all systems
- o All accounts on all systems
 - Users seek system with best response
 - LAT groups used to load-balance
 - Users move quickly if system crash
- DECnet on NI
 - o KL used as End Node
 - o DN20's Being Eliminated
 - o Increase in Number of Customers Using DECnet
 - o CTERM Chosen Over NRT for Terminals
- LAT on NI
 - o Terminals Being Eliminated From Front-End
 - o One Group-Name for All Systems in CFS Cluster
 - o Replacing other Electronic Switches
- TCP/IP
 - o Being used in Mixed Vendor Sites
 - o Only A Few Customers Beyond "ARPA"
 - o Not much Feedback
- MCA25/MG20
 - o MCA25
 - o Being Used to Buffer Transition to V6.1
 - o Providing 10%-20% Performance Increase
 - o MG20
 - o Popular in Sites With Tight Floorspace
- Other TOPS-20 V6.1 Features
 - o New EXEC Features - no information
 - o Password Encryption - most customers using it
 - o Other V6.1 Features - not much feedback
- Reliability
 - o Lo-o-o-o-ng Uptime
 - o Bigger Impact of Disk Hangs in CFS Environments
 - o Most CFS Related Problems With More Than 2 KL's

Performance Expectationations For V6.1

- Planning Guidelines
 - V5.1 (AP 10) to V6.1 — down 8%
 - V6.0 to V6.1 — up 4%
- Customer Feedback on V6.1 Performance
 - Mixed
 - Worst Performance is Specific Applications:
 - Heavy Random Access to Long Files
 - Frequent OPEN/CLOSE Sequences
 - More Even Response to Increased Load
 - Several Sites Believe Performance is Better
 - Benchmarks Show Mixed Results
 - Further Discussion at TOPS-20 USER PANEL

Near Future (Autopatch)

- Autopatch Tape 12 - Availability
 - Undergoing Final Verification
 - Expected Availability — Spring 86
- Autopatch Tape 12 - SPR Fixes
 - SPR fixes since Tape 11
 - Problems fixed after July 85
- Autopatch Tape 12 - Common File System
 - 3 and 4 System CFS Cluster Supportable
 - CFS Token Caching (Performance)
 - Elimination of Known System Hangs (Bugs)

Maintentance Goals and Progress

- SPR Backlog

250			
xxx			
xxx			
xxx	216		
xxx	xxx		
xxx	xxx	180	
xxx	xxx	xxx	
xxx	xxx	165	xxx
xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx
<hr/>			
Nov 84	May 85	Sep 85	Nov 85
- Customers Waiting

141			
xxx			
xxx			
xxx	121		
xxx	xxx		
xxx	xxx	102	
xxx	xxx	xxx	xxx
xxx	xxx	97	xxx
xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx
<hr/>			
Nov 84	May 85	Sep 85	Nov 85
- Goals For Next Year

- Resolve ALL CURRENT SPRs
- Resolve any SPR within 6 Months
- Resolve most SPRs within 2 Months
- Reach this level of Service During 1986
- Future Plan
 - Continue Maintaining V4.1 for KS Customers
 - KL Maintenance Base if V6.1
 - Performance Improvement
 - Reliability Improvement

Planning for V7.0

- Planning for TOPS-20 V7.0 Target Schedule
 - Finalize Requirements — Winter 86
 - Available for Field Test — Winter 87
 - Available to Customers — Fall 87
- TOPS-20 V7.0 OBJECTIVES
 - Complete Commitment
 - Increase Reliability (Up Time)
 - Enhance Performance
 - Enhance Maintainability
 - Minimize External Change
 - Minimize Impact of Upgrade
- Complete Commitment
 - 3 and 4 System CFS
 - Domains and EGP for ARPA Customers
- Increase Reliability
 - Forced Dismount of Hung Disks
 - Better Freespace Management
 - More Consistent Error Handling Under Evaluation
- Enhance Performance
 - Microcode
 - Buffering SYSERR Packets
 - Caching of Directory Index Pages
 - Pre-made Internal Memory Packets
 - HSC50 Disk CPU Time Under Evaluation
- Enhance Maintainability
 - Increase Error Detail
 - Find and Eliminate Kludges Under Evaluation
- Minimize External Change
 - Most Change Will Impact System Manager
 - Little Need For Changed User Documentation Under Evaluation
- Minimize Impact of Upgrade
 - No New Procedures
 - Simplified Packaging
 - Simplified Autopatch Mechanism Under Evaluation

TOPS-20 V6.1 for Users

Carla J. Rissmeyer
University of Texas at Austin Computation Center
Austin, TX 78712

Abstract

This session featured a presentation by DEC Software Engineers Jim McCollom and Bill Melohn about changes to the Exec and user impressions of the CFS environment. Also discussed were LAT and CTERM.

CHANGES TO THE EXEC

Besides many new commands, the V6.1 Exec has two major new features. The Exec now does multi-forking. As an example of this, a user could run a program, control-C out of it and keep it in order to continue it later. The Exec's second enhancement is the addition of ephemeral programs. Programs run ephemerally look like commands to the user.

Commands

INFORMATION FORK-STATUS shows the fork structure of a job

COMPILE /STAY compiles, keeping job at Exec level

CONTINUE continues a fork

- /BACKGROUND signals terminal for program input
- /NORMALLY continues normally
- /STAY used for programs which do not need input

START starts a fork

- /BACKGROUND signals terminal for program input
- /NORMALLY continues normally
- /STAY used for programs which do not need input

FORK specifies current fork

KEEP keeps a fork for the job

UNKEEP removes a kept fork

FREEZE temporarily holds a fork

RESET resets specified fork(s)

SET NAME sets name of the current fork

ERUN runs a program ephemerally

SET FILE [NO] EPHEMERAL sets or removes ephemeral status

SET DEFAULT PROGRAM [NO-]

- EPHEMERAL sets or removes ephemeral status default
- KEEP sets or removes kept status default
- NONE removes all defaults

INFORMATION DEFAULT PROGRAM shows defaults for program

SET PROGRAM [NO-]

- EPHEMERAL sets or removes ephemeral status

- KEEP (and) CONTINUE sets to continue when recalled
- KEEP (and) START sets to start when recalled
- KEEP (and) REENTER sets to reenter when recalled
- NONE clears defaults

^ESET [NO]

- FAST-LOGINS-ALLOWED takes no .cmd files on login
- LEVEL-ONE-MESSAGES enables level one messages
- LEVEL-ZERO-MESSAGES enables level zero messages
- WORKING-SET-PRELOADING enables working set preloads

SET [NO] TRAP JSYS

- /DEFINED sets trap for defined system calls
- /UNDEFINED sets trap for undefined system calls

TERMINAL

- [NO] INHIBIT prohibits receipt of all messages
- [NO] RECEIVE allows selection of types of messages
- VT200 set terminal type for VT200 series

TYPE UNFORMATTED does an image mode type

SET HOST runs CTERM server or program pointed to by NRT:

SET STATUS-WATCH allows settable char to type out file status

SYSTAT ORIGIN gives LAT information for each line

PERUSE runs EDITOR: with read-only option

PUSH runs program pointed to by DEFAULT-EXEC:

RECEIVE USER-MESSAGE receives for unprivileged TTMSG%

REFUSE USER-MESSAGE refuses from unprivileged TTMSG%

^ESEND now allows username instead of line number

LOGIN /FAST does not take .CMD files

LOGOUT now uses system and job-wide LOGOUT.CMD files

INFORMATION

- CLUSTER gives information on all CFS nodes
- DECNET [(nodename)]
- LOGICAL-NAMES now accepts wildcards
- JOB now outputs information on network host names

- o SUPERIOR displays number of the superior fork
 - o VERSION displays in decimal if VI%DEC is set
- ^ECEASE requires confirmation and adds NOW keyword
 COPY has SUPERSEDE [ALWAYS,NEVER,OLDER]
 subcommand
 DDT and MERGE now allow /OVERLAY switch
 DEFINE nows employs escape recognition
 DIRECTORY
- o COMPLETE includes all file information
- BUILD and ^ECREATE
- o PRESERVE leaves superior's quotas alone
 - o TOPS-10-PROGRAMMER-PROJECT-NUMBER
 - o PERMANENT INFINITY
 - o WORKING INFINITY
 - o Will not type out passwords

LAT

LAT stands for Local Area Transport. The LAT operates over ethernet. The TOPS-20 implementation does not require DECnet. LAT was designed to be used in the local environment, generally within one building, and was not made for use over bridges.

Characteristics of Communications Methods

Direct terminal communications:

- o Not flexible
- o One for one wiring
- o Use of multiple systems is difficult
- o Each system must be configured for maximum use

Plugboards:

- o Manual intervention is necessary
- o Limited flexibility
- o System side is often over-configured
- o More wiring than often needed

Electronic switches:

- o Intelligent switching
- o System side may be over-configured
- o Multiple wires
- o New switching problems
- o Pooling of modems across systems

Local Area Transports:

- o Intelligent switching
- o Minimal wire runs
- o No asynchronous ports on systems
- o Service groups of systems on ethernet
- o Cluster node addressing

CTERM

CTERM is the new DEC standard protocol which replaces the NRT protocol. It is layered on DECnet. At the present

time there are some incompatibilities with TOPS-20 to VMS communications.

THE CFS ENVIRONMENT

Perceptions

The user is generally unaware of the CFS environment. Multiple access to files and directories is similar to a single system. Some resources may be restricted.

Performance

Overhead in non-shared files is noticed only in opens or closes, as is overhead in read-only files.

Failure Modes

In disk failures, programs receive the same indications as they would with local disks. If feasible, disks will be accessed through other ports during a disk port failure. During a CI failure, there is no access to HSC disks, and access to dual-ported disks may be stopped by the monitor to avoid corruption. During a system failure, CFS restarts all votes and the systems which are not crashed continue normally.

Constraints

ENQ on a file works only on a single system. There is no reclaiming of inter-system PIDs. DBMS files opened with OF%DUD cause interlock problems.

COMMENTS FROM THE Q&A PERIOD

TTMSG% will not be going away.

A 6.1 Exec may be run with a 5.1 monitor. A 5.1 Exec may be run with a 6.1 monitor after one location is patched.

There are no plans for ATTACH or DETACH command files.

The LAT is down-line loadable from TOPS-20.

MIC, PCL and a command line editor are distributed (without support) on the tools tape.

Better documentation for MIC and PCL is needed before TOPS-20 is frozen.

Develop a mechanism for automatic mounting of and connecting to structures on login (non-PS: logins).

It would be nice to have a "save all except ----" tape command.

Patch LPTSPL to allow spooling to printing terminals.

TOPS-20 V6.1 for System Administrators

Douglas Bigelow
Wesleyan University
Middletown, Connecticut

Abstract

This session covered features in TOPS-20 monitor release 6.1 which are of particular interest to system administrators. Topics included system configuration file changes, new SETSPD options, new OPR commands, a complete DUMPER rewrite, and new GALAXY features.

Introduction

This session discussed the new features of the 6.1 monitor which were of importance to system administrators, involving changes to the operator interface, system backup performance improvements, changes to disk structure management, and DECnet Phase IV features. The speakers, all from Digital, were *Dave Braithwaite*, *Jim McCollom*, *Bill Melohn* and *Scott Mayo*.

General Features

New product features of TOPS-20 V6.1 include the following:

- NIA20 support.
- DECnet-20 Phase IV, V4.0 support.
- CTERM support for heterogeneous command terminals.
- DECnet Router and LAT terminal concentrator support.
- CI/CFS common file system support.
- Hardware support for MG20 memory, MCA25 cache upgrade, HSC50, RA81 and RA60.
- Maximum 4 Meg memory.

Software enhancements include password encryption, with a one-way encryption algorithm which can be replaced by a customer algorithm if necessary. Another optional security feature is password rejections, involving monitoring of failed login attempts.

RSX-20F will now autobaud from 110 to 9600 baud, and has had several other modifications for performance enhancement or new hardware support.

DECnet

Version 6.1 includes DECnet phase IV level 1 router support on both Ethernet and CI. Systems with no CI and no MCB can be an elective endnode on the Ethernet, at a considerable overhead savings and no loss in network access. Ethernet buffers have been expanded to 1450 bytes for greater throughput, and the end communications layer has been re-written with optimizations. NODES.EXE and MCBNRT have now been moved into the monitor.

There are several DECnet tools available, including the monitor utility *DNSNUP*, the packet printer *DNTATL*, *DCNSPY* to see monitor data structures, *NETPTH* to find packet paths and *RMTCON* to provide remote console access.

SETSPD has a number of new network oriented commands, including *Ethernet*, *LAT-State*, and *DECNET {Buffer-size, Default-Buffers, Default-Flow-Control, Maximum-Address, Minimum-Address, Router-Endnode, Router-Level-1}*.

SETSPD

SETSPD has several new features, including the ability to write monitor crash dumps to DMP: instead of to PS:<SYSTEM>. In addition, SETSPD is automatically run whenever a Massbus disk comes on-line (for MSCP.) New enable/disable options include *Fast-Login*, *Hangup-If-Logged-In*, *Hangup-If-Logged-Out*, *Job0-CTY-Output*, and system message level

control.

Additions for CI and MSCP include commands to regulate access to shared disks, namely *Allow*, *Restrict*, *DontCare* {drive-type serial-no}.

MOUNTR.CMD is no longer necessary, since the file SYSTEM:DEVICE-STATUS.BIN has replaced it's functionality.

DUMPER

DUMPER has had a complete re-write, including new documentation and the following improvements:

- Faster speed!
- Better ^E interrupt handling.
- Automatic detection of interchange tapes.
- Better error and information messages.
- Longer saveset names.
- Optional mail messages for several functions.
- Optional features may be assembled out to improve performance further.

Galaxy

New Galaxy commands include *Show Configuration*, *Set Port CI/NI {Un}Available*, *Set Structure Exclusive/Shared*. In addition, the new QUEUE% jsys allows user programs to communicate with Galaxy components. MOUNTR has changed or added several structure commands such as *Mount*, *Dismount* and *Undefine*.

TOPS-20 V6.1 for Systems Programmers

Douglas Bigelow
Wesleyan University
Middletown, Connecticut

Abstract

This session covered features in TOPS-20 monitor release 6.1 which are of particular interest to systems programmers. Topics included new JSYS calls, modifications to current JSYS calls, new DDT features, and new SYSDPY features.

Introduction

This session discussed the new features of the 6.1 monitor which were of importance to systems programmers, involving changes to the addressing space, new JSYS calls, and new versions of DDT and SYSDPY. The speakers, all from Digital, were *Dave Braithwaite*, *Jim McCollom*, *Bill Melohn* and *Scott Mayo*.

General Features

Many new extended memory sections are being used under 6.1, to accomodate among other things the additional data structures necessary for CI/CFS support. Multi-section support in general has been enhanced, with DDT in particular being better able to debug extended memory programs.

Increased network information and control was another major thrust, with new JSYS calls and many more displays in SYSDPY pertaining to network status.

New JSYS Calls

- **XPEEK**, an extended version of PEEK.
- **WSMGR**, for managing process working sets.
- **NTINF**, for general network information.
- **CNFIG**, for general hardware, software and CFS information.
- **SCS**, for CI communications status.
- **NI**, for Ethernet communications status.
- **LATOP**, for LAT control.

- **LLMOP**, for network interconnect remote console service.

In addition, changes were made to an additional 50 existing JSYS calls.

DDT Version 43

DDT was enhanced to provide better breakpoint control, with inter-section breakpoints provided. DDT is now loaded into the highest free section of memory and no longer interferes with the use of pages over 760. Non-zero section symbols may be used during debugging. Breakpoints may be set in any section, with n\$4M specifying the breakpoint block address in section n.

PDV support has been added, with commands to select PDVs by name or by address or to type the current name.

XDDT is the new version of DDT which uses it's own section. UDDT remains as a program stub to preserve the old DDT behavior when necessary. Single stepping has been enhanced, with \$X working in non-DDT sections.

Other new features include n-bit ASCII text support, byte pointer typeout mode via \$1T, symbol prefix searching, and \$1: to type the open module name.

Monitor DDT

- Monitor breakpoint blocks EDDBLK and MDDBLK are mapped to all code sections.
- Monitor PDV is at location MONPDV.
- All \$U commands work in KDDT.
- FILDDT is no longer able to patch the monitor.

All binary patching should be done using the normal sequence of GET; START 140; patches; SAVE.

SYSDPY

SYSDPY has a new version of the MONRD% jsys and the following new displays:

- ANCN: Show TCB for index n.
- ANG, ANN, ANT: Show Internet information.
- ARP: Show contents of ARP tables.
- DR: Show disk drive status.
- MS, MC: Show MSCP information.
- SC, SCn, SCD, SS: Show SCA information.

Ethernet Planning and Installation Considerations

Donald A Kassebaum
University of Texas at Austin Computation Center
Austin, TX 78712

Abstract

Dan Deuffel, Network Consultant for Large System Group, presented a session on Ethernet planning and installation considerations.

Mr. Deuffel presented and explained a set of slides, a copy of which is contained in this report. In terms of planning, topics such as Ethernet components, configuration rules and examples were discussed. Also reviewed were installation considerations, including common problems and diagnosis, maintenance aids and tips, and documentation sources. Use of the new thinwire Ethernet was included among the topics.

ETHERNET

- BRANCHING BUS TOPOLOGY
- 10 MEGABIT DATA RATE
- CSMA/CD
- MULTI-PROTOCOL
- ORIGINALLY BASEBAND
- BROADBAND IMPLEMENTATION AVAILABLE
- 802.3 STANDARD
- MULTI-VENDOR SUPPORT
- DIGITAL'S NETWORK INTERCONNECT

ETHERNET COMPONENTS

- ETHERNET CABLE
- ETHERNET TRANSCEIVER
- TRANSCEIVER CABLE
- ETHERNET CONTROLLER
- REPEATERS
- BRIDGES

ETHERNET CABLE

- CO-AXIAL - TEFLON COATED
- 23.4, 70.2, 117.0, OR 500 METER LENGTHS AVAILABLE
- MULTIPLE LENGTHS MAY BE JOINED

THINWIRE ETHERNET CABLE

- CO-AXIAL
- RG58

 DO NOT KINK

TRANSCEIVER CABLE

- IEEE 802.3 OR ETHERNET VERSIONS
- TEFLON OR PVC COATED
- 5, 10, 20, OR 40 METER LENGTHS
- STRAIGHT OR RIGHT ANGLE CONNECTOR
- USABLE WITH BOTH H400x AND DECOM

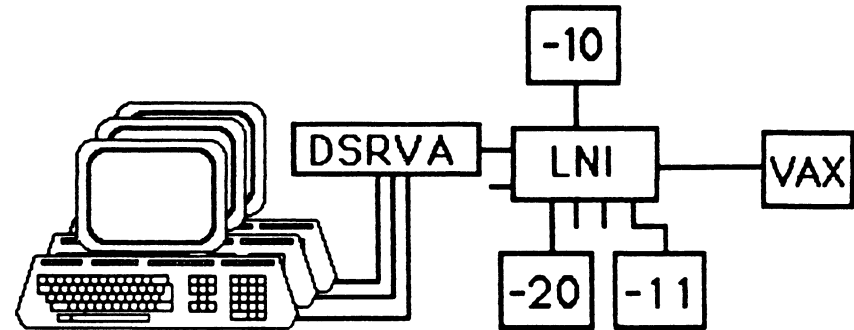
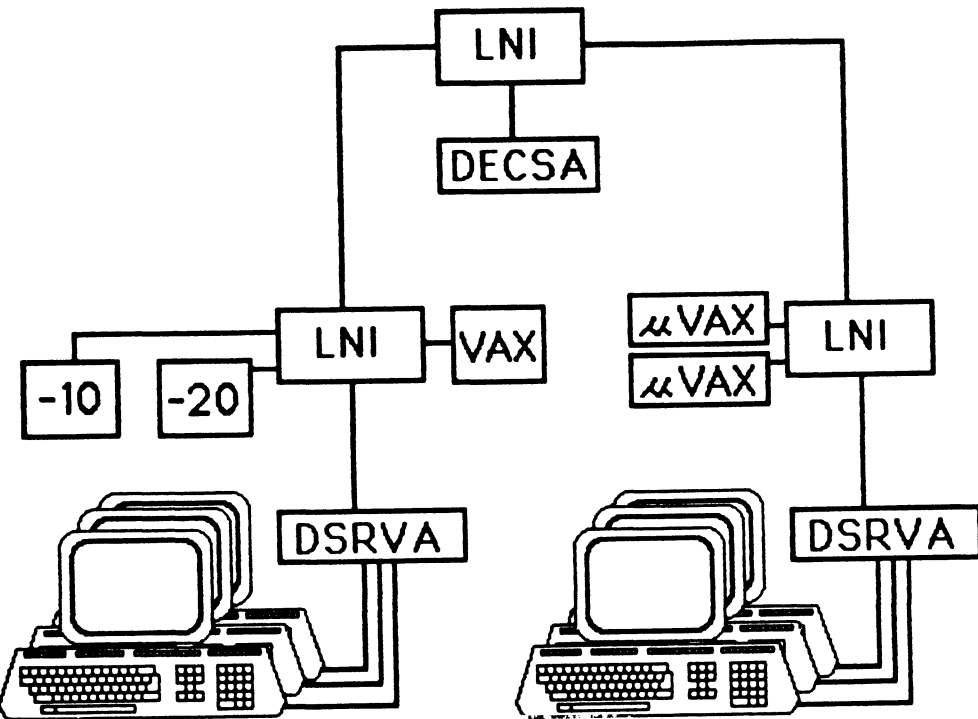
ETHERNET TRANSCEIVERS

- H4000 – BASEBAND TRANSCEIVER
 - ➔ VAMPIRE TAP
 - ➔ POWERED BY CONTROLLER
 - ➔ NO INSTALLATION DOWNTIME
- H4005 A/B – BASEBAND TRANSCEIVER
 - ➔ COMPATIBLE WITH EXISTING HARDWARE
 - ➔ IEEE 802.3 COMPATIBLE
 - ➔ AVAILABLE WITH HEARTBEAT (H4005-A)
 - ➔ OR WITHOUT HEARTBEAT (H4005-B)
- DESTA – THINWIRE STATION ADAPTOR
 - ➔ CONNECTS EXISTING ETHERNET INTERFACES TO A THINWIRE SEGMENT
 - ➔ CONNECTS TO T-CONNECTOR IN THE THINWIRE SEGMENT
- DECOM – BROADBAND TRANSCEIVER
 - ➔ USED WITH EITHER ONE CABLE OR TWO CABLE SYSTEMS
 - ➔ DEFTR – FREQUENCY TRANSLATOR FOR ONE CABLE SYSTEMS AVAILABLE

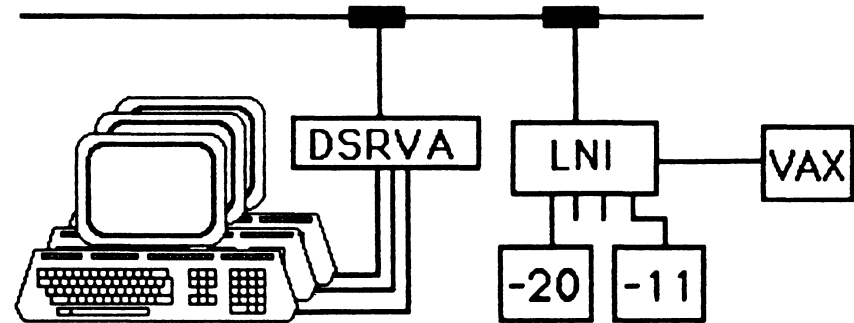
DELNI

- IN LOCAL MODE DELNIs MAY BE HOOKED TOGETHER TO CONNECT UP TO 64 NODES WITHOUT USING AN ETHERNET CABLE

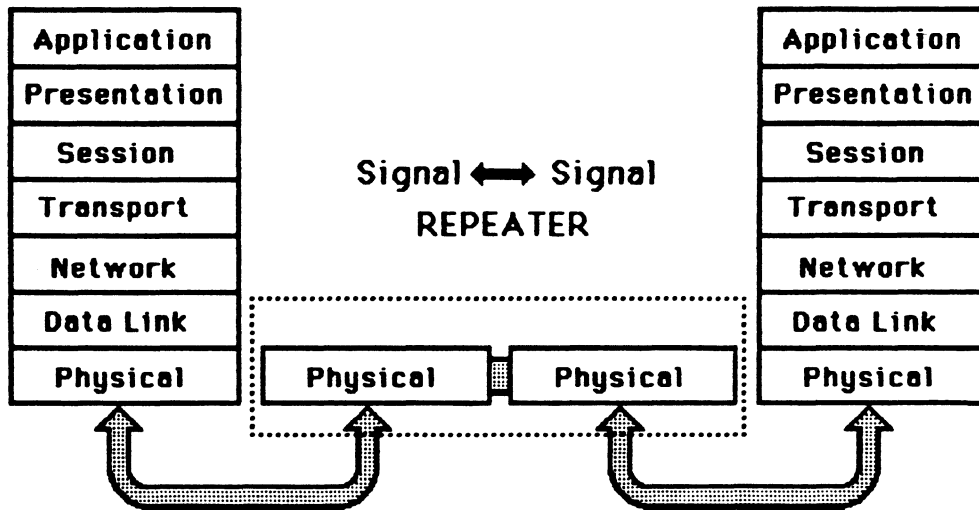
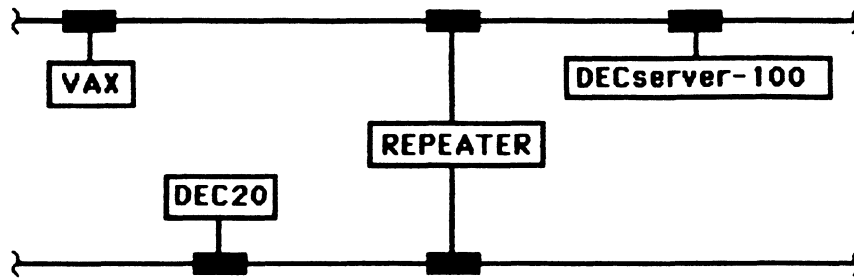
- CONCENTRATES 8 ETHERNET DEVICES
- OPERATES STANDALONE (LOCAL)



- CONNECTS TO CABLE (REMOTE)



REPEATER



REPEATERS

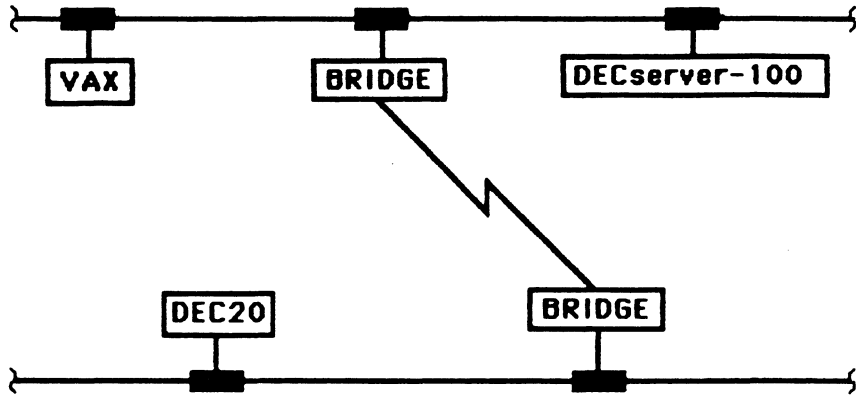
- ELECTRICALLY JOINS TWO CABLE SEGMENTS
- REGENERATES THE FRAME PREAMBLE
- DOES NOT MODIFY FRAME DATA

- THREE REPEATERS AVAILABLE
 - ➔ DEREP-AA
 - JOINS TWO LOCAL SEGMENTS
 - SMART (AUTO SEGMENT ISOLATION)

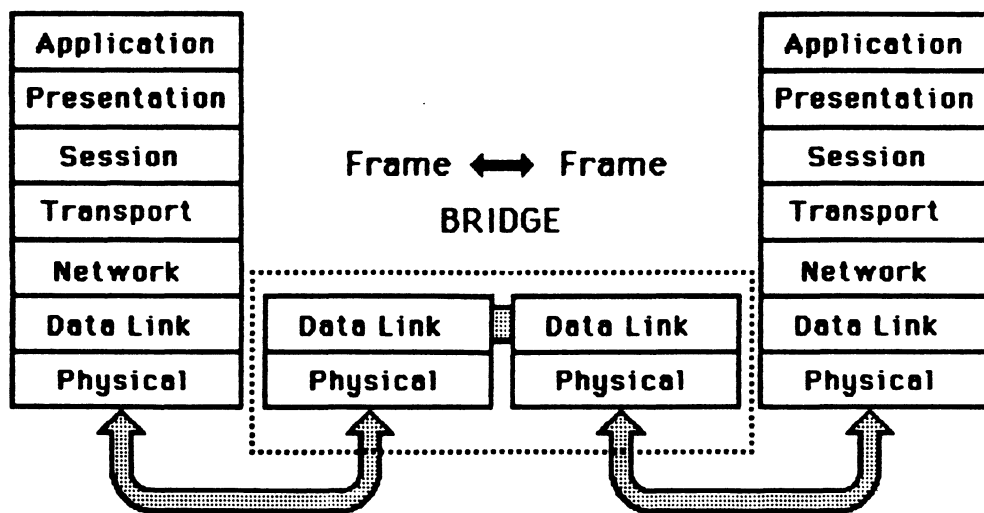
 - ➔ DEREP-RA
 - JOINS TWO REMOTE SEGMENTS
 - UP TO 1000 METERS OF FIBER OPTIC CABLE BETWEEN

 - ➔ DEMPR
 - JOINS UP TO EIGHT THINWIRE SEGMENTS
 - CONNECTS TO A BACKBONE SEGMENT

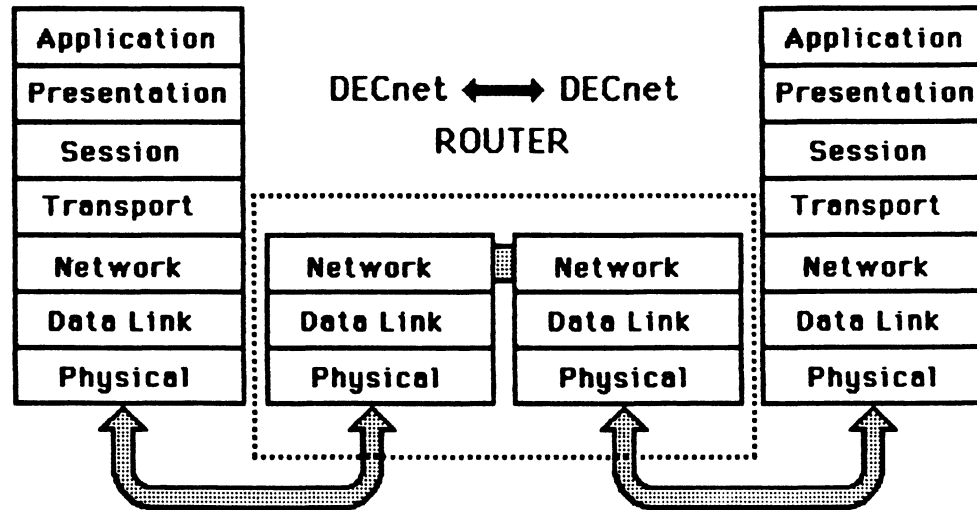
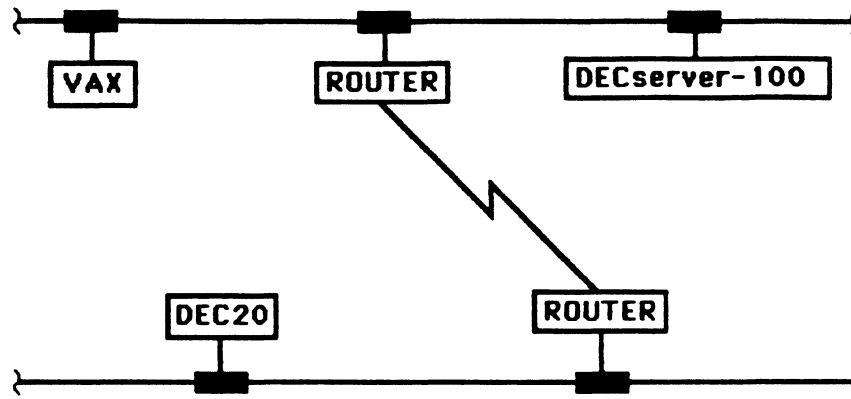
DEBET - LANBRIDGE 100



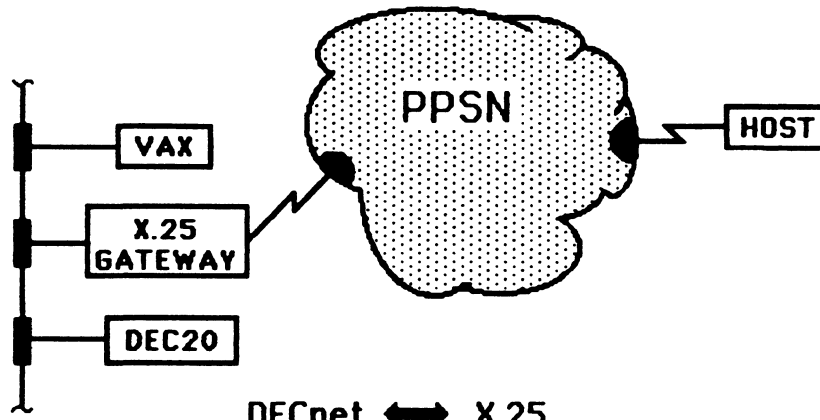
- EXTENDS ETHERNETS BEYOND 2800 METERS
- PROTOCOL INDEPENDENT
- STORE AND FORWARD
- PACKET ADDRESS FILTERING
- 802.3/ETHERNET COMPLIANT
- AUTOBACKUP CAPABILITIES
- TWO VERSIONS AVAILABLE
 - ➔ LOCAL BRIDGE (WITH TWO XCVR DROP CABLES)
 - ➔ REMOTE BRIDGE (WITH FIBER-OPTIC LINK)



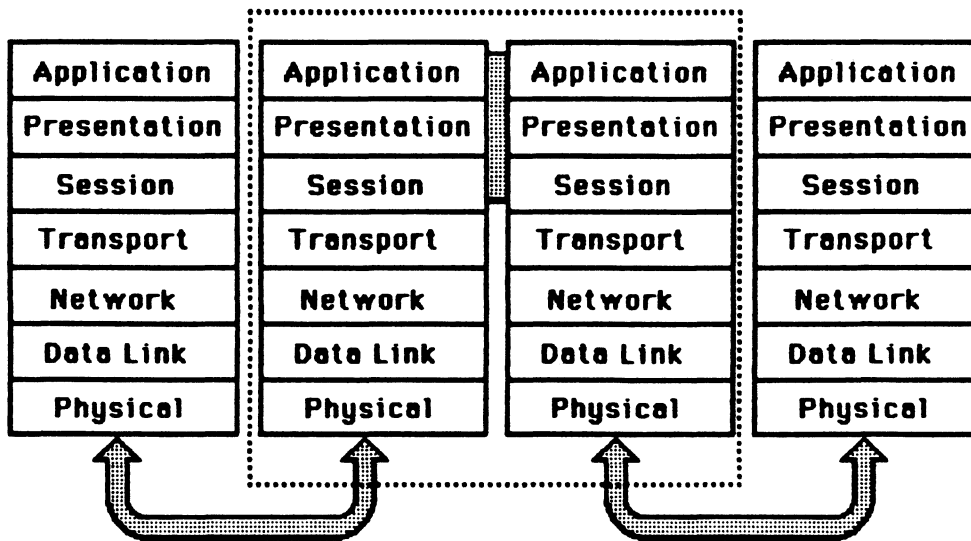
ROUTER



GATEWAY



DECnet ↔ X.25
GATEWAY

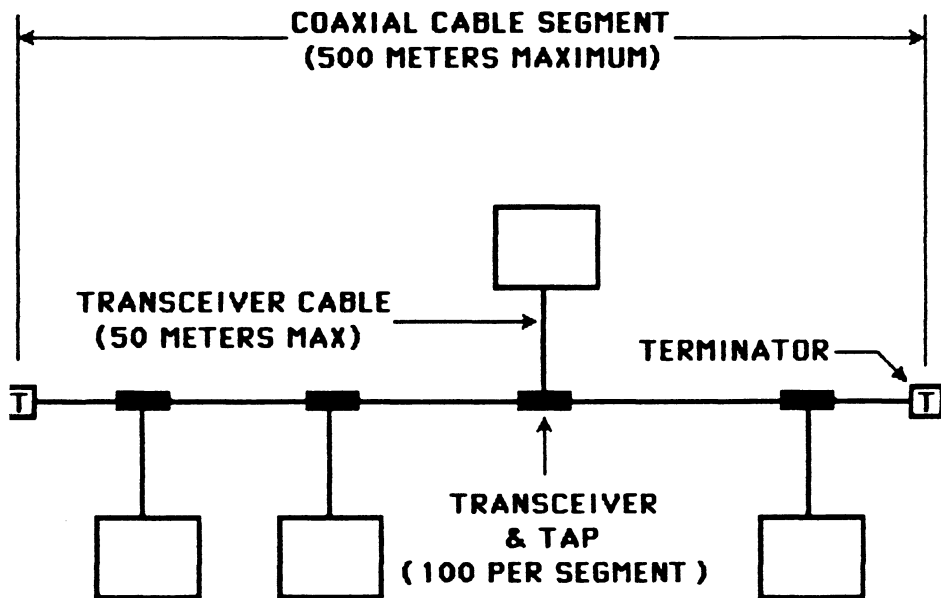


ETHERNET CONFIGURATION RULES

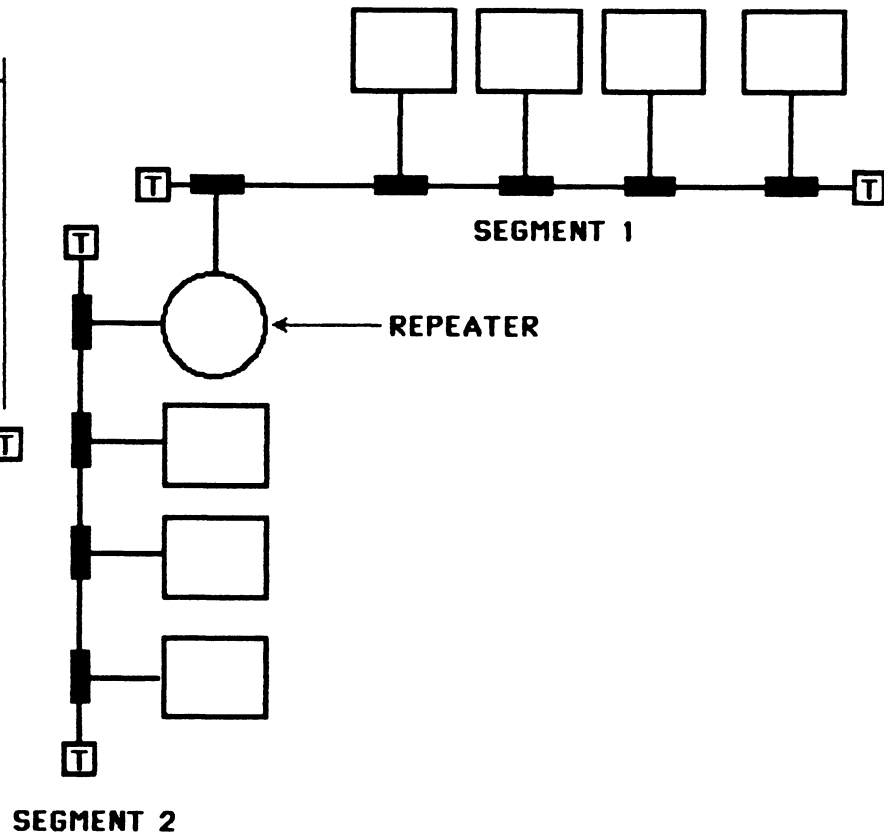
- 500 METERS MAXIMUM SEGMENT LENGTH
- MAXIMUM OF 100 TAPS PER SEGMENT
- 50 METERS MAXIMUM FROM TRANSCEIVER TO CONTROLLER
- 2800 METERS MAXIMUM END TO END DISTANCE
- MAXIMUM OF 2 REPEATERS BETWEEN ANY TWO STATIONS
- 2.5 METER TAP SPACING
- MAXIMUM OF 1024 STATIONS
- NO LOOPS ARE ALLOWED
- A DEREPA CAN'T BE CONNECTED TO A DELNI
- IF A DEREPA IS CONNECTED BY FIBER-OPTIC LINK TO A DEREPA, THE FIBER-OPTIC LENGTH COUNTS TOWARDS THE OVERALL LAN LENGTH
- MAXIMUM AGGREGATE OF 1000 METERS OF FIBER-OPTIC LINKS BETWEEN REPEATERS IS ALLOWED BETWEEN ANY TWO STATIONS

ETHERNET CONFIGURATIONS

A SMALL SINGLE SEGMENT CONFIGURATION

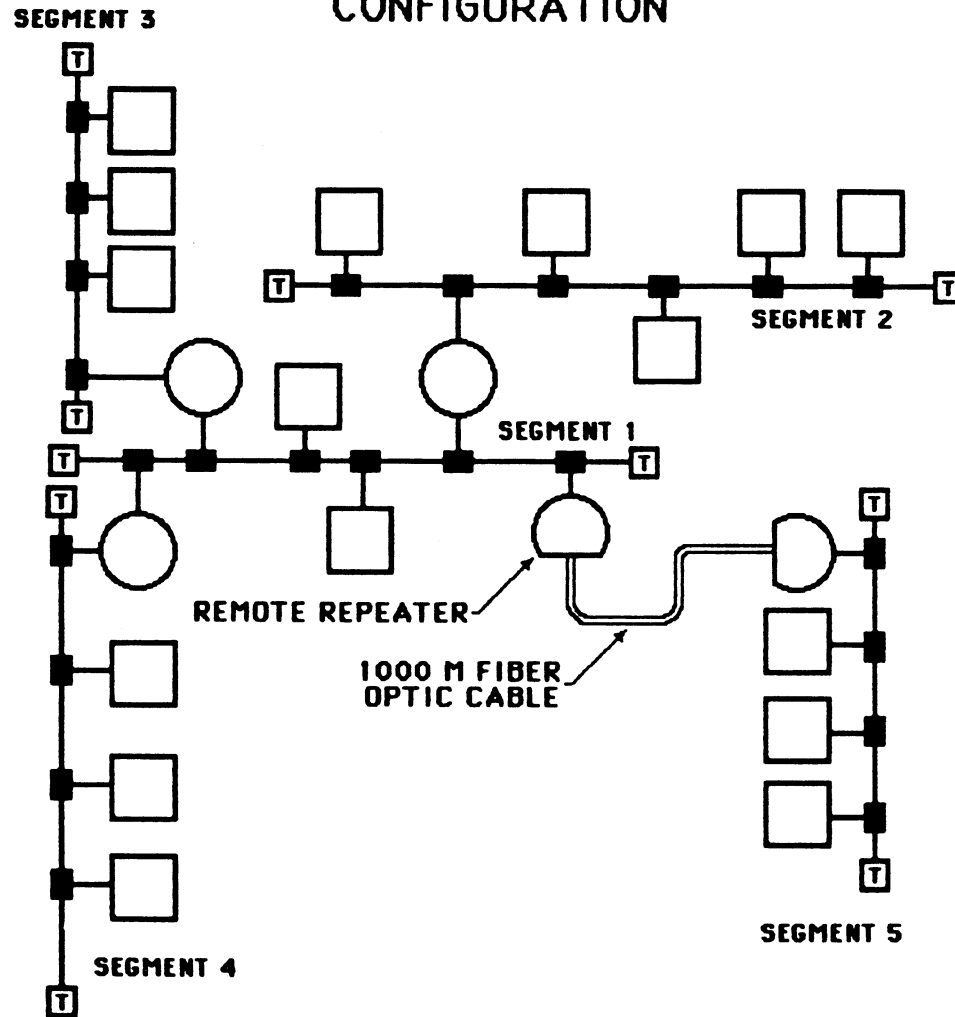


A MEDIUM TWO SEGMENT CONFIGURATION



ETHERNET CONFIGURATIONS

A LARGE FIVE SEGMENT CONFIGURATION



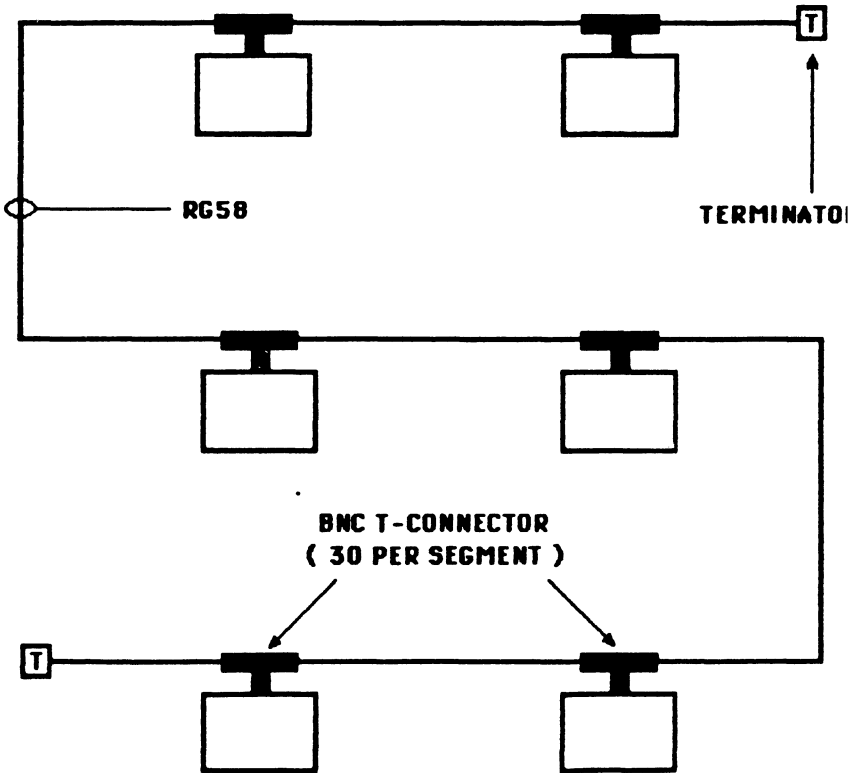
TRANCEIVER USAGE SUMMARY

- STATIONS MUST CONNECT TO H4000 OR H4005A
- DEBET MUST CONNECT TO H4000 OR H4005A
- DEREPR MUST CONNECT TO H4000
- DEMPR MUST CONNECT TO H4000 OR H4005B
- DENLI WITH DEMPR CONNECTED MUST CONNECT TO H4005B ONLY
- DELNI MUST CONNECT TO H4000 OR H4005A

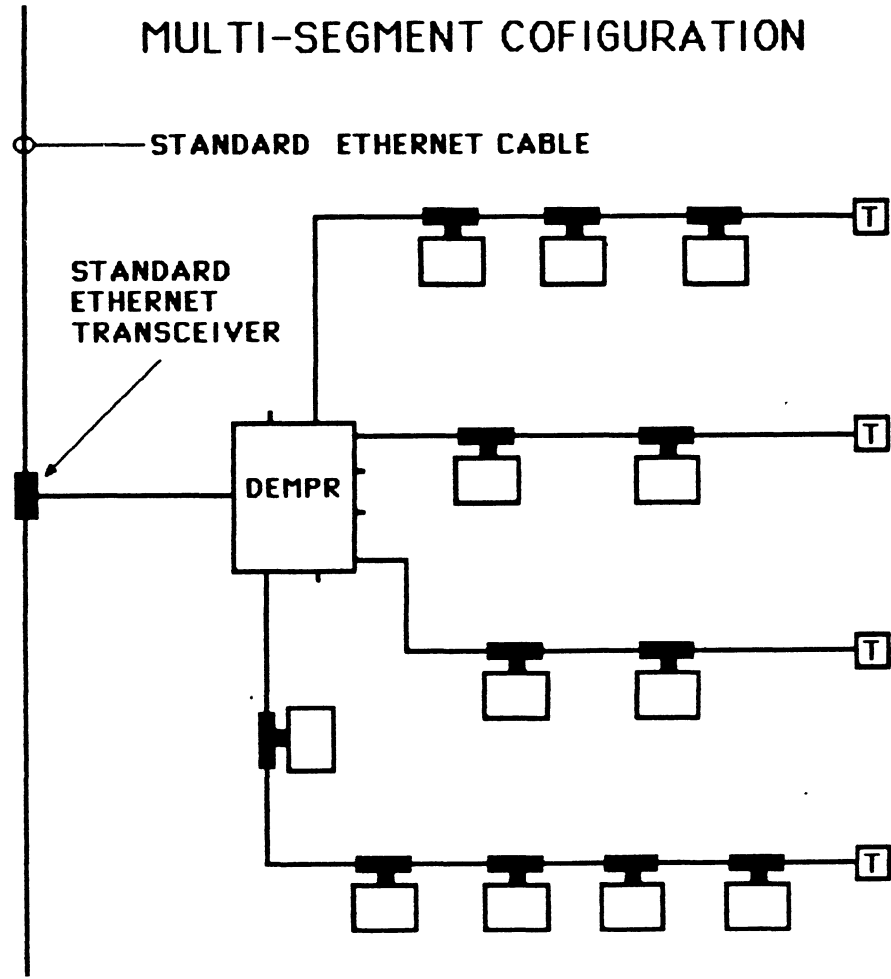
THINWIRE CONFIGURATION RULES

- 185 METERS MAXIMUM THINWIRE SEGMENT LENGTH
- MAXIMUM OF 30 STATIONS PER SEGMENT
- THINWIRE SEGMENTS CAN BE CONNECTED TOGETHER WITH A DEMPR
- DEMPRs MUST TERMINATE A SEGMENT
- MAXIMUM OF 2 REPEATERS BETWEEN ANY TWO STATIONS (DEREPs & DEMPRs EACH COUNT AS A REPEATER)
- STATIONS MUST BE CONNECTED DIRECTLY TO THE BNC T-CONNECTOR. NO CABLE IS ALLOWED BETWEEN THE STATION AND THE T-CONNECTOR

THINWIRE CONFIGURATION



MULTI-SEGMENT COFIGURATION

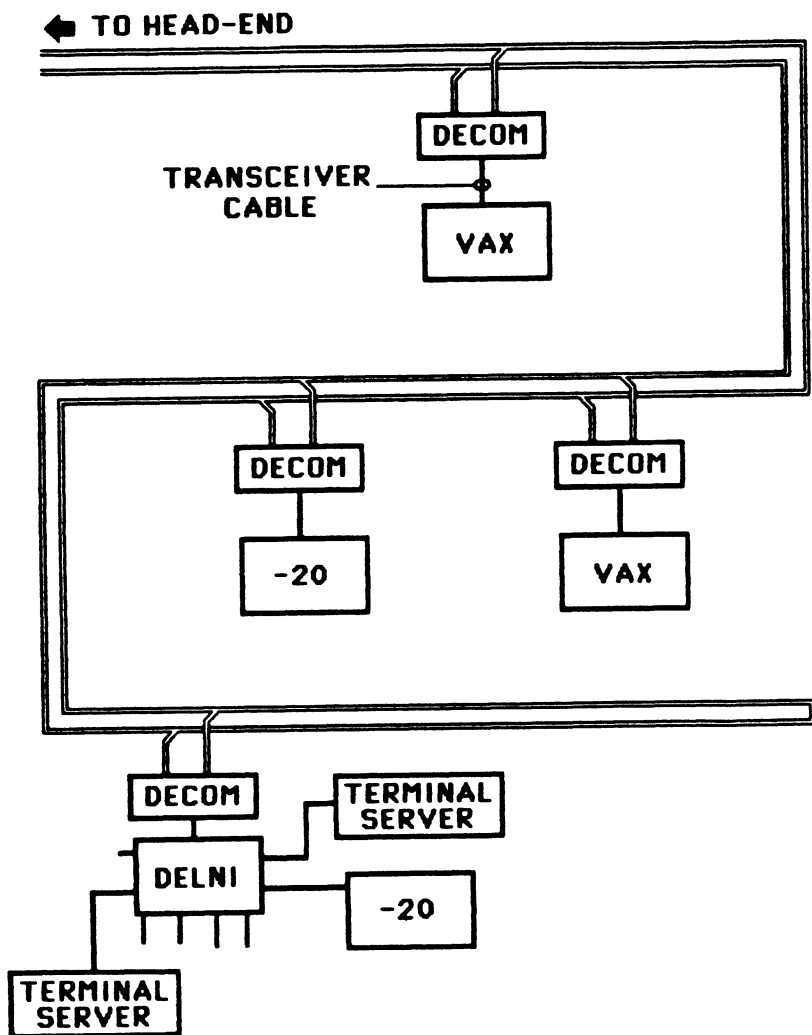


BROADBAND CONFIGURATION RULES

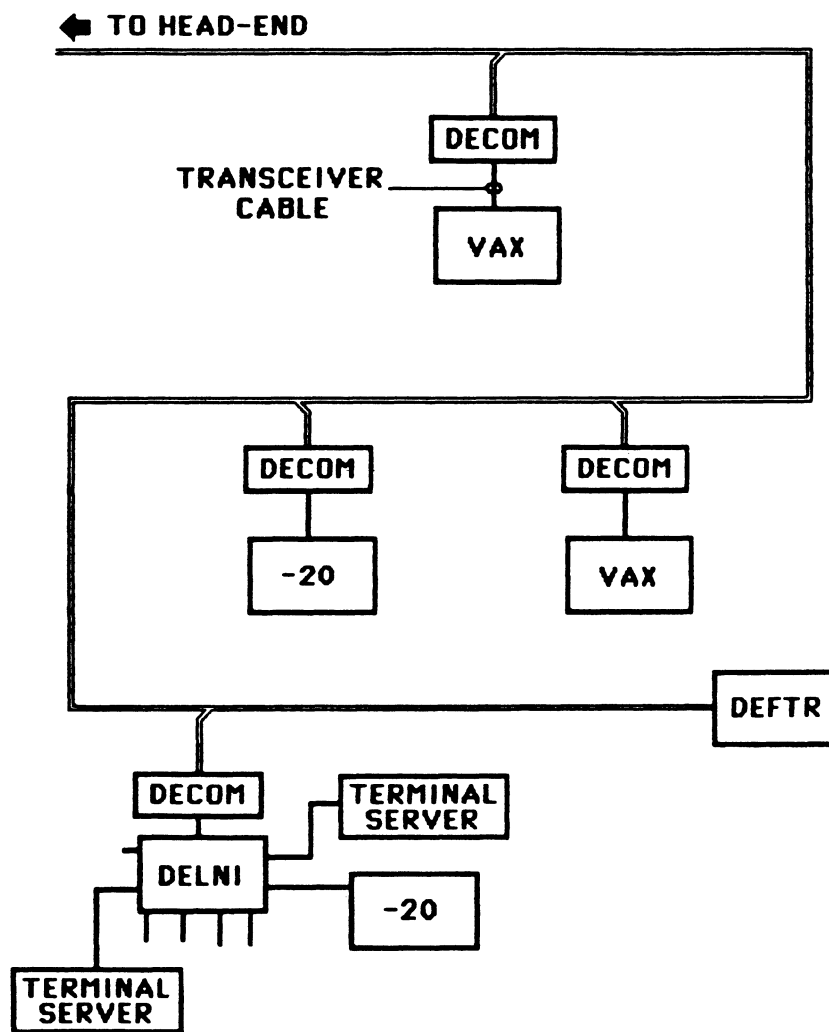
- THE ONLY WAY TO GET FROM A BROADBAND ETHERNET TO A STANDARD OR THINWIRE ETHERNET IS VIA BRIDGES
- NO REPEATERS MAY BE CONNECTED TO THE BROADBAND ETHERNET
- MAXIMUM LENGTH OF A DUAL BROADBAND CABLE IS 15.53 μ SECONDS OR APPROX. 4000 METERS OF TRUNK CABLE BEFORE DROP CABLE AND AMPLIFIER DELAYS ACCOUNTED FOR
- FOR A SINGLE CABLE SYSTEM THIS IS REDUCED TO 15.03 μ SEC. OR 3900 METERS (THIS ACCOUNTS FOR DEFTR DELAYS)

BROADBAND CONFIGURATION

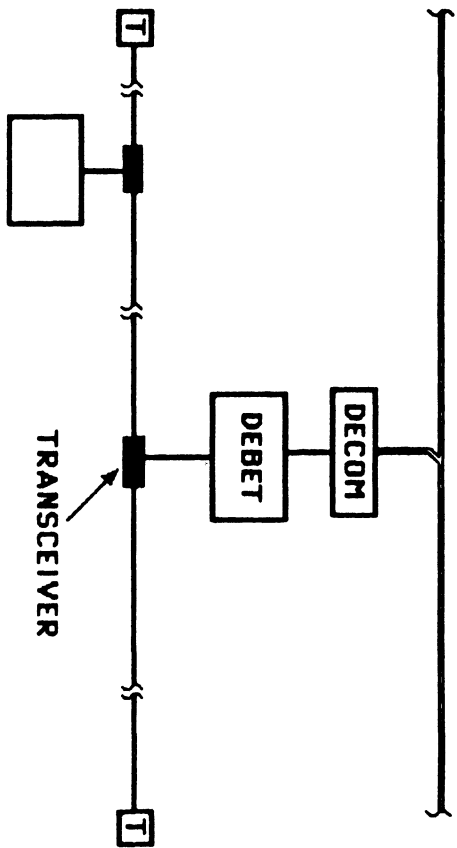
DUAL CABLE SYSTEM



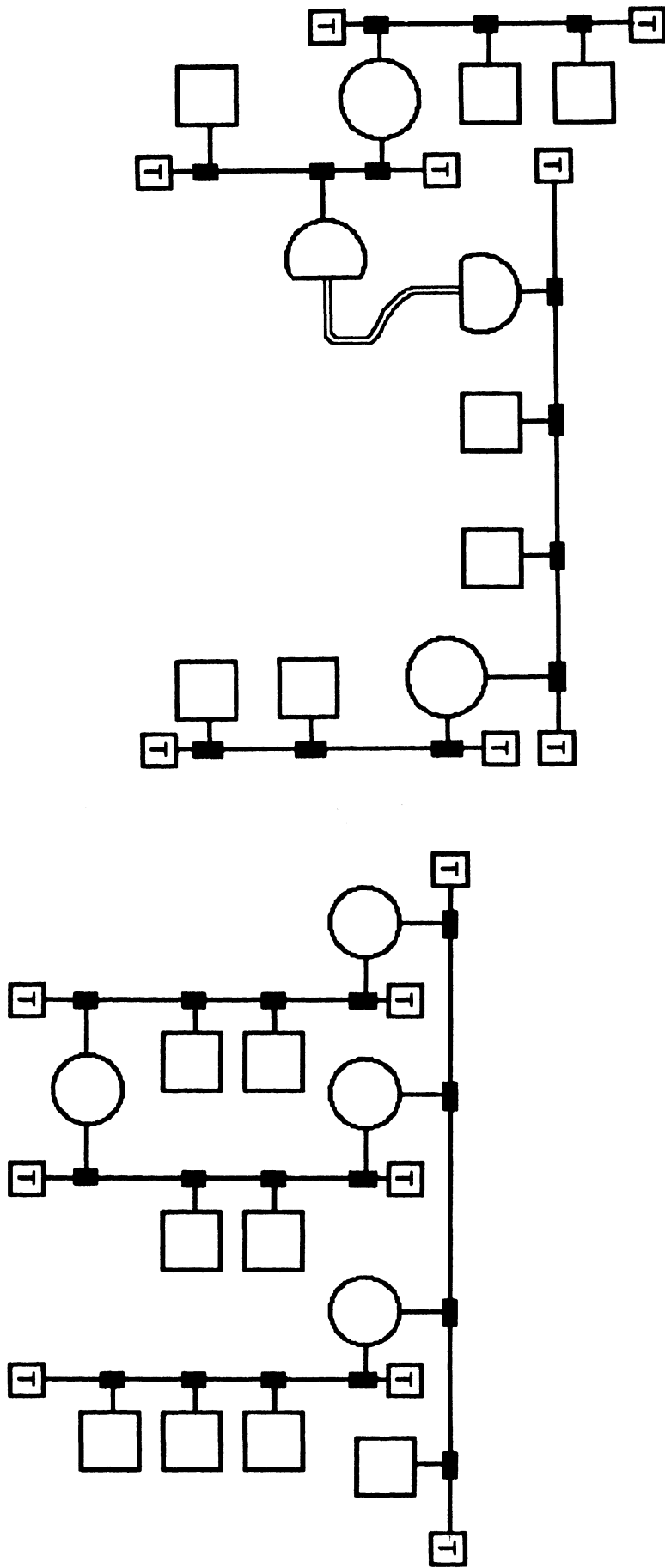
SINGLE CABLE SYSTEM



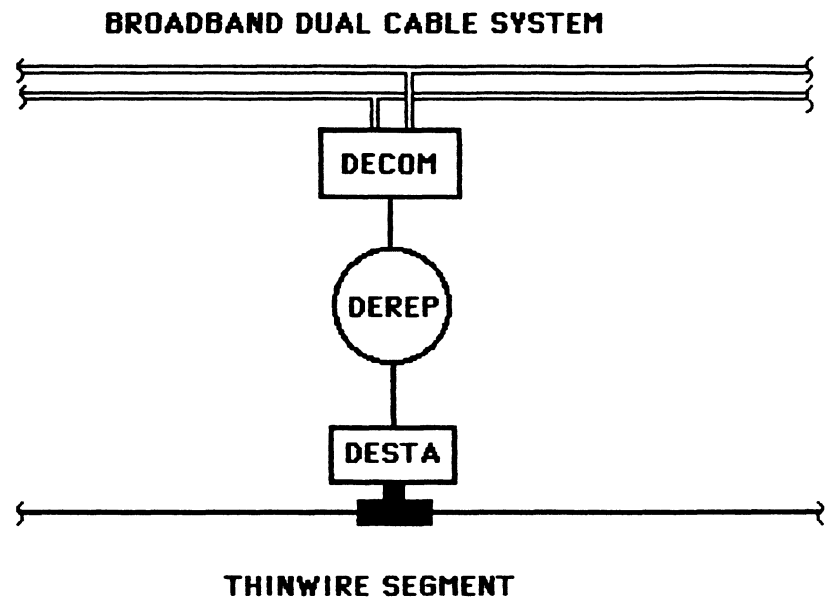
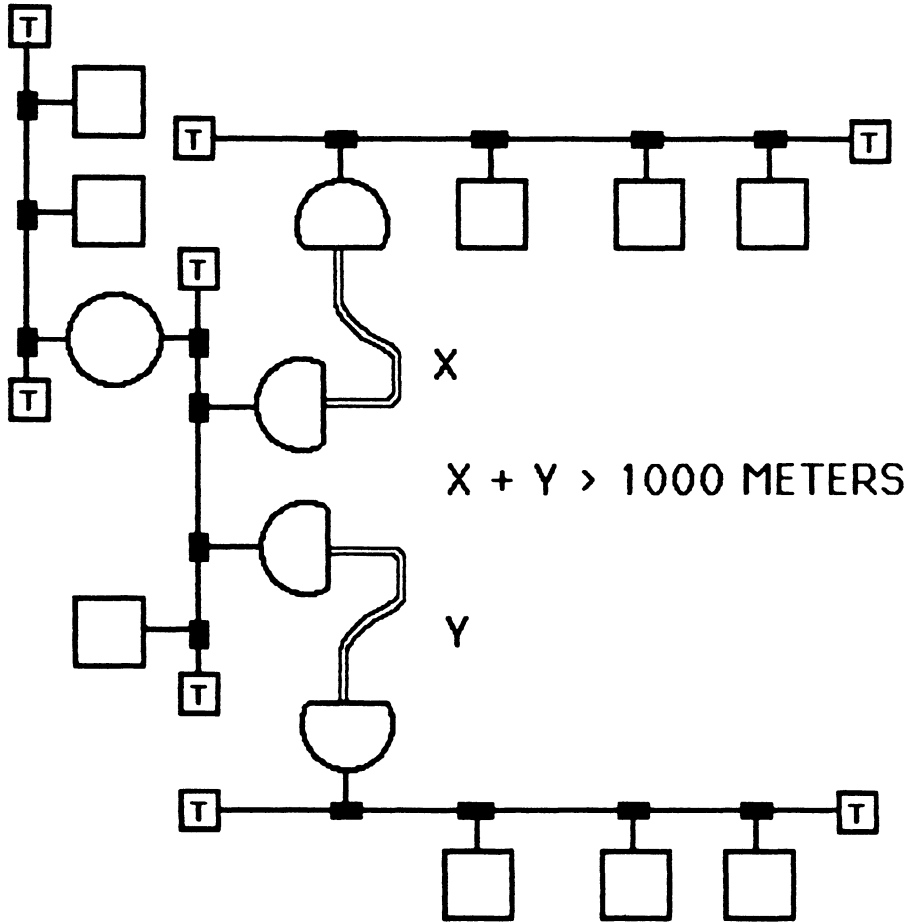
BROADBAND TO BASEBAND CONFIGURATION



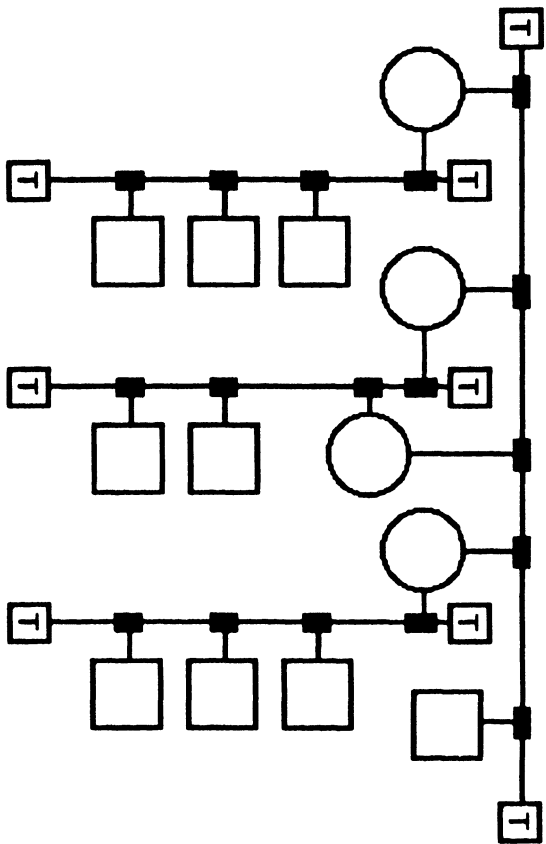
ILLEGAL ETHERNET CONFIGURATION



ILLEGAL ETHERNET CONFIGURATION



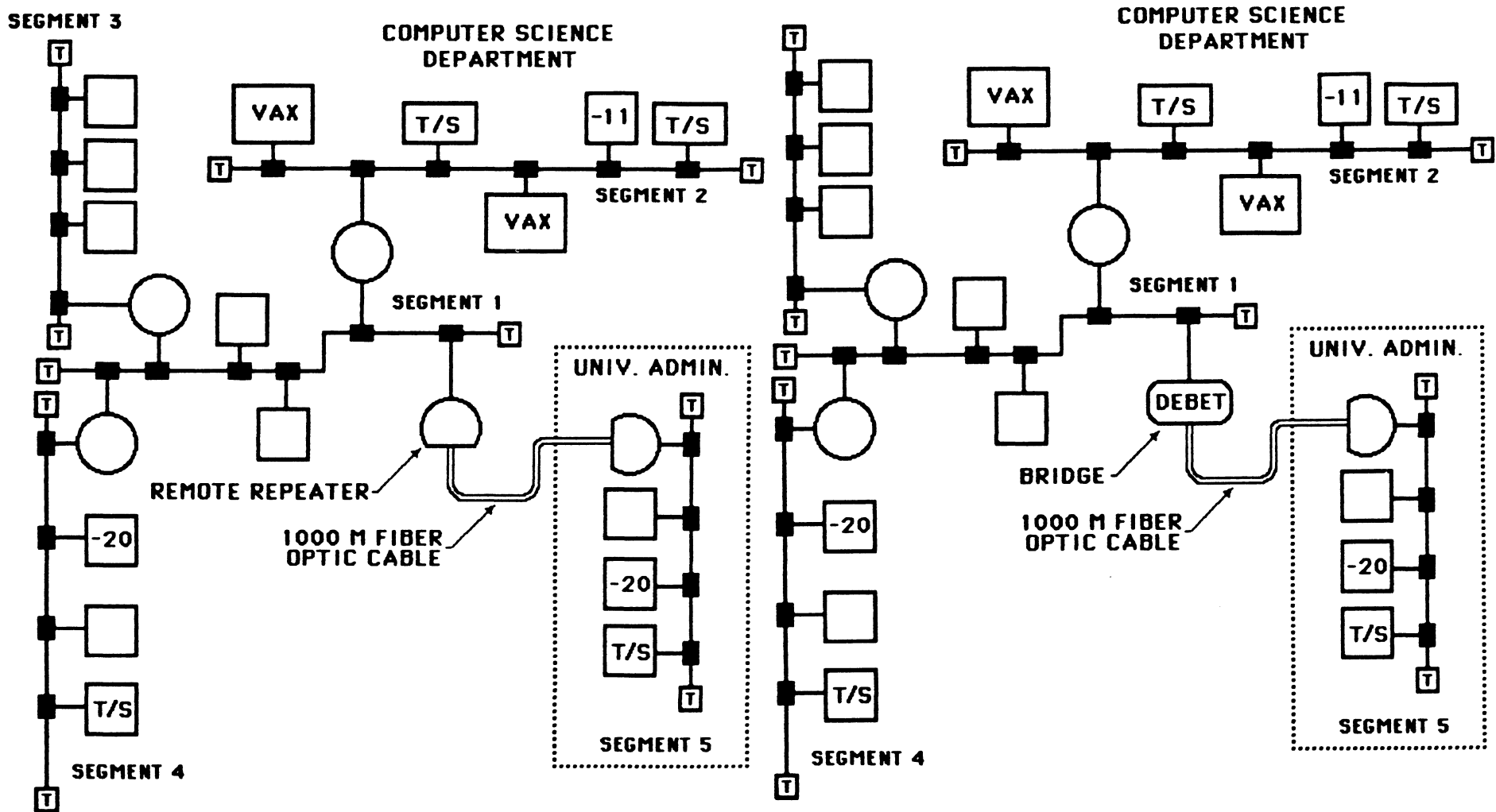
A LEGAL ETHERNET CONFIGURATION



ETHERNET PERFORMANCE CONSIDERATIONS

- KEEP THE NETWORK LENGTH AS SHORT AS POSSIBLE
- MONITOR NETWORK PERIODICALLY.
LOOK FOR SIGNS OF MALFUNCTIONING
HARDWARE.
- CONSIDER USING A BRIDGE OR BRIDGES
TO SEGREGATE USER COMMUNITIES

Harry Mudd University



COMMON PROBLEMS

- **NO NETWORK COMMUNICATIONS**

- **TYPICAL CAUSES**

- ➔ **CABLE BREAKS**

- ➔ **CABLE SHORTS**

- ➔ **JABBER**

- **STATION MALFUNCTION**

- ➔ **CONNECT NEW STATION TO TRANSCEIVER
IF SYMPTOMS PERSIST, REPLACE TRANSCEIVER**

- ➔ **IF NEW STATION WORKS, PROBLEM LIES IN
THE SUSPECT STATION'S TRANSCEIVER CABLE,
NETWORK INTERFACE, THE STATION ITSELF, OR
THE STATION'S SOFTWARE. USE A LOOPBACK
TRANSCEIVER AND DIAGNOSTICS TO ISOLATE
THE PROBLEM.**

COMMON CURES

CURES

- ➔ A TDR WILL AID IN LOCATING CABLE SHORTS AND BREAKS.

- ➔ LACKING A TDR, A VOLT-OHM METER WILL HELP IDENTIFY THE EXISTANCE OF A SHORT OR BREAK

- ➔ ISOLATE THE PROBLEM TO A SEGMENT. DISABLE REPEATERS AND DIVIDE CABLE INTO SMALLER INDIVIDUAL SEGMENTS.

- ➔ CONSULT THE OPERATIONS LOGBOOK.

MAINTENANCE AIDS

- VOLT-OHM METER
- TIME DOMAIN REFLECTOMETER (TDR)
- DOCUMENTATION
- A "CLEAN" INSPECTABLE INSTALLATION
- HOST SYSTEM DIAGNOSTICS
- SPARES (TRANSCEIVERS, CABLES, ETC.)

INSTALLATION TIPS

- **START BY READING ONE OR MORE OF THE BOOKS LISTED IN THE BIBLIOGRAPHY.**
- **PERFORM A SITE SURVEY.**
 - ➔ **IDENTIFY THE SYSTEMS WHICH NEED TO BE CONNECTED INTO THE NETWORK.**
 - ➔ **SPECIFY THE PROPOSED CABLE ROUTE AND LOCATION OF EQUIPMENT ON PAPER**
 - ➔ **VISIBLY INSPECT THE PROPOSED CABLE ROUTE**
- **PLAN AHEAD...BEFORE YOU INSTALL ANY CABLES, THINK ABOUT FUTURE NEEDS AND HOW YOUR PLANNED NETWORK WILL ACCOMODATE THEM OR IS EXTENSIBLE TO SUPPLY THE NEEDED SERVICES.**
- **DOCUMENT - COMPILE AND MAINTAIN A COMPLETE RECORD OF THE INSTALLATION**
- **LEAVE SUFFICIENT SLACK IN THE CABLE TO ALLOW EASY TRANSCEIVER INSTALLATION.**

DOCUMENTATION

- CABLES TYPICALLY HAVE MARKINGS AT 2.5 METER INTERVALS. DURING INSTALLATION NUMBER THE MARKINGS.

- DOCUMENT THOROUGHLY YOUR CABLE INSTALLATION. INCLUDE:

- ➔ CABLE ROUTES
- ➔ CABLE TRAY DISCRIPTION
- ➔ LOCATION OF CABLE MARKS
- ➔ LOCATION OF BARREL CONNECTORS
- ➔ LOCATION OF TRANSCEIVERS

- MAINTAIN AN OPERATIONS LOGBOOK. LOG

ALL

ADDITIONS, DELETIONS, & CHANGES

BIBLIOGRAPHY

- Local Area Networks – John E. McNamara. Bedford, Mass.: Digital Press, 1985
- Technical Aspects of Data Communication, 2nd ed. – John E. McNamara. Bedford, Mass.: Digital Press, 1982.
- Ethernet Installation Guide. Document EK-ETHER-IN. Digital Equipment Corp.
- The Ethernet Source Book – Edited by Robyn E. Shotwell. New York, N.Y.: Elsevier Science Publishing, 1985.

Gary Bremer
Emerson Electric Co.
St. Louis, Mo.

ABSTRACT

This session was presented by Reed Powell, Marketing Consultant with DEC Large Systems Marketing. In this session Reed covered DEC-10/20 peripherals and their possible application on VAX systems.

Information that came out in the presentation that was not on the slides:

There is no RP20 support on VMS. The TM03 controllers on KL's differ slightly from the TM03 controllers on VAX's. TM02 and KL TM03 controllers would need to be replaced in order to use TU45 and TU77 masters. Under VMS, TU70 and TU72 tape drives can be used for everything except booting the system. LP05 and LP07 printers with LP11 controllers, and LP27 printers with the long lines interface can be moved to VAX's. For other LP05, LP07 and LP27 printers, Dataproducts has never produced an upgrade or exchange facility to be able to migrate them to VMS. DN20's can be changed into LAT-11 terminal servers for an Ethernet. DEC is not interested in RP06's and DN20's for trade-ins.

Question: What do we have to do to change 18-bit RA81's which are on an HSC50 to 16-bit RA81's?

Answer: The 18-bit disk drives can be reformatted on-site if the drive is in excellent shape. If not, it will have to be removed and reformatted in Colorado.

Question: What is the recommended buy to put 6250 bpi tape drive capability on my KL that I can transfer to my 8600 in 2 years?

Answer: Buy a TU78 master, and when it is migrated change it into a TA78.

Question: Can I move my LP07 printer with an LP100 controller?

Answer: I don't think so.

RP06, RP07 DISK DRIVES

MASSBUS DISKS

- WILL WORK ON VAXES
- DISKS MUST BE REFORMATTED
- MASSBUS BEING PHASED OUT ON VAX

TU45, TU77 TAPES DRIVES

MASSBUS TAPES

- SLAVES CAN BE USED ON VAX
- NEED NEW TM03 FORMATTER FOR VAX
- MASSBUS BEING PHASED OUT ON VAX

TU78 TAPE DRIVES

MASSBUS

- SLAVES CAN BE USED ON VAX
- FORMATTER NEEDS CHIP CHANGE
- TU78 = TA78 UPGRADE AVAILABLE
- BEST BET: MOVE THEM TO HSC/TA78

TU70, TU72 TAPE DRIVES

MASSBUS & IBM CHANNEL

- DRIVER AVAILABLE FOR VMS
- MUST ALREADY HAVE DRIVES
- A FEW TX02/TX03/TX05S AVAILABLE
- A FEW DX20S ARE AVAILABLE
- MASSBUS NOT STANDARD ON 8600/8650
- BEING DEMOED IN DECUS BOOTH
- BEING DEMOED IN DECUS BOOTH

PRINTERS

MANY, MANY PROBLEMS

- IO BUS PRINTERS ARE PDP-8
- CFE PRINTERS USE LP20
- DN87/DN20 PRINTERS USE LP11
- LP11 CAN MOVE TO VMS

COMMUNICATIONS GEAR

MOST CANNOT MOVE TO VAX

- DC20-X, DN81-X (DH11/DM11) NOT ON VMS
- DZ-11 (AKA DN25-X) SUPPORTED ON VMS
- DMC-11 (AKA DN21) SUPPORTED ON VMS
- DMR-11 SUPPORTED ON VMS

KL10 = VAX-8600/8650 TRADE

CASE BY CASE BASIS

- SOME PERIPHERALS WILL BE TAKEN
- SOME ARE NEEDED FOR SPARES
- DEPENEDS ON THE PERIPHERAL

10/20 HARDWARE MIGRATION

SUMMARY

- MOST PERIPHERALS CANNOT MOVE
- SOME WILL BE TAKEN BACK IN TRADE
- OTHERS MAY BE USED ON PDP11
- OTHERS MAY BE SOLD: OPEN MRKT

Gary Bremer
Emerson Electric Co.
St. Louis, Mo.

ABSTRACT

This session was presented by Thomas Blinn, Technical Consultant with DEC Large Systems Marketing. This session was a repeat of the one given at the last symposium in New Orleans. Using data from various benchmarks that he ran on a 2065 and a 780 along with benchmark data for a 785 and an 8600 obtained from other performance groups within DEC, Tom produced some performance graphs. These graphs were presented along with additional explanation in this session.

Information that came out in the presentation that was not on the slides:

Tom did not have the VAX 8650 performance data with sufficient lead time to incorporate that data into this comparison. He stated that one could presume that for compute intensive benchmarks, which are the majority of these benchmarks, the 8650 will be 40-45% faster than the 8600.

The Whetstone performance comparisons were based on fully optimized computations (optimization showed little or no performance improvement on the 2065).

On single precision computations the 2065 fell in between the 780 and 785, and was about 1/3 the speed of the 8600. With double precision the 2065 came in slower than the 780 and 785, and about 1/4 the rate of the 8600. Though the 2065 performed better than either the 780 or the 785 on G floating, the 8600, which was designed to do G floating well, was 6.5 to 7 times faster than the 2065. With the exception of the 8600 each of the systems did double precision significantly better than G floating. Overall, for these compute intensive Whetstone benchmarks the 8600 came in about 4 times faster than a 780 and 3.5 times faster than a 2065.

A total of 69 different FORTRAN benchmarks were run.

In the integer benchmarks which were used, the 8600 was 2.4 to 3.5 times faster than the 2065.

Most of the benchmarks were single precision benchmarks. With these the 8600 was from 2.3 to almost 13 times faster than the 2065.

The ratio of performance is not a consistent number because the rules the VAX used for compiling FORTRAN programs were not the same as for the KL. The VAX FORTRAN compiler does a lot of optimization that the KL does not do. These benchmarks reflect not only the differences between the operating systems and the hardware, but also the differences in the layered products, such as the FORTRAN compilers. These benchmarks were almost exclusively computation and include very little I/O.

With double precision arithmetic the 8600 ran from 3 to 4 times faster than the KL.

For all the FORTRAN benchmarks the range appears to be somewhere between 2 and 4 times faster than the 2065 for compute performance, with most falling between 3 and 4 times faster.

On the compute intensive single-user COBOL benchmarks the KL does not do as well (50%-100%) as the 780. The COBOL benchmarks on the 8600 range from 3.7 to 6.1 times the performance of those benchmarks on the 2065. COBOL on a KL does a lot better when doing SIXBIT than when it does ASCII. Using SIXBIT the KL is roughly equivalent to the 780, but with ASCII, the 780 was more than 4 times faster.

On the sorting benchmark the 8600 was roughly twice as fast as the 2065.

One benchmark was run that was nominally a multi-user benchmark. It used a line oriented text editor to do some simple editing, then it compiles, links and runs a simple program, and does that repeatedly. For the 8600 the benchmark was running in PDP-11 compatibility mode for the editing tasks. With the 2065 response time rose gradually out to about 85 users at which point it spiked. The 8600 also rose gradually, but did not begin spiking until about 190 users. The KL and the 8600 were roughly comparable out to about 70 users.

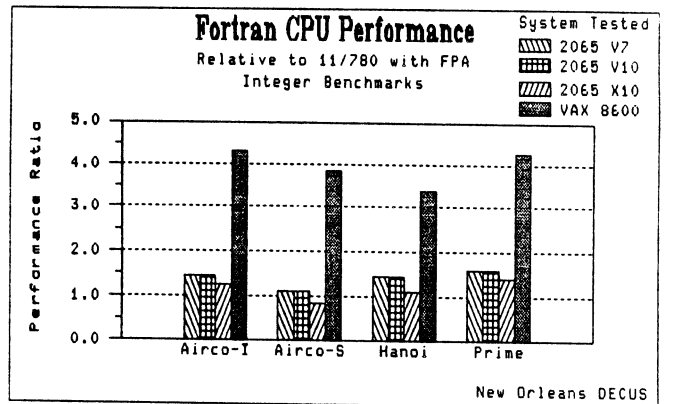
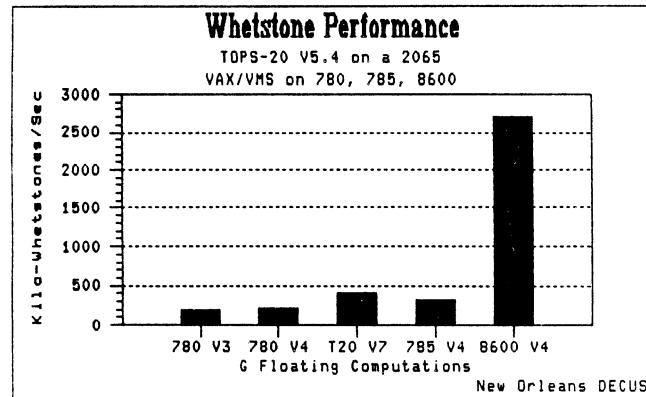
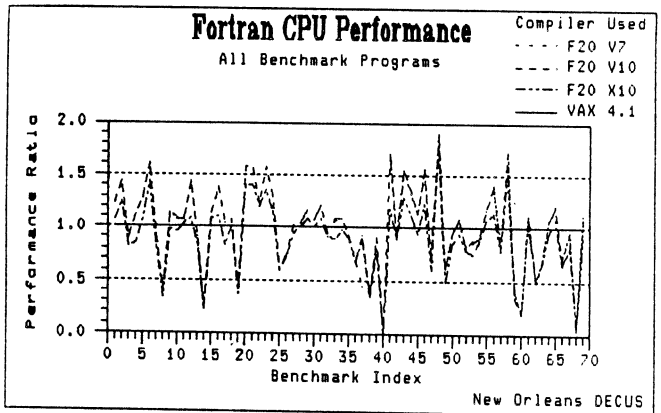
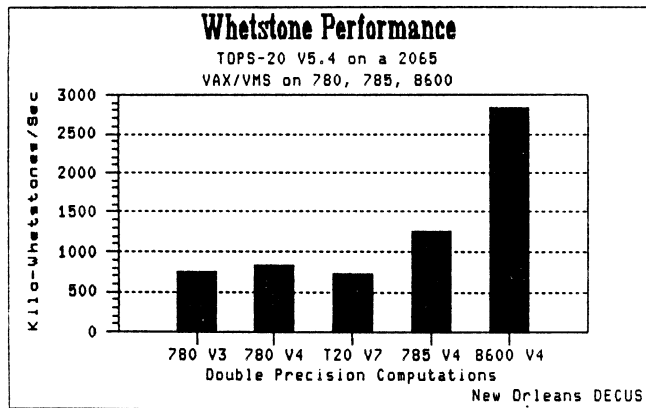
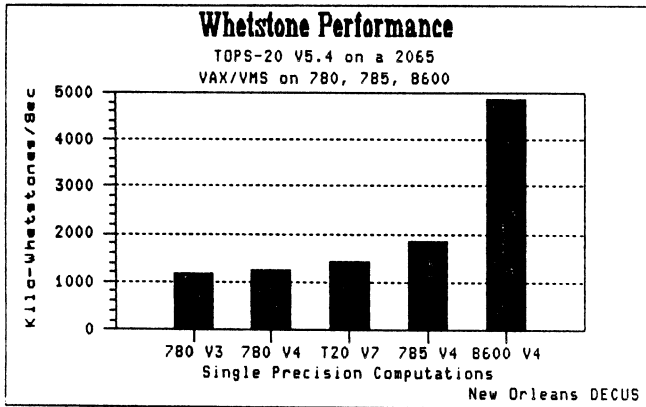
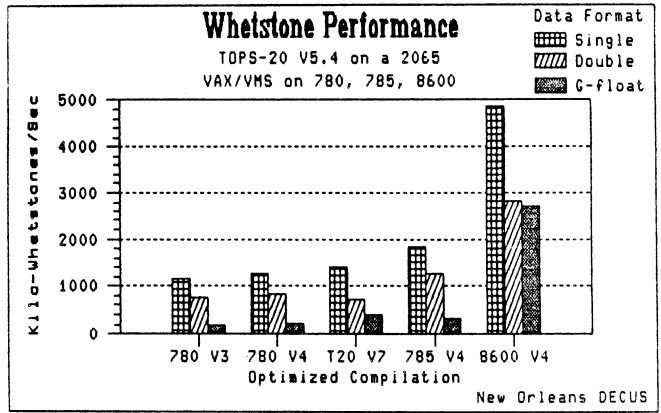
If you really want to know how well an 8600 will handle your applications, you probably want to run some benchmarks using your applications.

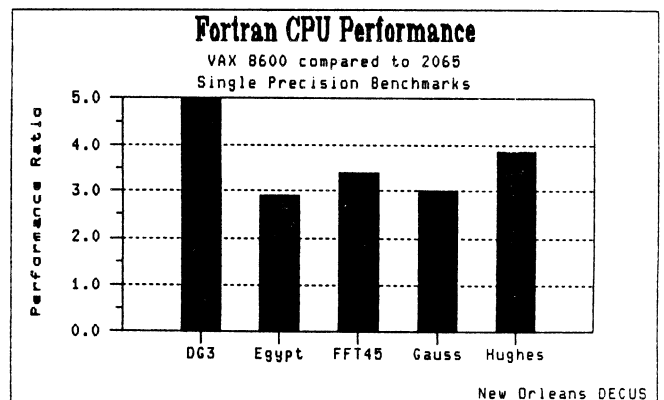
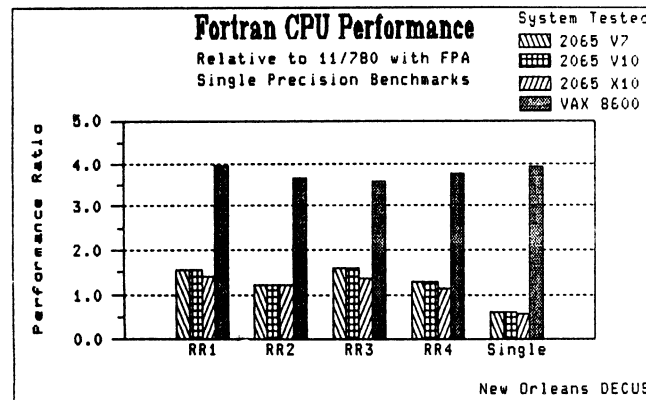
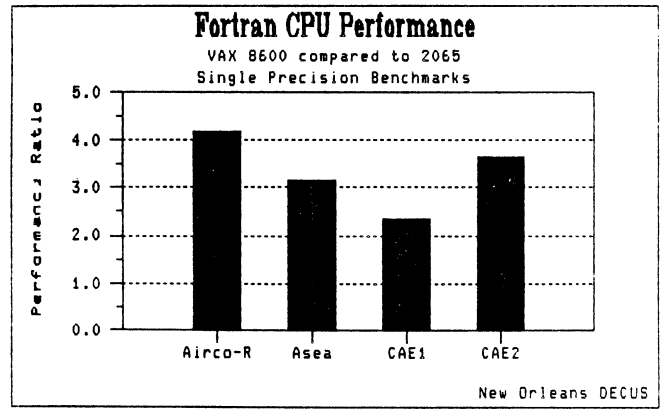
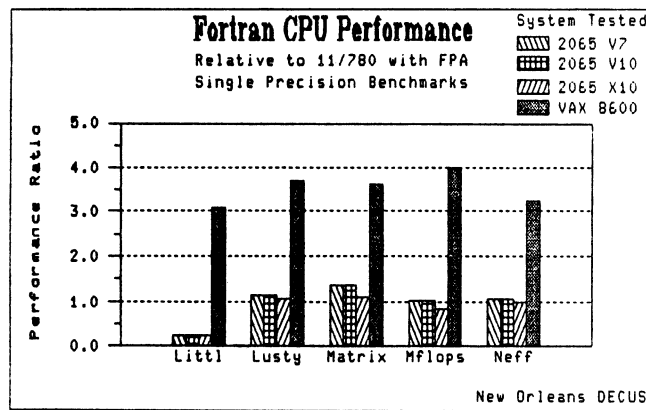
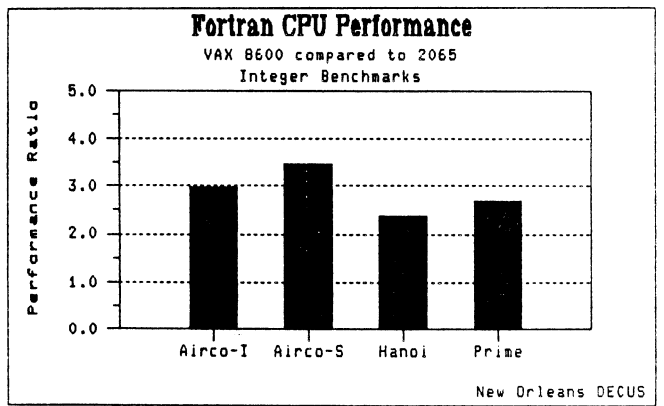
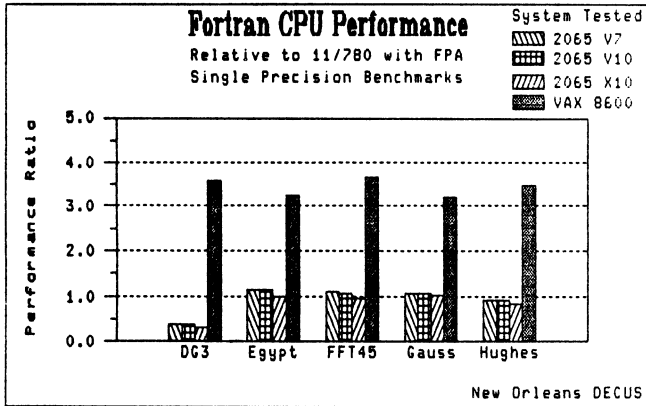
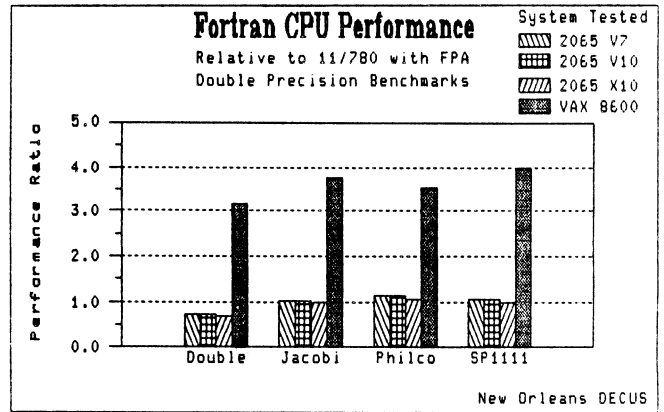
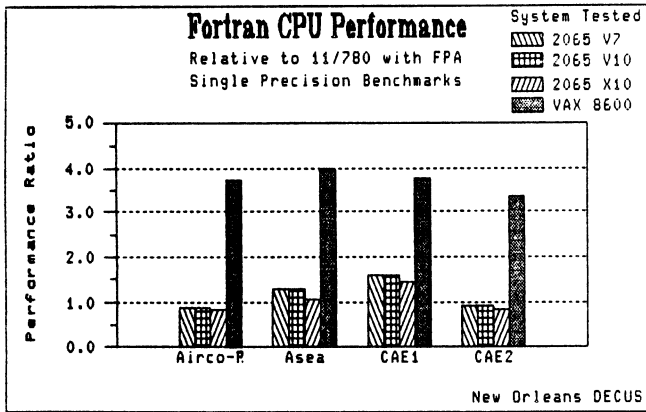
TOPS-20 spends a lot of its time and energy on scheduling the system and trying to keep the system very responsive for interactive users, possibly at the cost of not getting production work done. The VMS layered products do better optimization.

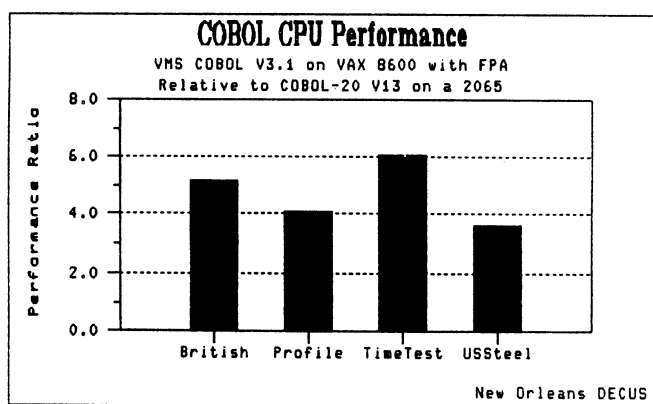
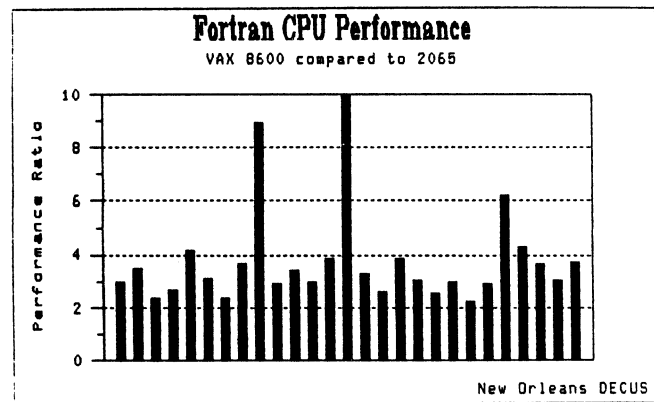
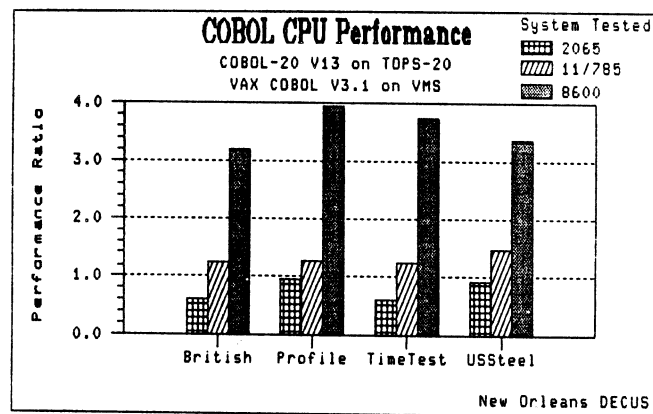
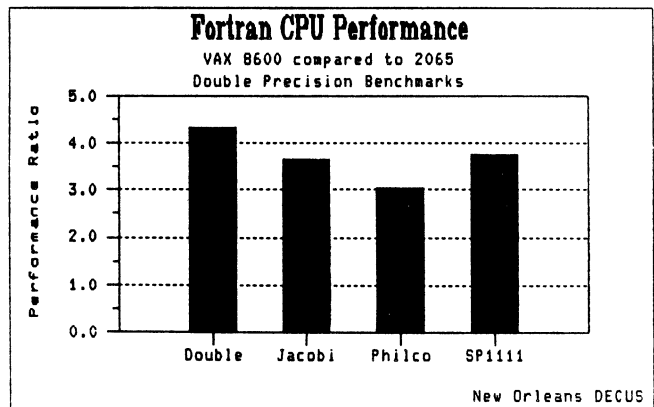
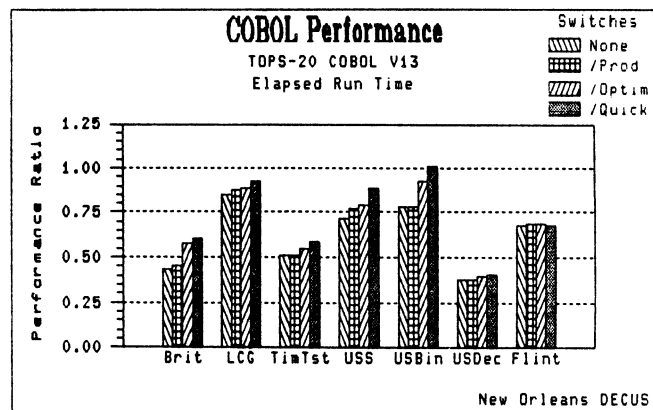
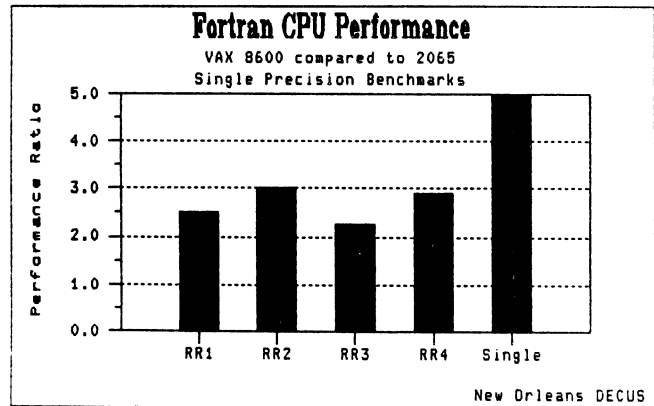
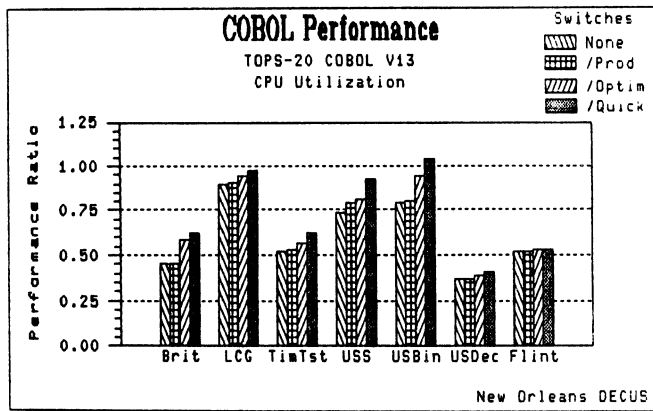
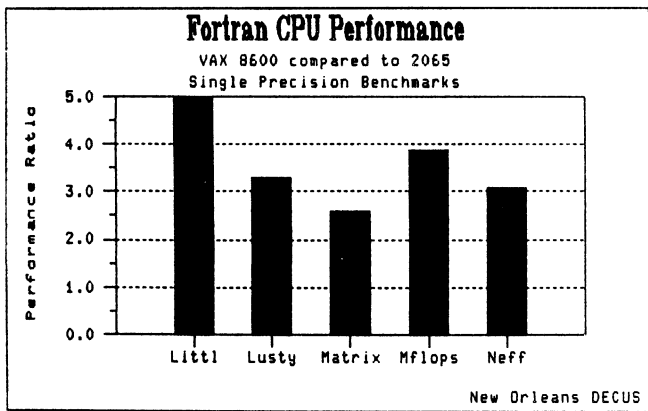
System Configurations

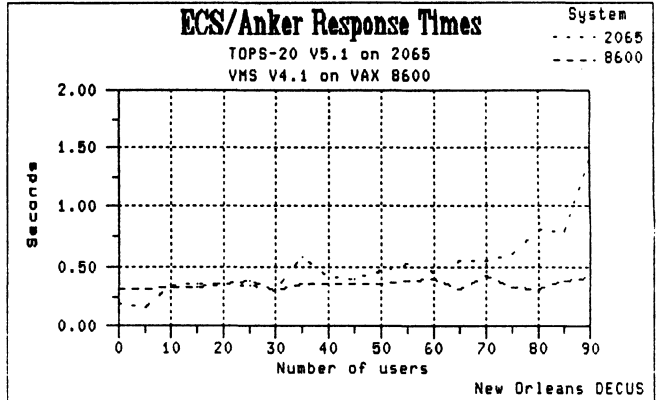
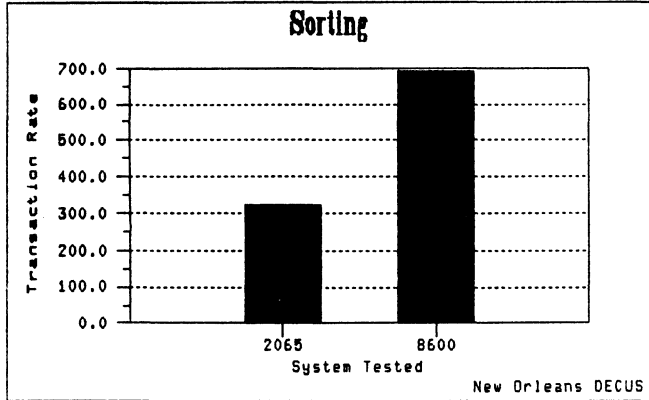
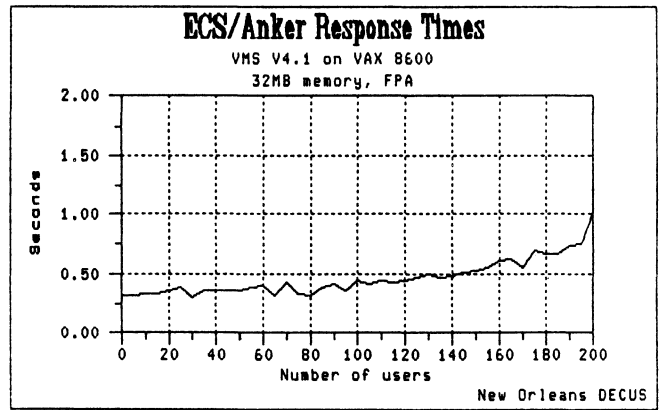
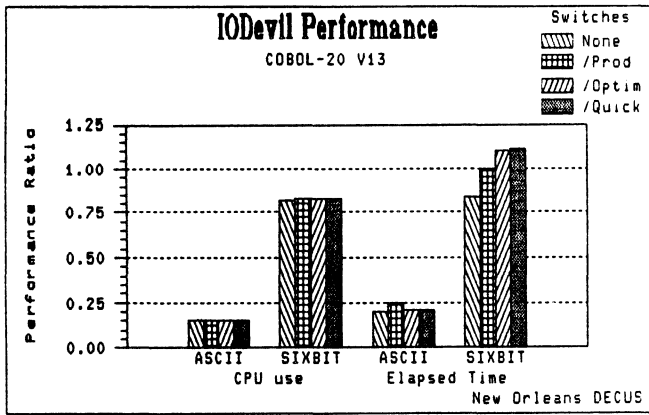
TOPS-20 on a 2MW 2065
 Monitor version 5.4
 Field-image layered software

VAX/VMS on 32MB 11/780, 11/785, 8600
 Monitor version 4.1
 Field-image layered software

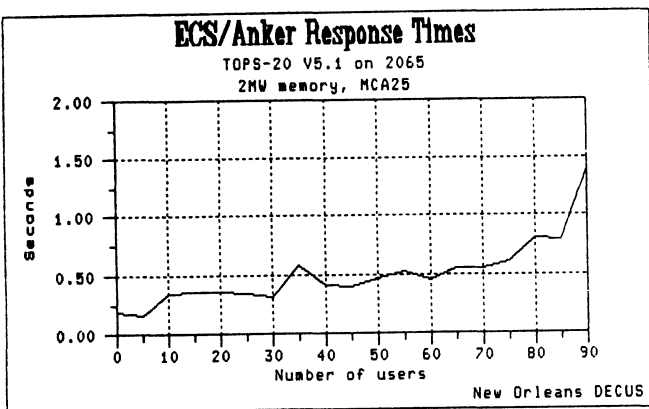








Multi-user
"ECS/Anker"
Benchmark
Results



Gary Bremer
Emerson Electric Co.
St. Louis, Mo.

ABSTRACT

This session was presented by Reed Powell, Marketing Consultant with DEC Large Systems Marketing, and was a review directed to TOPS-10/20 people, of VMS layered products and their significant features.

COBOL

- ANSI 74
- MOST ANSI 85 FEATURES
- SUPPORTS DBMS'S DML
- LSE EDITOR SUPPORT
- REPORT WRITER SUPPORT
- FORMS SUPPORT WITH ACCEPT/DISPL
- IO: RANDOM, INDEXED, SEQUENT

FORTRAN

- ANSI 77
- POSSIBLE TO SELECT ANSI 66
- OPTIMIZED CODE
- GENERATE SHARABLE CODE
- COMMON DATA DICTIONARY SUPPORT

BASIC

- RMS FILE IO SUPPORT
- LSE EDITOR SUPPORT
- CONDITIONAL COMPILATION
- CHARACTER LABELS
- STRUCTURED PROGRAMMING CONSTR

C

- FULL K&R IMPLEMENTATION
- LSE EDITOR SUPPORT
- OPTIMIZED VAX CODE
- USER DEFINED/ENUM SCALERS

APL

- INTERACTIVE INTERPRETER
- TABLES UP TO 64K DIMENSIONS

ADA

- STRUCTURED PROGRAMMING LANG
- DOD SPECIFICATION
- PROGRAMMING ENVIRONMENT
- COMPIL LIBRARY MANAGER
- VMS DEBUGGER
- CAN CALL RUNTIME LIBRARY

PASCAL

- ANSI/ISO COMPATIBLE
- BLOCK STRUCTURED LANGUAGE
- DEBUGGER SUPPORT
- LSE EDITOR SUPPORT

PL/I

- "EXTENDED SUBSET" OF ANSI 76
- USEFUL FOR SYSTEM PROGRAMMING
- GOOD FOR BUSINESS (RMS SUPPORT)
- SUPPORT FOR ALL VAX DATATYPES
- LSE EDITOR SUPPORT
- BLOCK STRUCTURED
- AUTOMATIC INITIALIZATION OF VARS

LSE

LANGUAGE SENSITIVE EDITOR

- "KNOWS" ABOUT LANGUAGES
- EDIT-COMPILE-DEBUG SCENARIO
- INVOKE COMPILERS FROM EDITOR
- INVOKE LANGUAGE CONSTRUCTS
- EDT KEYPAD
- ONLINE HELP
- E (EXPAND) COMMAND
- TAILORABLE TO SPECIFIC ENVIRONMT
- TAILORABLE TO SPECIFIC ENVIRONMT

SCAN

- GENERALIZED TEXT TO TEXT XLATION
- PROCEDURALIZED LANGUAGE
- PROGRAMMING LANGUAGE CONV
- TEXT CONVERSION

Gary Bremer
Emerson Electric Co.
St. Louis, Mo.

ABSTRACT

This session was presented by Kathy Rosenbluh from the DEC Large Systems Marketing Technical Support Group. The main goals of this session were for TOPS people to become familiar with the utilities and decisions necessary for a system manager to set up a VAX system. A large screen projection TV system was set up to display the screen of a terminal connected to one of the VAX's in the exhibit hall.

Using the projection TV system Kathy demonstrated:

1. How to show user parameters in the utility AUTHORIZE and explained what those parameters were.
2. How to add a new user by copying an existing user and changing some of the user parameters.
3. How to create a directory for the default directory.
4. Using the utility DISKQUOTA did a SHOW of directory for the directory being copied from and did an ADD of a directory.
5. Using the utility AUTHORIZE again, did a SHOW of proxy accounts.
6. Displayed what gets accounted for buy using SHOW ACCOUNTING.
7. Demonstrated how to display accounting reports for a particular user including how to tailor them.

Goals

- o Introduce VMS Sys Management Concepts
- o Give feel for similarities/differences
- o Explain choices and options in VMS

Topics

- o Setting up user accounts
- o Accounting
- o System startup, shutdown, and error analysis
- o File system, file structures
- o Security
- o Work load, image considerations, tuning
- o Mountable devices
- o Batch and Print Queues

User Accounts

- o It's all done with Authorize, Directory Creation, and Disk Quotas
- o User == existence of user record in the rights database
- o Authorize == utility to manipulate SYSUAF.DAT
- o Record defines: Allowable types and times of Login

Quotas and privileges

Default directory

Login (and logout) procedures

Default CLI

Username, UIC, password

Determining Username and UIC

- o Personal or functional account?
- o Determine group structures (common files and logical names)
- o GRPPRIV gives some sysmanager control
- o One user can be a member of only one group
- o Some UIC group numbers mean "System Manager"
- o Raw UIC is almost never seen by end user
- o Usernames can't have tree structure

Passwords

- o Three possible password levels:
 - System password
 - Primary user password
 - Secondary user password
 - o Max of 2 can be enabled on any one system
 - o Password can be changed by end user -OR-
 - Password changing can be disabled per user
 - o Account can be set up with no password required
- Can enforce:
- Password length requirements
 - Regular password changing
 - Use of generated passwords
- o Passwords don't echo on login.
 - They CAN echo in network file operations -AND-
 - in batch log files

Quotas

- o 18 quotas. All have defined defaults
- o Systemwide ones are about Max Jobs and Processes
- o Pooled are about open files, enqueues, timer queues, byte i/o, subprocesses
- o Nondeductible (each sub/process gets entire amount): AST, buffered and direct i/o, working set
- o CPUtime is deductible. Usually set to infinite.

Privileges

35 Individual Privileges; 7 Functional Levels

None
Normal
Group
Devour (resources)
System (for operators)
Files
All

Special types of accounts

- o Captive account
- o Proxy account
- o Application-only account
- o Group manager account

Other Account Control Parameters

Specify:

- priorities for user processes
- allowable hours for access
- allowable connection modes (DIALUP, INTERACTIVE, LOCAL, REMOTE)
- o Can audit security-relevant actions
- o Can disable mail receipt, control-Y interrupts, etc.
- o Can disable the account completely
- o Enforce systemwide, groupwide & personal login proc execution
- o Define alternate CLI

Proxy Accounts

- o Represented by records in NETUAF.DAT.
 - Created by Authorize
- o Provides access to FILES from deconnected node
 - Without login or access string
- o Can provide access
 - for all users or for individual from remote system x
 - to all accounts on your system which match remote username
 - or to a specific account on your system
- o Incoming or outgoing proxy access can be dynamically turned on/off

Authorize is not enough...

- o Default login directory is defined in SYSUAF.DAT
- o It must also be created, using
`$CREATE /DIRECTORY = device:[topdir.subdir] /OWNER_UIC = [xxx,yyy]`

- o Permanent and overdraft quotas may be established

PER UIC PER DISK

using the DISKQUOTA utility. This is optional, on a PER DISK basis.

Accounting

- o What gets accounted for
- o Format of accounting file
- o Utility to extract records
- o SET ACCOUNTING command to start shifts

Accounting

SET ACCOUNTING causes records to be sent to ACCOUNTING.DAT for:

System initialization

Process, image termination

Print job completion

Login failure

Optional user messages

Accounting

Accounting utility sorts, selects and reports accounting records

2 important packets: RESOURCE and PRINT

Resource packet: includes info on process performance (page faults, working set size, i/o)

Print packet: includes info on QIO's, symbiont CPU time

System Files

PAGEFILE	Operator Log
SWAPFILE	Error Log
DUMPFIL	Old system dumps
Secondary *FILE	Security audit file

System Generation and Modification

- o Use AUTOGEN to redefine hardware configuration, sizes of system files (PAGEFILE, SWAPFILE, DUMPFIL)
- o AUTOGEN can also figure "best values" of above on its own
- o If you modify some parameters via AUTOGEN, it will adjust other relevant parameters on its own to match
- o (AUTOGEN runs SYSGEN once the parameter files are OK)
- o SYSGEN -- modify dynamic parameters on current system

System Startup Procedure

- o Automatic restart -- no human intervention required
Can be set via SYSGEN
- o Else, in simplest case, at the console
type CONTROL-P to halt system
deposit pc = -1 and psl = 1F0000
and BOOT to reboot
- o BOOT procedure can be nonstop or conversational
 - SYSGEN params can be redefined

Elements in System Startup

DEFBOO.CMD
STARTUP.COM
SYSTARTUP.COM
SYCONFIG.COM
AUTOGEN
SYSGEN

DEFBOO

- o Variety of bootstrap command files provided
- o Choice of nonstop or conversational modes
- o Rename the one you want as default to DEFBOO.CMD
- o Edit file to modify the unit number used, and the directory use
- OR- deposit mod interactively in console mode

SYSTARTUP.COM

STARTUP executes SYCONFIG, runs SYSGEN, and autoconfigures
SYSTARTUP used to define queues, logical names,

install images, set terminal characteristics
manage large system files, make announcements
set # of interactive users, submit batch jobs
mount public disks

System Shutdown

Use SHUTDOWN procedure to gracefully stop queues, logins, etc

-OR- run OPCCRASH

-OR- halt from console by running CRASH
(Gets PC, PSL, and stack pointers first)

Crash Dumps

- o Errors are logged in SYSSERRORLOG:ERROR.SYS
 - o ANALYZE/ERROR LOG to produce report
 - o SDA can be run from SYSTARTUP
 - o SYSDUMP.DMP must exist; size = physical memory + 4 (in ...)
 - o Typical commands: SHOW CRASH, SHOW STACK, SHOW
 - o Other useful files: SYS.STB, SYSDEF.STB, SYS.MAP
- Checking on System Performance

- o Primary tools are MONITOR
- o SHOW commands (PROCESS, SYSTEM, CLUSTOR have dpy m
- o Accounting records
- o Less useful: ANALYZE/SYSTEM

System Tuning – Most Useful Goals

Try for:

- Nearly-zero page faulting
- No job swapping
- No direct or buffered I/O waits
- No outswapped computable
- No DZ's

System Tuning Tools

Tuning is always a trade-off

Primary tools:

- AUTOGEN (SYSGEN)
- Authorize facility (UAF)
- More code sharing

Installing Known and Shared Images

- o Saves physical memory requirements and activation time
- o Uses up some memory for resident headers, etc. even when image is not being used
- o Usually done in SYSTARTUP.COM , can be done anytime

Image Privileges

- o Executable images can be installed with extra privs allowing non-privileged users to run them.
- o Shareable images can be installed with extra privs, allowing non-privileged executable images to run them.

Setting up File Structures

- o Decisions: Public or Private?
 - Quota'd or non-quota'd
 - Multi-pack or single pack
 - Separate User Login disk from System disk?
 - Separate Swapping space from System disk?

Disk Space Management

- o Disk quotas
- o File expiration dates
- o Archiving program from Decus
- o 3rd party Archive program

Optimizing File System Performance

- o Ensure proper RMS_EXTEND_SIZE is being used (SYSGEN)
- o Ancillary Control Processes:
 - MONITOR FCP to see if more are needed
 - Set up separate ACP for slow disks
 - Distribute swapping to 2 disks and add an ACP
 - Extra ACP's cost memory

Mountable devices - Disks

- o At disk initialization time, select values for:
 - o ACCESSED, CLUSTER_SIZE, EXTENSION, MAX_FILES, etc.
 - o Users can request MOUNT on generic device type
 - o (MOUNT utility picks a drive and tells operator)
 - o A single volume disk can be expanded in size with:
 - MOUNT/BIND (unreversible)

Tapes

- o Can use DISMOUNT/NOUNLOAD but No /REWIND
- o No /NOWAIT
- o Users can request a tape to be mounted on a generic device-
- o INIT needed only for writing labels
- o Useful MOUNT switches:
 - /FOREIGN, /OVERRIDE=, /NOLABEL, /RECORD
- o During write operation, continuation tape automatically requested from operator
- o Multi-tape volume sets can be mounted on one or more drives
- o Multiple tape drives can be preallocated and loaded for later use by a batch process

Kinds of Queues

- o Batch, Printer, Terminal, Symbiont
- o Execution ("real")
- o Generic
 - Batch only on clusters
- o Logical ("holding pattern") for printers
 - HOLD switch for batch
- o Server

- o UIC protection scheme applies to queues
- o User can re-start batch jobs
- o INIT/QUEUE doesn't delete jobs
- o Operator can restart a stopped print queue, and set starting position within file
- o Failing jobs can be retained in the queue
- o Operator can re-assign jobs to different queue

Queue Control – Printers

- o Users can force forms changes on programmable printers
- o Default settings for job/file burst flag and trailer pages
 - Users can modify for their jobs
- o Default settings for job separation pages
 - Users can't modify
- o Defaults can be modified

Queue Performance

- o Batch Queue performance manipulation:
 - PRIORITY
 - JOB_LIMIT, CPU_LIMIT, Working set quotas
 - Swap mode
- o Print Queue performance manipulation:
 - by size in blocks expressible as range
 - schedule by [no]size
 - by priority

Queue Management Functions

- o At startup, start the queue manager and open JBCSYSQUE file
- o Select and define spool medium
- o INITIALIZE and START queues

Backup

- o Backup is a command with switches, not a program
- o Backup to disks, to tapes, or to disk and then to tape
- o IMAGE SAVE COPY modes
- o Standalone backup for system disks
- o Effects of /IMAGE, /PHYSICAL, /COPY, /INCREMENTAL

- o OPCOM
 - REQUEST
 - REPLY
- o SYSS\$NOTICE_TEXT
- o SYSS\$ANNOUNCE
- o SYSS\$WELCOME

Security – ACL's

- logging security events
- captive accounts
- elevated privs for images
- password control
- volume erase and highwater
- tape volume protection

INTEGRATION TOOLS FOR TOPS CUSTOMERS

Gary Bremer
Emerson Electric Co.
St. Louis, Mo.

ABSTRACT

This session was presented by two speakers, Thomas Blinn, Technical Consultant with DEC Large Systems Marketing, and Peggy Sullivan, Integration Tools Coordinator in Large Systems Marketing. Joining them for the Questions and Answers was Larry Burke with High Performance Systems in DEC Maintainability Engineering. Tom presented the previously announced tools and how to receive them, and Peggy announced the new tools products.

Information that came out in the presentation that was not on the slides:

Mail gateways which are presently used to get mail between TOPS-20 and VMS will become obsolete when MS is bundled with TOPS-20 in spring 1986. PHONE-20 will allow phone between TOPS-20 and VMS systems. The HOST program will be obseleted by CTERM which is being shipped with TOPS-20 V6.1.

The VAX TU70/72 device driver will be distributed to TOPS-10 and TOPS-20 customers and is for sharing these drives between TOPS systems and VMS systems. MVUSRS moves user accounts with their passwords and priviledges from TOPS systems to VMS.

Question: How often will tools tapes be coming out?

Answer: We are hoping for quarterly. You need to make sure you are on the mailing list.

Question: Will the tape utility support reading of BACKUP tapes in non-INTERCHANGE format?

Answer: I believe that is the goal.

Question: Are the clearinghouse tools available on-line?

Answer: Some of them are, but I suggest using magtape instead of KERMIT. Also, I will try to provide sources for most of the VAX tools which have sources on MARKET.

Question: Currently we are on TOPS-10 V7.01A. Are we going to have to go to V7.02 in order to share TU72's?

Answer: We won't look.

Question: Are you looking for VAX 8600 test sites for the TU70/72 device driver?

Answer: Yes. See me, Peggy Sullivan.

TOOLS CLEARINGHOUSE

- o Similar to DECUS Library
- o Tools Received from Customers or from within Digital
- o No Support or Warranty
- o Distributed "AS-IS"
- o Software and Documentation

SOFTWARE TOOLS

- o Utilities
- o User Commands and Interface
- o Program and Data Conversion
- o Text Editors

UTILITIES

- o Tape Utilities
- o Mail Gateways
- o PHONE-20
- o New HOST program
- o DECnet-10 Utilities
- o REV for VMS
- o Others

TAPE PROCESSING UTILITIES

- TENVAX - TOPS-10 Program to READ/WRITE VMS ANSI Tapes
- 10BACKUP - VMS Program to READ TOPS-10 BACKUP Tapes
- CONVRT - VMS Program to READ TOPS-10 BACKUP Tapes
- DUMPERC - VMS Program to READ TOPS-20 DUMPER Tapes

MAIL GATEWAY UTILITIES

- o VMM - "MM" for VMS V3.X
- o VMAIL/VMAILR - Exchange Mail Between VAX-MAIL and TOPS-20 DECMail-MS

NEW HOST PROGRAM

- o For TOPS-20
- o Establishes Virtual Connections to VMS via DECnet
- o Passes Escape Sequences Used by Video Terminals

DECNET-10 UTILITIES

- o FAL-10 Patches for VMS Connections
- o "TELL" Program for Remote Execution of User Commands

REV FOR VMS

- o Used to Aid Users in Disk Space Management
- o Interactive Deletion, Viewing & Organizing of Disk Files

USER COMMANDS AND INTERFACE

- o Inline Help Facility
- o TOPS-20 Commands on VMS: Command File Library
- o TOPS-10 Commands on VMS: Command File Library
- o COMPILE Class Commands

TEACH-VMS

- o Runs on VMS
- o Interprets TOPS-20 Commands
- o Displays VMS Equivalents
- o Provides Command Completion and Incremental Help

TOPS-10 COMMAND FILE LIBRARY

- o Library of Command Files
- o LOCATE Command Simulation

"COMPILE" AND "COMPIL2"

- o Support for TOPS-Style Commands: COMPILE, EXECUTE, LOAD, DEBUG,...
- o Data-checking on .OBJ Files

INLINE HELP FACILITY - "RECOG"

- o Runs on VMS
- o Provides TOPS-20 Style Help
- o Command and Filename Completion
- o Incremental Help - "?"

PROGRAM AND DATA CONVERSION

- o APLSF Workspace Converter
- o COBOL Conversion Aids
- o BATCH and MIC Command Conversion
- o SPSS Conversion Aids
- o 1022 to 1032 Data Conversion

SFTOVX-APLSF WORKSPACE CONVERTER

- o Use with APLSF-10/20
- o Converts Workspace to VAX-11 APL
- o Written in APLSF

COBOL CONVERSION AIDS

- o ISACON - ISAM File Unload/Reload Program Generator
- o Reads TOPS COBOL Program
- o Writes TOPS Program to UNLOAD ISAM File
- o Writes VMS COBOL Program to Load RMS/ISAM File
- o Writes COBOL Record Definitions for Use with the VAX Common Data Dictionary

MIC & BATCH FILE CONVERSION

- o CONBAT Converts Commands
- o Generates Comments Which Show Incompatible Commands
- o Allows for User Enhancement of Tools

SPSS CONVERSION AIDS

- o SPSS Routines
- o Aids in Converting SPSS Files to VMS

1022 TO 1032 DATA CONVERSION

- o "DSCONV1"
- o Written by Software House, Inc.
- o Performs Dataset Conversion from 1022 to 1032

TEXT EDITORS

- o TV for VMS
- o SED for VMS
- o EDT-10

DOCUMENTATION

- o Comparisons
- o Case Studies
- o Third-Party Software

COMPARISONS

- o Command Language
- o Editors
 - TECO
- o Languages
 - BASIC
 - COBOL
 - FORTRAN

CASE STUDIES

- o COBOL-10/20, DBMS-20 & TRAFFIC-20 to VMS COBOL, VAX-DBMS, & VAX-TDMS
- o DEC M.I.S. Department
- o COBOL-10 & DBMS to VAX COBOL & VAX-DBMS
- o FORTRAN to VAX-FORTRAN

THIRD-PARTY SOFTWARE

- o Clearinghouse Contains Documentation and Related Information
- o Software is Believed to be of General Interest
- o Documentation Includes:
 - Resource Accounting Package
 - EMACS Editor

- o On MARKET, LSM's 2065 Timesharing System
- o Dialup: (617)467-7437
- o ARPAnet: DEC-MARLBORO
- o Interactive Access - at the "@"
LOGIN LCG.CUSTOMER CUSTOMER

GETTING INFORMATION

- o Use "HELP ?" to Get a List of Topics You Can Select
- o Use "HELP Topic" to Get Information on a Specific Topic, e.g., "HELP REQUEST"
- o Use "NEWS" to Read the Integration Bulletin Board
- o Use "TOOLS" to Read the Tools Bulletin Board

SHARING INFORMATION

- o Issue the Command "MAIL"
- o Send to: TOOLS to Place a Message on the TOOLS Bulletin Board
- o Send to: NEWS to Place a Message on the NEWS Bulletin Board
- o Answer the Other Prompts with the Subject of the Message, Your Name, and so Forth

REQUESTING TOOLS

- o Issue the Command "MAIL"
- o Answer "To:" with "REQUEST"
- o Answer "Subject:" with "REQUEST FOR TOOLS"
- o Answer the Other Prompts with Your Name, Phone, etc.
- o We Will Mail the Appropriate Tape(s) to You at no Charge

OBTAINING TOOLS BY MAIL

- o US Mail, Send Request to:
Technical Support - Tools Clearinghouse
Digital Equipment Corporation
MR02-2/8D2
1 Iron Way
Marlboro, MA 01752
- o Specify Whether You Have a DEC-10 or a DEC-20

TOOLS SUBMISSION

- o Please Share Tools!
- o Use the LCG.CUSTOMER Account to Send Mail to SUBMISSION
- o Use Postal Mail to:
Tools Submission
Large Systems Marketing
Digital Equipment Corporation
One Iron Way MR02-2/C2
Marlboro, MA 01752
U.S.A.

- o LSM's Newspaper
- o Free to Interested Parties
- o Covers Integration, Tools, and Other Topics
- o Send Subscription Request to:
P&CS Mailing List Maintenance
Digital Equipment Corporation
10 Forbes Road NR03-1/M1
Northboro, MA 01532-2597
- o Include Your Name, Company, Job Title, Complete Mailing Address, and Telephone Number

NEW INTEGRATION PRODUCT ANNOUNCEMENT

VAX TU70/72 DEVICE DRIVER

- o Allows TU70/TU72 Tape Subsystems to be Shared with VAX-11/780, VAX-11/782, or VAX-11/785 Systems Running VMS
- o Protects Hardware Investment
- o Provides High Duty Cycle Tape Drives
- o Future Support for VAX 8600

VAX/VMS LAYERED PRODUCT

- o No Special DCL Interface Required
- o Full Support from Digital

PREREQUISITES:

- o VMS V4.3 or Later
- o TOPS-10 V7.02 or Later
or
TOPS-20 V6.0 or Later

SOFTWARE PACKAGES INCLUDE:

- o QE187 Device Driver
- o Diagnostics
- o Documentation

HARDWARE OPTIONS

- o TX03A-AA/AB = DX20 + RH780 + TX03
- o Cables
- o TX03A-BA/BB = RH780 + Cables

THIRD-PARTY INTEGRATION TOOLS

CLONE

- o Convert COBOL-10/20 to VAX-11 COBOL
- o Menu Driven
- o User Definable
- o Online or Batch Mode

RAF - REMOTE ACCESS FACILITY

- o Access Remote System 1022, System 1032, Databases
- o Request Remote Processors to:
- Execute Procedures
- Perform Predefined Remote

FUTURE INIEGRATION TOOLS

FORTRAN TRANSLATOR

- o Translates FORTRAN-10/20 to VAX-11 FORTRAN
- o Rule-based System

GENERALIZED TAPE UTILITY

- o Read BACKUP and DUMPER Tapes Under VMS
- o Read Foreign Tapes Under VMS

Converting FORTRAN Programs from TOPS-10/20 VAX/VMS

William G. Gerken
Personal Products Company
Milltown, N.J. 08850

Abstract

This session involved a discussion on TRANSFORT, the automatic language translator which converts DECsystem-10 and DECSYSTEM-20 FORTRAN programs to VAX/VMS FORTRAN.

Peggy Sullivan, DEC's Integration Tools Coordinator, described TRANSFORT as a FORTRAN translator that will help to convert FORTRAN programs from a DECsystem-10 or DECSYSTEM-20 to VAX FORTRAN. TRANSFORT was developed by the Lexeme Corporation in Pittsburgh to run on the VAX and works on all versions of FORTRAN programs created on FORTRAN V5 thru FORTRAN V10 on the 10's and 20's.

Since FORTRAN 10/20 has many extensions and VAX FORTRAN follows much closer to the ANSI standard, it makes translation very difficult with an automatic translator. TRANSFORT will do some, but not all translations. Based on previous testing TRANSFORT will be able to convert and eliminate approximately 50% of the errors that would occur during a first pass thru the VAX FORTRAN compiler. TRANSFORT will comment on all code it has converted and will give warnings on what differences it cannot convert. The number of differences that cannot be converted will depend on how closely your code follows the ANSI standard. In regard to A5 and A10 formats, TRANSFORT will only give warnings.

TRANSFORT has gone through an acceptance testing at Digital and was sent back to Lexeme for modifications. The new version was received by Digital and sent to 3 or 4 "field test" customers at the end of November. Due to the short period of time between distribution and this symposium, the feedback has been minimal. Peggy Sullivan reported that John Collinger of the University of Pittsburgh had run a "fairly large" FORTRAN program thru TRANSFORT and reduced 90 some errors thru the first pass of VAX FORTRAN to about 42 errors with no adjustments (as described by the warnings). Another copy was sent DEC's Colorado Springs facility where George Clinard reported that he originally had problems with running TRANSFORT until he set the page fault limit on the VAX to 30000 and then it ran fine. He also experienced that approximately 50% of the errors found in his programs were corrected.

Peggy reported that TRANSFORT should be available thru the Integrations Tools Clearinghouse possibly by the end of February. Although it does not handle all of the errors found in the code, TRANSFORT will help in the "grunt" work of converting your FORTRAN programs to the VAX.



BENCHMARKING THE 8600: A DEC-20 USER MAKES THE MOVE TO VAXLAND

by

Samuel B. Whidden

Director, Computer Services, American Mathematical Society¹

DECUS — Anaheim, CA, December, 1985

Abstract—The American Mathematical Society has been a satisfied user of DECSYSTEM-20s since 1978 and at present has two 2060s. DEC's 1983 Jupiter decision forced the AMS to select a new computer system to succeed its 20s. Believing that VAXes and VMS were as foreign to its experience as any other system, in 1984 the AMS issued a Request for Proposal to several computer vendors, describing in detail required hardware and software capabilities. The RFP covered requirements for performance, networking and communications, office automation, and applications development in light of the AMS' 5-year plan for acquisition of new computing resources.

This article will examine the AMS' comparison of Digital's VAX 8600, Data General's MV/10000, and IBM's 4381 as solutions to its computing requirements. The systems were compared on the basis of cost-performance indexes, communications facilities, and potential software functionality. The performance of each machine was benchmarked against the 2060s, using performance tests devised from in-house production applications. From the point of view of AMS requirements, the 8600 out-performed its competition in nearly all respects.

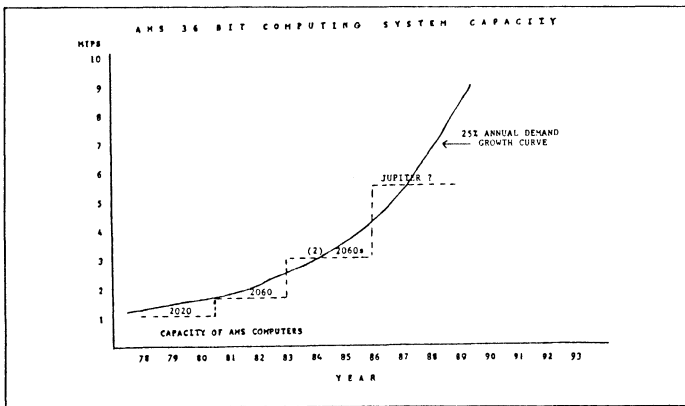


Figure 1: AMS 36-bit System Capacity

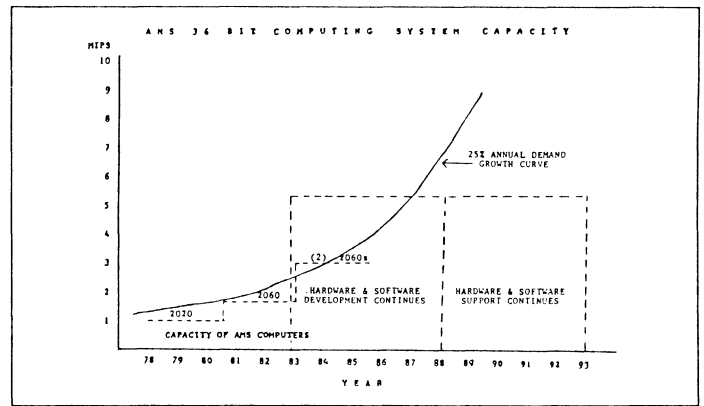


Figure 2: DEC's 36-bit phase-out schedule

The American Mathematical Society, a 36-bit user. The AMS bought its first DEC-20, a 2020, in 1978. The power (approximately 0.7 MIPS) of our original 2020 was quickly exhausted. The 2020 was replaced by a 2060 (1.3 MIPS) in 1980. A second 2060 was added in 1983, giving us a total of 2.6 MIPS. Figure 1 shows our consumption of MIPS over the years. A growth curve of 25% per year, relatively common in the industry, fits these points reasonably well. Projecting that curve out a few years showed a potential need for 8 MIPS by 1989. We couldn't be sure that growth rate would persist, but we had to plan as though it would.

Is there a Jupiter in your future? It seemed clear that we would eventually be a candidate for the Jupiter. Jupiter was rumored to be a processor four times as powerful as the 2060's KL-10. We attended a non-disclosure meeting about Jupiter at Digital and began to plan for the time when we would make this major upgrade. It looked as though our Jupiter ought to arrive in 1986.

But at the Spring DECUS of 1983 in St. Louis, DEC an-

nounced Jupiter's cancellation. With Jupiter went the end of DEC's 36-bit product line. We were faced not only with no path for expansion, but the computer systems in which we had invested years of development and use would have to be replaced.

DEC's timetable for 36-bit euthanasia. (Figure 2.) DEC promised that DECSYSTEM-20 hardware and software development would continue for five years, through 1988. They promised that hardware and software support would be available for at least five years beyond that, through 1993. In 1984, the year after the Jupiter decision, the AMS had to decide what its new strategy would be. Since our DEC-20s were facing extinction, we knew we had to abandon them. One question was, how quickly? We decided that, with DEC promising support through 1993, we would be wise to be independent of the 20s by 1990. 1990 was far enough away to give us plenty of time to make the shift, but was early enough to give a comfortable margin of safety in case either our schedules slipped during the interval, or the level of DEC support dropped faster than expected. So we developed a five-year plan covering 1985 through 1989, during which we would select, install, and migrate our extensive applications systems to

¹©American Mathematical Society, 1985.

THE REQUEST FOR PROPOSAL

Table of Contents:

- The American Mathematical Society
- The Existing Computing Environment
- The Proposed Computing Environment
- The Five-Year Hardware/Software Plan
- Vendors Proposal: Instructions and Requirements
- General System Requirements
- Technical System Information

Figure 3: RFP, Table of Contents

some new computing system.

But which computing system? Should it be a VAX? To us as DEC 36-bit users, VAX was as foreign as any system from any other manufacturer. Worse, VAX was a product of Digital Equipment Corporation, who had just badly let us down over Jupiter. We weren't about to jump blindly onto the VAXwagon. We decided to prepare a formal RFP for a new system, describing all the characteristics of hardware and software we wanted, and submit it to vendors who might have the systems we needed.

The RFP. (Figure 3.) Our RFP was very detailed. We wanted to cover all the bases, to set forth all the ideas we had accumulated about the ideal (for us) system. These included hardware and operating system capacity and performance, availability of vendor and third party software utilities and tools, and the future outlook.

We described our existing environment in detail, and the new environment we wanted to construct. We outlined our five-year development plan.

We told the vendors how we wanted them to propose and what we wanted them to propose, both generally and in technical detail. In appendixes, we provided them with all the characteristics and usage statistics we could for our existing systems. Altogether, we provided some 70 pages of specifications.

We discussed our RFP with more than a dozen vendors and actually submitted it to eight who seemed to qualify by virtue of their size and product line and their apparent stability in the large computer market. We received responses from six vendors and eventually narrowed the field to three: IBM, Data General, and DEC.

We went over each of the three proposals with the vendor in detail. Each vendor refined his proposal more than once, making it more responsive and more competitive. We invited three different groups of independent consultants to examine our options, including the hardware, software, and communications aspects of each of the proposals. It was apparent that any of the three systems was capable of meeting our requirements. Our task became that of selecting the one that would do so best.

In the end, we and our consultants chose the 8600. What made a good choice for the American Mathematical Society won't necessarily be the right one for someone else. But there are common concerns in almost any computer selection and I think our experience is worth sharing.

SOME CONCERNS:

- A Distributed Or A Centralized System?
- We Had Little Need To Convert Existing Applications
- But timing was important
- "We're Interested In Your Future, But You Can Only Sell Us What You Can Deliver"

Figure 4: Some Criteria For Computer System Selection

A distributed vs. a centralized system. (Figure 4.) At first we thought that our existing centralized system should become more distributed. We thought that for some, if not many, purposes, individual users should have personal computers; perhaps some of the computing power in our present central computers should be distributed to users. Our users are largely office workers, not engineers with CAD/CAM requirements, but industry literature seemed to be telling us that users wanted desk-top computing and independence from central domination. Perhaps when the user recognized that the impossibly heavy load on his computer was due to nobody but himself, he might ease up a little in his criticism of the computing center.

But none of the vendors with whom we talked offered a convincing distributed system. Our office is heavily computerized, with the number of terminals roughly equalling the number of employees (about 200). Although there was no broadly integrated package of office automation tools available for TOPS-20, our unsophisticated users had available to them many powerful and reasonably user-friendly tools, in addition to the TOPS-20 operating system itself, through the mainframes. These were such tools as EMACS, the highly functional MM mail system, a spelling checker, a spreadsheet, SCRIBE, and good laser hard-copy output. At the time of our investigation, micro technology had not reached a point where that kind of power could be made available to that many users at a price much less than about twice the cost of similarly equipping them through the central computer. Nor were we much encouraged by the degree of user friendliness we found in micro systems, or by the horror stories we heard from users in firms where micros grew like mushrooms in every dark corner. Nor did it seem that good distributed access to central data bases was yet available. So early on, we moved away from the idea of distributing our computing power, and concentrated on cost-effective shared systems.

We didn't need to convert existing applications. At the time of the Jupiter announcement, the AMS was a year into a major software redevelopment project. We had earlier decided to recreate our business data processing applications, pretty much from scratch. Most of these applications were publishing and financial systems written in-house in the early 1970's. They had become too worn and patched to be serviceable. We had decided it was time to redevelop them using up-to-date specifications. When we undertook this redevelopment project, it was with our 2060s, or their 36-bit successor, in mind as the operational computer. The Jupiter decision changed our expectations, but we did have the advantage of not having the large-scale conversion of existing applications as one of the major criteria for selection of a replacement. We didn't necessarily have to have a new sys-

SOFTWARE REQUIREMENTS:

- A powerful, user-friendly operating system
- Good application development tools
- Integrated Office Automation tools
- Broad availability of third-party software

Figure 5: Software Requirements

tem that would support TOPS-20. We could select a system on which to develop brand new versions of our applications.

But timing was important. Our board of trustees had approved an applications development schedule that called for the start of program development by January 1, 1986. That meant that benchmarking, selection, and testing of new development tools had to start by the preceding September 1. The new computer, whatever it was, had to be delivered and up and running by then.

“We’re very interested in the future, but you can only sell us what you can deliver”. We started our investigation in September, 1984. Because of our development schedule, we needed delivery of the first new system by September, 1985. We were very interested, of course, in vendors’ future plans, and we listened to them eagerly, if a bit skeptically, at each non-disclosure session. But we couldn’t buy it if the vendor couldn’t deliver it to meet our schedule. So vendors were forced to restrain some of their enthusiasm for future products and concentrate on showing us how their present equipment beat the competition.

A powerful, user-friendly operating system. (Figure 5.) We had nearly 200 interactive users who had become accustomed to the very user-friendly TOPS-20 operating system. We couldn’t afford to dump them onto some user-insensitive, unforgiving operating system with cryptic error messages and un-mnemonic commands. Nor could we afford to sacrifice operating system power for ease of use. In addition, the operating system had to be one which supported the T_EX typesetting system, which the AMS uses for publishing much of its mathematical literature.

Good application development tools. We had been charged with a massive applications redevelopment program. In addition to standard compilers and utilities, our programmer/analysts needed the best fourth-generation applications development tools we could afford. Either our new computer vendor had to supply them, or they had to be available from third parties for the new equipment.

Integrated Office Automation tools. We wanted to offer those 200 interactive, managerial, administrative, and clerical users a broadly competent, thoroughly integrated office automation system—one which combined word processing, editing, electronic mail, spreadsheet, business graphics, calculator, calendar, scheduler, and whatever else was available, into an easy-to-use, efficient office tool.

Broad availability of third-party software. One of the weaknesses of TOPS-20 has been the relative scarcity of good third-party software. We wanted our new system to improve on that.

A Local Area Network... (Figure 6.) Inevitably, our system would grow beyond a single machine. Our two DEC-20s were

COMMUNICATIONS REQUIREMENTS:

- A Local Area Network...
- ...that would couple the new system efficiently to the DEC-20s

Figure 6: Communications

HARDWARE REQUIREMENTS:

- Computing Power
- Cost/Performance
- State Of The Art

Figure 7: Hardware Requirements

connected only by a version of FTP and a shared disk drive. We wanted virtual terminal and terminal server capabilities. We wanted to be able to add specialized work stations to the system when we needed to.

...that would couple the new system efficiently to the DEC-20’s. The 20s would be around for five years or more. We needed easy access to them for existing data and programs as we rebuilt our applications on the new system. Whatever the new system was, there had to be a mechanism to connect it smoothly to the 20s.

Computing power. (Figure 7). The new system had to assure us enough computing power to meet our projected need for 8 MIPS, or the equivalent, by 1989.

Cost-Performance. The AMS is neither large nor rich, and we needed the lowest ratio of cost to performance we could find.

State of the art. We wanted our new system to be of recent-enough architecture that we wouldn’t spend a lot of time in the next year or two wishing we had waited for something better.

What the Vendors Offered. The vendors responded to our RFP as shown in Figure 8. IBM suggested we start with a 4381 Model Group 1, to be replaced by a Model Group 2 machine in 1987, to be replaced in turn by a 4381 Model 3, the largest in this series, in the last year of our 5-year plan. We calculated total 5-year costs for this option to be about \$2.6 million, including all the costs of the system: vendor and third-party hardware and software.

DEC offered twin 8600s, one in 1986 and another the following year. These two machines together would provide for our tentative requirement for 8 MIPS by the end of the planning period. Since at the time of the proposal, DEC could not promise delivery of the first 8600 in 1985, when we needed it, they offered to install, temporarily, an 11/785 at no extra cost. Overall costs were projected at just under \$2 million.

Data General proposed three MV10000s over the planning period to bring us close to the 8 MIPS level, at a 5-year overall cost of about \$1.9 million.

With any of these proposals, careful monitoring of machine usage would be the real determinant of timing for installation of machines after the first.

IBM’s proposal left us substantially below that 8 MIPS figure, but both we and IBM believed that the mainframe architecture

The Vendors' Proposals:						
	1985	1986	1987	1988	1989	Total ²
IBM (VM/CMS)						
Syst:	4381-1		4381-2		4381-3	\$2,607K
MIPS:	2.1		2.7		4.6	4.6
DEC (VMS)						
Syst:	11/785	8600	8600			\$1,974K
MIPS:	1.7	4.46	4.46			8.9
Data General (AOS/VS)						
Syst:	MV10000		MV10000		MV10000	\$1,906K
MIPS:	2.5		2.5		2.5	7.5

Figure 8: The Choices

of the 4381 would allow that machine to perform as well as, or better than, the DEC or Data General superminis despite the difference in MIPS. IBM's representatives explained to us that mainframes like the 4381 utilize relatively powerful and intelligent peripheral controllers to relieve the CPU of the overhead and I/O that burden minicomputer processors. In the end, though, our findings didn't support this contention.

A Benchmark for AMS. Our systems programming group, led in this task by Betsy Ramsey and Barbara Beeton, devised a series of benchmarks by means of which we could compare these machines.

Computing at the AMS mainly comprises four categories: business data processing (primarily batch production in the areas of membership records, subscription and order fulfillment, and financial record keeping); typesetting; office automation; and application development. We based our estimates of machine efficiency on the first two areas: on performance measurements in application processing and in production typesetting. For fourth-generation development tools and for office productivity software, we planned to examine vendor and third-party software offerings.

The COBOL Benchmark. We expect to move eventually to fourth generation development tools, but we've been a COBOL shop for years so an important part of our benchmark was based on COBOL. A multi-program segment of one of the Society's business application systems was selected as the production benchmark. Successful compilation and linking of the several programs and modules (one of which was a Fortran module), as well as the actual running of the application, was required as part of the benchmark of each vendor's system. The COBOL portion of the benchmark represented an I/O-intensive application.

The T_EX Benchmark. The AMS publishes several journals and book series of mathematical research. Such material is

²Costs are for complete system, including both vendor and third-party hardware and software, not for computers alone.

Benchmark Seconds:				
	DEC	VAX	MV	IBM
	2060	8600	10000	4381-2
Memory:	1 MW	12 Mb	6 Mb	16 Mb
CPU Time, Standalone:				
TeX run	248.9	119.9	339.6	151.0
Cobol compile	6.1	3.8	12.7	1.7
Fortran compile	1.3	.7	1.4	.3
Application run	73.3	9.6	59.4	7.7
CPU Time, With Load:				
TeX run	283.5	122.2	N/A	160.8
Cobol compile	6.1	3.8	12.8	2.3
Fortran compile	1.3	.7	1.4	.3
Application run	78.0	9.7	61.4	8.1
Elapsed Time, Standalone:				
TeX run	285.0	128.0	373.0	170.0
Cobol compile	13.3	4.9	27.0	6.0
Fortran compile	4.4	2.3	8.0	2.0
Application run	99.0	37.0	310.0	35.0
Elapsed Time, With Load :				
TeX run	2210.0	209.0	N/A	329.0
Cobol compile	37.4	10.6	37.0	15.0
Fortran compile	11.2	7.9	18.0	2.0
Application run	519.3	78.0	1209.0	52.0

Figure 9: Raw Benchmark Data

difficult and costly to typeset by conventional methods, and we have turned in recent years to in-house, computerized composition. As a result, the AMS has become a leading supporter of the T_EX typesetting language, a system devised by Donald Knuth of Stanford for the typesetting of scientific and mathematical literature. Versions of T_EX were already available for each of the target machines, and part of our benchmark consisted of a production run of this typesetting language. The T_EX typesetting portion of the benchmark represented a relatively compute-intensive application.

Each vendor was given a tape containing our sources and input files. Each was asked to do the necessary installations, compiles and links, to run the benchmarks, and to report the results. Although we were present to observe the benchmark runs in most cases, we were not there for all. But vendors were told that, were we to buy their machine, our acceptance test would consist of duplicating the reported benchmark results.

For comparison, the full benchmark was run on our DEC 2060, as well.

None of the three vendors had any difficulty installing the T_EX language on their systems and processing our test input file. None of the three, however, was able to port our COBOL application completely successfully. They all did finally run the application successfully enough to get what we believe were reliable timings, but fully correct final output was never obtained by any of the

Standalone Power Of Machines, Relative To 2060:				
	2060	8600	MV10000	4381-2
Tex Run:				
CPU	1.00	2.08	.73	1.65
Wall	1.00	2.23	.76	1.68
Application Run:				
CPU	1.00	7.64	1.23	9.52
Wall	1.00	2.68	.32	2.83

Figure 10: Speed relative to the 2060

Standalone Power Of Proposals, Relative To 2060:			
	8600	MV10000	4381-3
TeX Run:			
CPU	4.16	2.19	2.81
Wall	4.46	2.28	2.86
Application Run:			
CPU	15.28	3.69	16.18
Wall	5.36	.96	4.81

Figure 11: Relative Processing Power Of Each Proposal

three vendors.

Benchmark Performances. Figure 9 presents the raw benchmark data. Each machine was configured with the amount of memory we would buy initially.

There were four parts to the benchmark: the T_EX production run, a series of COBOL compiles, a Fortran compile, and a production run of the resulting application system. CPU time and elapsed time were measured for each part under both standalone and loaded conditions. For tests under load, a moderate standard load was applied, consisting of four continuously self-resubmitting batch streams containing the four parts of the benchmark. Timings were then taken on each part of the benchmark, run separately. Each measurement was taken a number of times.

An error occurred during the timing of the typesetting production run on the Data General machine under loaded conditions which invalidated the results.

IBM ran the benchmark on the 4381 Model Group 2 but would not run it on the Model Group 3 machine. Instead, they suggested we estimate the speed of the 4381-3 as 1.7 times that of the Model Group 2, which we did.

Computing Speed Relative to the 2060. Because our experience with our data processing workload is based entirely on DECSYSTEM-2060s, we wanted to understand these proposals in terms of the 2060. Figure 10 compares the speed of each machine to that of the 2060 on the two production runs in standalone mode. The numbers represent the time taken by the 2060 divided by the time taken by each of the other machines; the higher the number, the faster the machine (the greater the machine's power, by this measure). The 8600 showed about twice the power of the 2060 on the compute-intensive T_EX benchmark. The 4381-2 was a little slower than the 8600 on this test.

The 8600 performed more than seven times faster than the 2060 on the I/O intensive COBOL application run when CPU time was measured, but fell to little more than twice the speed of the 2060 in wallclock terms. We took this as a reflection of relatively high waiting time for I/O service in the 8600, under the particular circumstances of this COBOL application. The 4381-2 was superior to the 8600 on this test, giving a performance equal to more than nine 2060s in CPU terms. But to our surprise, the

4381 showed an even greater vulnerability to I/O than the 8600, using nearly the elapsed time required by the 8600.

The generally better CPU performance shown in the application run, relative to the T_EX run, is due to the fact that the computing requirements imposed by the COBOL application were lighter in both quality and quantity than those of the T_EX run. Not only does T_EX do its own memory management, but, because of its stringent requirement for portability, it uses only integer arithmetic—its results on one machine must be precisely the same as those on any other. The Fortran module imbedded in the COBOL application, on the other hand, does all its calculations in floating point, at which both the 8600 and the 4381, equipped with floating point accelerators, excel.

The numbers in Figure 10 can be read as the number of 2060s to which each of these machines would be equivalent, under the special circumstances of the AMS production benchmarks. The 8600 gave a performance equal to between two and seven and a half 2060s. The performance of the 4381-2 equalled about one and a half to nine and a half 2060s. Since AMS data processing contains roughly 12 times as much COBOL application processing as T_EX processing, the extra strength of these machines in this area was important.

Our performance tests did not yield quantitative comparisons of these machines in interactive mode, although we did assess qualitatively the strengths of each vendor's office automation system. We were also unable to measure machine performance under overloaded conditions. At the time of our investigation (early spring of 1985), no user whom we contacted had had an 8600 long enough to saturate it, in terms of either batch production or number of interactive users. Subsequently, there was this tantalizing quote from the introduction to Tom Blinn's session at the New Orleans Symposium, comparing 2065 vs. 8600 benchmark results:

ECS Anker was the basis for the multi user benchmark. This benchmark on the KL produced acceptable response times which ranged from 1/4 second to 1 1/2 seconds at 90 users. The 8600's response curve was flat out to 90 users; the maximum response time was 1 second and occurred at 200 users... There is still a need for better multi-user benchmarks...⁸

Throughput Comparisons:				
Throughput = CPU Time/wall	DEC 2060	VAX 8600	MV 10000	IBM 4381-2
Standalone:				
TeX Run:	.87	.94	.91	.89
Applic. Run:	.74	.26	.19	.22
With Load:				
TeX Run:	.13	.58		.49
Applic. Run:	.15	.12	.05	.16

Figure 12: Relative Throughput

Comparing Proposals. (Figure 11.) To assess the power of the full set of machines in each proposal, we simply multiplied the relative results obtained for each machine alone by the number of machines proposed by each vendor: two for DEC, three for Data General, and 1.7 to relate the IBM 4381 Model Group 3 to the 4381-2. The numbers in this figure are again relative to the performance of a single DECSYSTEM-2060. They can be interpreted as the "2060-equivalence" of each of the three proposals, again under these limited circumstances.

Throughput Comparisons. The ratio of CPU time to elapsed time can be viewed as a measure of "throughput efficiency". That is, in the standalone case, it should reveal the percentage of available CPU cycles devoted to the problem task. The remaining CPU cycles can be assumed to be devoted to operating system overhead and input/output waiting time. In the case of the TeX benchmark, I/O wait time is small, so the throughput ratio can be viewed as a rough measure of operating system efficiency. The standalone numbers in Figure 12 show the 8600, with a ratio of 94%, to be the most efficient in this respect. The MV/10000 yielded 91%. The 2060 CPU devoted only 87% of available cycles to the task, and the IBM 4381-2 89%, possibly as a result of these systems' relatively high level of attention to time-sharing overhead.

The COBOL application run involves greater I/O activity than the TeX run. Comparing the throughput ratios of the various machines under this circumstance gives a rough estimate of their sensitivity to I/O demands. The 2060, with its ratio of 74%, appeared to be relatively "I/O-resistant". Both the 8600, its throughput ratio dropping from 94% to 26%, and the MV/10000, dropping from 91% to 19%, were, at least in this test, much more affected by I/O demands. The low resistance to I/O of the 4381, as measured here, surprised us. As I noted earlier, it was our understanding that, in the supermini computers—the 8600 and the MV/10000—I/O is handled largely by the CPU itself, whereas in mainframes, such as the 2060 and the IBM machines,

Standalone Weighted Production Units:				
	2060	8600	MV10000	4381-2
Weighted Units:	1128.5	235.1	1052.4	243.4
Rel. to 2060	1.00	4.80	1.07	4.64

Figure 13: Weighted Benchmark Seconds

separate intelligent I/O controllers are supposed to absorb much of that load.

When a load is added, the CPU must deal with the increased overhead requirements of scheduling, page swapping, additional memory management, and background tasks, as well as with the computing requirements of competing tasks and their I/O requirements. It can be seen by comparing the throughput ratios for the TeX benchmark under conditions of load and no load, that the 2060 is more sensitive to load than the 8600 when I/O demands are small. Adding an I/O demand puts a strain on both machines, with the 2060 perhaps a slightly better performer under the limited conditions of this test. Complete data are not available for the MV/10000, but it appears sensitive to I/O, with or without load. The 4381's performance shows a greater resistance to workload than to I/O, very much like the 8600. We would have expected the reverse to be true, and the 4381 to resemble more the 2060 than the VAX.

Summarizing The Benchmark Results. To take account of our mix of production requirements, we derived a weighted benchmark unit for each machine, which was simply twelve times the number of CPU seconds required for the COBOL application run, plus the number of seconds used for the typesetting run. Figure 13 relates these weighted units to the performance of the 2060. We used these weighted units to summarize our findings.

Figure 14 summarizes our benchmark results. We compared the three proposals quantitatively in several ways. We had three basic pieces of information to work with: the total cost of each proposal, the total number of MIPS each proposal would provide us, and the benchmarked performance of each machine.

Once again, the **total cost** figures given in Figure 14 represent all projected costs, over the five-year planning period, for vendor and third-party hardware and software, not just the cost of the proposed machines. The figure for **total MIPS** is the number of MIPS for each machine times the number of machines in the proposal. **Weighted benchmark units** are those derived in Figure 13.

One way to view the proposals was to rank them in terms of a known quantity. "2060-equivalents" is the number of 2060s to which each machine was equivalent, determined in Figure 13, multiplied by the number of machines in the proposal. Liberally interpreted, this is the number of 2060s we would have to buy to accomplish the same amount of production (in the mix represented by the benchmark) of which these proposals are capable.

Cost per MIPS and **cost per 2060-equivalent** are simply the total proposal costs divided by total MIPS or by 2060-equivalents. These ratios help put the total costs in perspective.

³Blinn, Thomas P. and Kazzaz, Dan, "TOPS/VMS Performance Comparisons", *Proceedings of the Digital Equipment Users Society*, Spring, 1985

VENDOR	No. of MACHINES	TOTAL COST \$000'S	TOTAL MIPS	COST/ MIPS \$000'S	WT'D BENCH-MARK UNITS	COST-PERFORM-ANCE INDEX	2060 EQUIVA-LENTS	COST/ 2060-EQUIV \$000'S
DEC	2	1,974	8.9	222	235.1	23.2	9.6	206
D.G.	3	1,906	7.5	254	1052.4	66.9	3.2	592
IBM	1.7	2,607	4.6	567	243.4	37.3	7.9	331

Figure 14: Comparing The Options

Finally, The **cost-performance index** is the proposal's total cost per machine (its total cost divided by the proposed number of machines) multiplied by the weighted benchmark units for that machine (divided by 1000). Both cost and benchmark units are better the smaller they are, and so is the index. The product of the cost and the measured speed of each machine constitutes its cost-performance index.

The Conclusion. Admittedly, our benchmarks were a small sample of the kinds of work that might be done on these machines. Nor could we control all the variables to the extent we would have liked. Nonetheless, to whatever extent they are valid, every way we analyze these results appears to tell us that the VAX 8600 offers the best fit for our mix of production and our pocketbook. In every case, the vendors themselves ran the benchmark, either with our people or on their own. All the vendors saw the results of these tests, were invited to criticize our analyses, and could not (or did not) offer alternate interpretations.

There were other factors, of course, beyond the benchmark, that contributed to our decision. A major one was VAX clustering, in which the common file system makes it relatively simple and inexpensive to add computing power to the system under a wide range of circumstances, minimizing the cost of being wrong about the power needed. Another was DECnet/Ethernet, allowing easy communication between the new system and the existing DECSYSTEM-2060s. And finally, the VMS operating system

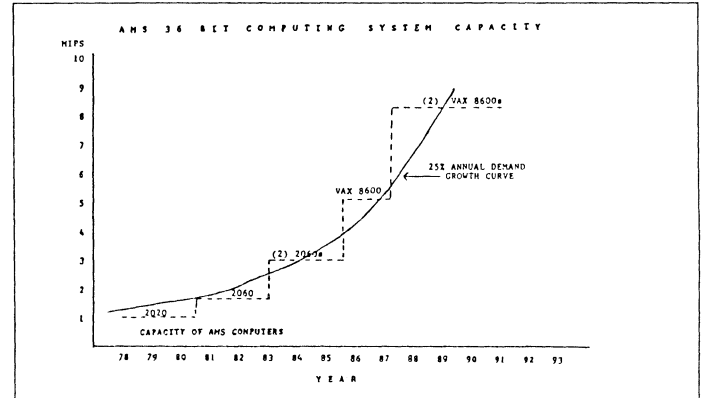


Figure 15: Two 8600s Match the Demand Curve

comes as close to TOPS-20 as any we could find.

A good match for projected demand. And, as Figure 15 shows, two 8600s will certainly fit our expected 25%/year curve of growth.

All we need now is for DEC to stay on course through the rest of this decade. And maybe a little into the next.

January 3, 1986 8:44



LANGUAGES AND TOOLS SIG



Typesetting Articles for the DECUS Proceedings with T_EX¹

Barbara N. Beeton
American Mathematical Society
Providence, Rhode Island

Abstract

The *DECUS Proceedings* have traditionally been published from copy supplied by the authors, prepared according to rules devised for typewritten material. The power of the computer typesetting language T_EX has now been applied to this task, and a formatting package, named DEPROC, has been submitted to the DECUS Program Library for use by authors who have access to a working T_EX system. (The T_EX program and related software, created by Donald Knuth of Stanford, are in the public domain.)

This paper presents the important features of DEPROC and, through examples, shows how it is to be used. Use of DEPROC, which is encouraged, will produce the author's work, nicely typeset, in the standard *Proceedings* format. There is a general description of how the package works and of the mechanical requirements for camera copy of *Proceedings* articles, which will be created on the author's local output device.

No prior knowledge of T_EX is required, but authors using DEPROC will be expected to learn some rudiments, especially if their papers contain special notation or formats such as tables.

The *DECUS Proceedings*, like the conference proceedings of many other organizations, is rushed to publication as quickly as possible so that the material will reach the conference participants and other interested readers before its value is diminished by time. Reproducing author-prepared copy eliminates the considerable bother and expense of typesetting, proofreading and corrections. The published document should be compact, uniform in appearance, and readable, regardless of the kind or quality of printing device available to the author. For these reasons, instructions to authors have heretofore assumed that nothing more elaborate is available than an ordinary typewriter or dot matrix printer.

To enforce uniformity, the author is provided with "model paper", on which are printed (in non-reproducing ink) column and page borders, alignment marks, and instructions for placement of title, author, and the other parts of a proceedings article. The dimensions of the model paper are almost always larger than those of the published *Proceedings*—this permits more text to be packed onto each page, and also improves its appearance or "quality" when photographically reduced, smoothing out the rough edges of letters and symbols generated by a typewriter, dot-matrix printer or other "low-resolution" device.

Within the past few years, advances in laser-printer technology have made good-quality output accessible to a growing number of users, through a widening selection of low-cost output systems based on print engines with 300 dot-per-inch resolution and (relatively) easy-to-use interfaces. Such devices have been attached to most kinds of DEC computers, and drivers now exist to print the output from such programs as Scribe² T_EX and Troff. Most low-end laser printers cannot use paper wider than 8 1/2", however, so even if both a good composition program and output printer had been available, until now an author would have been discouraged from using them for mechanical reasons.

The editor of the *DECUS Proceedings* has now agreed to accept typeset copy printed on such a system at 100% on 8 1/2 × 11" paper, provided it conforms to the published format. This article (which has itself been produced by the technique it describes) introduces a package, DEPROC, designed to prepare *Proceedings* articles using T_EX.

What is T_EX?

T_EX is a public-domain typesetting language created by Donald Knuth of Stanford University. His original aim was to typeset his own books, in particular *The Art of Computer Programming* [ACP], with a quality equal

¹ T_EX is a trademark of the American Mathematical Society.

² Scribe is a trademark of Unilogic Ltd.

no space is wanted between the last item on a line and the first item on the next, a % can be used to suppress it intentionally.

Control sequences, also called macros

A “control sequence” **cs** is an instruction for **T_EX** to perform some action or to produce a particular symbol. A **cs** begins with a backslash, \. There are two types of **cs**-es:

- A “control word” consists of \ followed by one or more letters. It is terminated by any non-letter, including a space. Spaces after a control word disappear, and a special technique (see next paragraph) is required to create an output space after a control word. **\T_EX** is an example of a control word; it produces the **T_EX** logo.
- A “control symbol” consists of \ followed by exactly one non-letter. Since its length is known, no special terminator is required. **\&** is a control symbol to produce an &. **_** (\ followed by a space) is an explicit space, to be used where an output space should follow an element input as a control word.

New **cs**-es can be defined within a document to make input easier or clearer. A few rules governing **cs** names should be observed carefully.

- Case matters; **\csname** is not the same as **\Cname** or **\CSName**. Try to pick a name that means something to you, and is easy to type.
- A new definition will replace an existing one. If you specify a **cs** name that performs an important function in the document formatting, results, as they say “may be unpredictable”.

You can use the **PLAIN** method of defining a **cs**:

```
\def\csname{...something...}
```

but if you are not really familiar with **T_EX** or with **DEPROC**, you might choose a **\csname** that already exists. **DEPROC** provides a simple alternative, that checks to make sure your **\csname** isn’t already reserved for something else:

```
\define\csname{...something...}
```

If this name has been used before, **T_EX** will stop to give you a warning, which you may ignore by responding with a **<CR>**, after which the definition will be made as requested. You may continue to ignore the warning, if you know the prior use won’t affect you, but it’s usually better to change **\csname** to avoid the interruption.

The control symbols **\0, ..., \9** always start out undefined, so they are available for transient use without checking.

Math

Mathematical expressions are input between **\$. . . \$**. Display math is begun and ended with **\$\$. . . \$\$**. For details of math input, see **[FG]**, **[TB]** or **[Joy]**, in order of increasing complexity of expressions handled.

Starting a *DECUS Proceedings* article

The first step in preparing an article is to create a file. The first line in this file should be

```
\input deproc
```

This will cause the formatting definitions to be loaded when the file is input to **T_EX**.

Next, enter the “top matter”. This consists of such things as the title of the article, the author(s) and their addresses, and the abstract.

Title and authors

For an article with a short title and one author, they are input like this:

```
\title{A One-Line Title}
\author{Author Name\
  Author’s Organization\
  City, State}
```

The double backslashes **** indicate line breaks. This technique is also used to break up long titles:

```
\title{Here We Have a Particularly
  Long Title\\That Can’t Possibly
  Fit on a Single Line}
```

This will be set (in a boldface font slightly larger than text size) as

**Here We Have a Particularly Long Title
That Can’t Possibly Fit on a Single Line**

Notice that the way the lines are broken in the input file is not how they appear in the output — only **** matters to **T_EX**. Actually, **T_EX** will break long titles into lines short enough to fit on the page, but a multi-line title usually makes more sense to the reader if the author decides where the line breaks should occur.

For multiple authors, the same **\author** tag is used with **\and** or **\And**:

```
\author{First Author
  \and
  Second Author\
  Common Organization\
  City, State}
```

or

```
\author{First Author\
  First Organization\
  City, State
  \And
  Second Author\
  Second Organization\
  City, State}
```

and so forth, which will appear thus in the output:

**First Author and Second Author
Common Organization
City, State**

or

First Author
First Organization
City, State

Second Author
Second Organization
City, State

Authors' names (the first line, and the first line after `\And`) are printed in boldface; if an author name is to appear on any other line, begin that line with `\bf` (the `TEX` instruction for boldface type).

The title and author of the present paper look like this in the file:

```
\title{Typesetting Articles for the DECUS  
Proceedings with \TeX\footnotemark[1]}  
\author{Barbara N. Beeton\  
  \ANS\  
  Providence, Rhode Island}
```

A couple of additional points can be noted in these few lines. First is the `\footnotemark[1]` which follows `\TeX`. Where is the footnote text? (After all, `TEX` is supposed to take care of formatting, leaving the author free to worry about content.) For `TEX` technical reasons too complicated to explain here, footnotes on items in the top matter are lost unless extraordinary measures are taken, and I hadn't developed a reliable solution in time to install it in the first release. Fortunately, there is a simple alternative, to enter the footnote marker (as shown here) and the footnote text separately. This is covered in more detail in the section on footnotes.

The other item to look at is `\ANS`, which becomes American Mathematical Society in the output. This is an example of a "local definition", something that is not likely to be useful to anyone else, but can save the author a lot of time correcting typing errors. Local definitions that are used throughout an article are best input right after the request to load `DEPROC`:

```
\input deproc  
\define\ANS{American Mathematical Society}  
...
```

Abstract

The abstract is the final part of the top matter.

```
\begin{abstract}  
This is a short summary of what  
the article is about.  
\end{abstract}
```

The heading "Abstract" is provided automatically; don't input it. The abstract may contain more than one paragraph. Paragraphs are separated by a blank line or by `\par`, as usual.

The top matter is now complete. The body of the article follows.

```
\begin{document}  
\maketitle  
(Text of footnotes to the top matter is given here)
```

```
This is the first sentence of article text.  
...  
\end{document}
```

The body of the article

An article can start out with text or with a heading. Three levels of headings are provided by `DEPROC`:

```
\section Section heading\par  
\subsection Subsection heading\par  
\subsubsection Subsubsection heading\par
```

(As usual, a blank line is equivalent to `\par`.) These produce headings (with extra space above and below, not shown here) in the following styles:

Section heading

Subsection heading

Subsubsection heading

The first paragraph following a heading will not be indented in the default style. This may be changed if you prefer, so that all paragraphs will be indented, by specifying

```
\NormalParIndent
```

on the line after `\begin{document}`.

Paragraph indentation can be suppressed throughout by specifying `\NoParIndent` but that would probably make the article too hard to read without other alterations to the style; for an example of unindented style, see the article by Richard Southall [TD] in `TUGboat`.

To suppress indentation on a single paragraph, precede it by `\noindent`.

Footnotes

The use of `\footnotemark` in a title has already been illustrated. A footnote consists of two parts, the mark and the text. These are usually entered as a unit³:

```
... as a unit\footnote[3]{Like this.}
```

When the marks and text must be entered separately^{4,5}, two statements are needed:

```
... entered separately\footnotemark[4,5]  
\footnotetext[4]{Two footnotes ...  
... separately.}\footnotetext[5]{This ...}
```

³ Like this.

⁴ Two footnotes assigned to one item must be marked and entered separately

⁵ This situation is slightly different from what happens in the top matter.

Note the use of the % after the `\footnotemark`—care must be taken to avoid spaces that might creep unwanted into the text, and suppressing the `(CR)` at the end of the line is probably the easiest way.

Marks are set as superscripts in the default style, but this can be changed by specifying

```
\footnotemarkstext
just after \input deproc.
```

Quotations

Short quotations, of less than a paragraph, are set with

```
\begin{quote}
If you can't fix it, ... {\em Button}
\end{quote}
```

and look like this:

If you can't fix it, call it a feature. *Button*

For longer quotations, use

```
\begin{quotation}
...
\end{quotation}
```

in a similar manner, separating paragraphs with blank lines as usual.

Lists

Itemized and enumerated lists occur in many *DECUS Proceedings* articles. \LaTeX provides automatic counters and up to four levels of nesting. *DEPROC* users will have to make do with two levels, and no automatic counters. The general input structure is this:

```
\item[label] text
\itemitem[label] text
```

Any desired labels can be used;

- `\bullet`,
- `\circ`, and
- en-dash (input as `--`)

are common choices, as are numerals and single letters.

Here is a short example of a two-level list.

```
\item[\bullet] first item
\item[\bullet] second item
\itemitem[\circ] new level
\itemitem[\circ] one more
\item[\bullet] back a level
```

Here's what this looks like, after padding out the text a bit to show how longer items look.

- The first item in this list isn't particularly interesting, but it has to be long enough to make two lines.
- The second item isn't either.
 - Even going to a new level doesn't add very much excitement to this exercise.
 - We'll do one more at this level.
- Then we'll go back a level to finish things off.

Each item comprises one paragraph; an unlabeled paragraph can be produced by specifying an empty label.

Extra space above and below a list can be obtained by specifying `\smallskip` or `\medskip`. It is usually advisable either to insert extra space after a list or to apply `\noindent` to the following paragraph, for clarity. For example, the sample list above ended this way:

```
...
... finish things off.
\smallskip
\noindent
```

Figures

Figures come in several sizes and shapes:

- small figures which can be set in place, i.e., in the same relative position where they occur in the input file;
- one-column figures to be set at the top or bottom of the first available column;
- double-column figures to be set at the top or bottom of the first available page;
- full-page figures.

Not all of these formats are supported yet by *DEPROC*. In particular, two-column figures cannot be incorporated into text pages, although there is a mechanism for leaving space at the bottom of a page, so that separately-prepared figures can be pasted in later. Similarly, space (i.e. blank pages) can be reserved for full-page figures.

One-column figures

To get a single-column, in-line figure, enter

```
\begin{figure}
content of figure
\caption{Figure n. Caption text}
\end{figure}
```

“Figure” and the figure number must be input. (This is not necessary in \LaTeX , where figures are numbered by a counter reserved for that purpose.) If the figure will be prepared separately and pasted in, space can be reserved:

```
\begin{figure}
\vspace{2.5in}
\caption{Figure n. Caption text}
\end{figure}
```

Space equivalent to a blank line is skipped above and below an in-line figure, and a half-line between the figure and the caption, so the dimension given with `\vspace` should be precisely the size of the item to be pasted in.

If insufficient space remains in the column to accommodate the figure in-line, it will automatically be shifted to the top of the next available column. Figures can be placed at the tops of columns explicitly by specifying

```
\begin{topfigure}
...
\end{topfigure}
```

A `topfigure` will be set at the top of the current column, if space is available, otherwise at the top of the

next available column. When both regular **figures** and **topfigures** are used concurrently, they may possibly be set out of order, depending on their sizes and the other contents of the article; it is safer to use only one style or the other within a single article, and even then, to check the results carefully before sending your article off to be published.

Two-column figures

Double-column figures, as mentioned earlier, cannot yet be incorporated into text pages by **DEPROC**. However, space can be reserved for figures prepared separately; see the next section, full-page figures.

To reserve space, you must know the page number, or the relative page, on which you want to place the figure; you must also know the size of the figure, including its caption. Space will be reserved by shortening the text area on the specified page, so double-column figures at present will be placed only at the bottom of a page.

The command to shorten a page must appear on a line by itself.

```
\ShortenPage n by dimen.
```

or

```
\ShortenPage +n by dimen.
```

(The spaces and the period at the end are necessary.) The first form would be used if you know the page number; otherwise, *+n* equals the number of pages *after the one on which the command is given*. For example, if you give the command on page 1, and the figure is to appear on page 3, *n* = 2. For *dimen*, enter the height of the figure as measured, plus about .2" for appearance (extra space is not added, unlike **figure** and **topfigure**). This height may also be expressed in terms of number of text lines:

```
... by 12\lines.
```

will reserve the same space occupied by 12 lines of text, about 2".

\ShortenPage will not shorten the first page of an article, and only one **\ShortenPage** command can be in effect at once. But a page that is being shortened can also include a command **\ShortenPage +1** to adjust the next page.

Full-page figures

Full-page figures are partially supported by **DEPROC**. Pages are reserved during the main **TEX** run, and figures can be prepared separately, to replace the reserved pages. To reserve a page for a figure, on the page before the one to be reserved (this will most likely be near the first reference to the figure), enter a line of the form

```
\reservefigurepages[1]{Replace by Figure N}
```

Multiple pages may be reserved at once, say for a program listing:

```
\reservefigurepages[3]{Listing of XYPROG}
```

Each reserved page will contain only a page number and a two-line message:

Page reserved for figures

Text input with **\reservefigurepages**

A separate file can be created for figure input, if you wish to use any of the **DEPROC** facilities. It should begin with these three lines:

```
\input deproc
\figurepages
\pageno=nn
```

The value assigned to **\pageno** should be the page number to appear on the first figure page; subsequent page numbers can be reset in the same manner as appropriate. **\figurepages** will redefine the output format to be one column 5" wide. This page format is called **\onenarrow**. Two other single-column page formats are available:

```
\onecol is the same width as the two-column page;
\onemedium is 6" wide.
```

(The normal output format is called **\twocol**.) Any one of these output formats can be specified just after a page break, before anything has been set on the new page. A page break can be forced by the command **\newpage**. (In two-column format, a column break is forced by **\newcol**; in one-column format, **\newcol** is equivalent to **\newpage**.)

Within the figure input file, most **DEPROC** options are available for use. Exceptions are the top matter commands (**\author**, **\title** and **\abstract**), **\begin{document}** and **\end{document}**, and **\maketitle**. Almost all plain **TEX** facilities can be used, although it is advisable to check the **DEPROC** definitions if you intend to make changes in page format.

When you have completed the file of figure input, end it with the command **\bye**.

Tables

There is no support yet in **DEPROC** for tables. Tables can be coded using plain **TEX** rules for tabbed or **\halign** environments. See *The TEXbook* for details.

Verbatim

Verbatim items are printed in so-called "typewriter" style, using **TEX**'s **\tt** font. In-text verbatim items are enclosed in vertical bars |...|; blocks of verbatim code are delimited by

```
\begintt
...
\endtt
```

\begintt and **\endtt** should be on lines by themselves. Within verbatim mode, **<CR>**s are obeyed as line breaks, not spaces. An input line that is too long for the current column width will be broken at a space if possible, and the remainder of the line hanging indented on the next output line; since this may change the meaning of the

verbatim passage, such passages should be checked with special care in the output. Overlong lines also frequently result in overfull `\hboxes`, which are indicated clearly on the output by black boxes: ■.

A passage between `\begintt... \endtt` is treated as a unit by `TEX`—if it is too long for the vertical space available, it either will be carried over as a unit to the next column or page, or will result in an overfull `\vbox`, which will be noted only in the transcript of the `TEX` run. In such a case, the best remedy is to break the passage in two, by inserting another `\endtt \begintt`.

Verbatim mode is suitable for program listings, indicating keyboarding instructions, file names, and similar uses.

References, bibliography

References in text to items in the bibliography are input as

```
\bibref{label}
```

where the label that will be used in the bibliography must be entered; there is no automatic association between the two as in `LATEX`. For a label “ABC”, the text reference will be rendered [ABC].

Before you start to input the reference list, some housekeeping is required—you must decide what you want the list to look like. This is what the input looks like for one of the items in the reference list at the end of this article:

```
\bibitem[TB] Knuth, Donald E., \TB,  
Addison-Wesley and \AMS, 1979.
```

(`\TB` and `\AMS` are among the local definitions for this article.) Default output looks like this:

```
[TB] Knuth, Donald E., The TEXbook, Addison-Wesley and  
American Mathematical Society, 1979.
```

If you prefer the reference labels to be enclosed in parentheses instead of square brackets, `\bibbrackets()` will replace them. (`\bibbrackets[]` will restore the square brackets.) Brackets may be eliminated by `\omitbibbrackets`. If your labels are simply numbers, 1, ..., *n*, brackets will be omitted automatically (see below).

Now give the command to begin the reference list

```
\Bibliography{widest label}
```

entering the widest label that will actually be used in the list; this will be used to control the formatting.

If you do not wish to use labels, substitute `\omit` for the widest label. (The [...] are still required in the context of `\bibitem`.)

```
\bibitem[] Knuth, Donald E., ...
```

will result in

```
Knuth, Donald E., The TEXbook, Addison-Wesley and American  
Mathematical Society, 1979.
```

If your labels are numeric, substitute `\numeric` for the widest label. The input

```
\bibitem[2] Knuth, Donald E., ...
```

will now look like this:

```
2. Knuth, Donald E., The TEXbook, Addison-Wesley and  
American Mathematical Society, 1979.
```

Caveats

`DEPROC` and this article were created on a `DECSYSTEM-20` at the American Mathematical Society, running `TEX` version 1.5. The `AMS` installation is standard in all ways except that a few memory cells have been increased for reasons not relevant to, and not affecting the performance of, `DEPROC`.

With one exception, none of the changes to the `TEX` program since version 1.0 should have any noticeable effect on an article produced with `DEPROC`. The exception is large, complex tables—tables incorporating many boxes and rules require large amounts of `TEX` memory. Memory management was radically changed in version 1.3 to make more memory available to the user without actually changing the physical memory allotment. (Otherwise, if you run out of memory, the most likely cause is an input error.)

Although thorough testing has been attempted, no one outside the `AMS` has tried to use `DEPROC` yet, so bugs are sure to be found. In fact, the version of `DEPROC` first placed in the Program Library should best be considered a beta test version. If you find a bug, please communicate it to the author, accompanied by an example which demonstrates the bug as simply as possible. Suggestions for improvements are also welcome. Send everything to

Barbara Beeton
American Mathematical Society
P. O. Box 6248
Providence, RI 02940

References

- [ACP] Knuth, Donald E., *The Art of Computer Programming*, Addison-Wesley, Vol. 2, second edition, 1981.
- [TB] Knuth, Donald E., *The T_EXbook*, Addison-Wesley and American Mathematical Society, 1979.
- [LT] Lamport, Leslie, *L_AT_EX, A document preparation system*, Addison-Wesley, 1985.
- [TD] Southall, Richard, First principles of typographic design for document production, *TUGboat* Vol. 5 (1984), No. 2, 79–90; Corrigenda, Vol. 6 (1985), No. 1, p. 6.
- [FG] Samuel, Arthur, *First Grade T_EX*, `TEX` Users Group, 1984.
- [Joy] Spivak, Michael, *The Joy of T_EX*, American Mathematical Society, 1980; new edition in preparation, 1985.
- [TUB] *TUGboat, the Newsletter of the T_EX Users Group*, `TEX` Users Group, c/o American Mathematical Society, P. O. Box 9506, Providence, RI, 02940.

Robert Lanphar
 Senior Scientist
 Hughes Aircraft Company
 Fullerton, California
 (714) 732-6207

I Introduction

Numerous other speakers and authors at DECUS Symposia present software engineering concepts which are amenable to the building of small products. This paper presents a functionally complete software engineering environment which is in use at Hughes Aircraft Company. This environment has as its goal the support of the many diversified product lines of the company, the support of both classified and unclassified projects, project staffing ranging from 5 to 200, language requirements from assembler through Ada(1), and application sizes ranging from several thousand to over one million lines of code. The talk focuses on those issues which have made the Hughes Software Engineering Environment (Hughes SE²) a success. The paper presents the Hughes SE² in terms of the justification, evolution, current configuration, and a summary(2).

II Justification

The Hughes SE² is a creation of necessity resulting from the ever-increasing complexity of software required for large scale defense systems. For over 20 years, Hughes has been involved with the development of large command and control systems requiring hundreds of thousands of lines of code and upwards of two hundred software engineers. (See Table 1). The major contracts have been largely fixed-price, thereby making the proper management of software development a matter of survival. Thus, Hughes considers its environment a vital resource in its ability to build large-scale military systems.

- (1) Ada is a registered trademark of the U.S. Government, Ada Joint Program Office
- (2) SE² is a registered trademark of Hughes Aircraft Company

Developing a competitive, state-of-the-art Hughes SE² has proved a difficult process. The fact that the environment must satisfy customers from the Air Force, Navy, Marines, Army, and foreign nations has placed extraordinary demands on the capabilities. As the environment expands to provide more automated capability, the demand of the users increases. A direct outcome of this explosive growth is a four-year plan to communicate and coordinate the needs and goals of the different user communities. However, it is not planning alone, but the day-to-day use on many projects, the constant assessment of galloping technology, and the consistent solving of user problems which has made Hughes SE² so necessary for its 1700 users.

The Hughes SE² is not just software tools but a collection of entities (Figure 1) working in concert to enable the quality production of software products. The ORGANIZATION entity consists of approximately 21 large projects (the paying customers), a technology lab which assists in the orderly evolution of the environment, and various traditional business management functions. The SOFTWARE PRODUCTS entity consists of approximately seven million lines of code and their associated documents and plans, both deliverable and non-deliverable, which have been successfully produced. The FACILITIES entity consists of the various computers (16 large VAX systems, corporate mainframes, and workstations), buildings, and networking which assist in the quality production of software products. The TOOLS entity consists of approximately 47 software development tools and the project specific tools. Projects are allowed some degree of freedom in tailoring their project specific environment to satisfy their objectives. The METHODS entity consists of approximately 74 formal standards and procedures, Hughes Practices, and the installed base of engineering expertise. The methods govern the way software is produced. The PEOPLE entity consists of approximately 1700 software engineers with extensive experience and education, interacting with the environment to produce quality software products.

The entire environment must be considered in deploying the Hughes SE² to ensure that the quality production of software products is accentuated and not debilitated.

<u>Product Line</u>	<u>Language</u>	<u>Number of Lines of Code</u>
o Command and Control Information Systems	FORTRAN	745 K
o COMM Systems	CMS-2, FORTRAN	748 K
o Sonar Systems	CMS-2, Ultra-32	672 K
o Radar Control Systems	Jovial, FORTRAN	365 K
o Air Defense Systems	Jovial	4000 K

Table 1: Typical Hughes SE² Products And Their Characteristics

III Evolution

Hughes SE² has evolved from a card-oriented batch environment (pre-1977) to an interactive software development support environment (1977-1984) to a workstation-based software engineering environment (post 1984) (see Figure 2). The architectural description of this environment is explained in the context of these time periods or stages. Currently, the environment is in a transition period moving from a centralized terminal environment to a distributed workstation-based software engineering environment, where all the activities of the software engineer are supported. This transition is necessary to take advantage of today's technology, which is required to meet the needs of complex military systems development. Hughes' expertise in developing software engineering environments grew along with the evolution. Hughes' philosophy in building the environment has emphasized methods, procedures, and techniques that work in a software production environment, coupled with continual research in new areas of the software engineering field.

III.1 Pre-1977 Evolution

Prior to 1977 the software environment was the target environment. Coding forms were prepared, cards punched, and (perhaps) paper tape was cut. The programs were compiled from cards and paper tape to magnetic tape or disk to form the executing system. The software was debugged by using the target computer front panel controls and by obtaining memory dumps.

Because the system generation process was lengthy, patch decks were used to make small changes to the system. The patch decks were incorporated into the source code (by punching new source code cards) every week or two. That is, the system was "baselined" every one or two weeks and a new set of source listings was produced. This process was normally performed overnight.

The primary methodologies used were block diagrams for requirements and flowcharts, and Program Design Language (PDL) for design. The primary analysis tool was simulation. The source code listings comprised the bulk of the maintenance documentation and secretaries typed the documents.

This software development environment was adequate for the sixties because the systems were relatively small and less sophisticated. A typical system was usually built by a half-dozen software engineers in two years. The software executed in a single computer, or in two or three simply-connected, very small computers. Hardware was expensive and documentation was minimal. Software accounted for less than two percent of the total system cost.

During the early seventies, this began to change. Software began to be used as the system "glue". By 1977 software accounted for 6-10 percent of the total system cost. A typical system required 30 software engineers for two or three years. Some systems contained computers which were arranged in a geographically dispersed network. Some systems contained multiple centralized groups of computers. For example, an Air Defense system usually contained four colocated computers per site with perhaps six geographically dispersed sites in the network.

This ever-increasing emphasis on software put greater pressure on the software methodologies and documentation. The size of the projects alone caused significant problems in communications. The primary problem was in the area of software design. The high level design methodology shifted from flowcharts to PDL, which had the versatility and data orientation that seemed to be needed, but lacked the graphics vital to rapid communications. The Structured Design Methodology was introduced in 1974 because it had the needed data orientation and graphics and appeared to be one of the better software design methodologies. The first use of this methodology proved successful on a large Air Defense project, but it took four more years to transfer this technology to all of the Hughes projects.

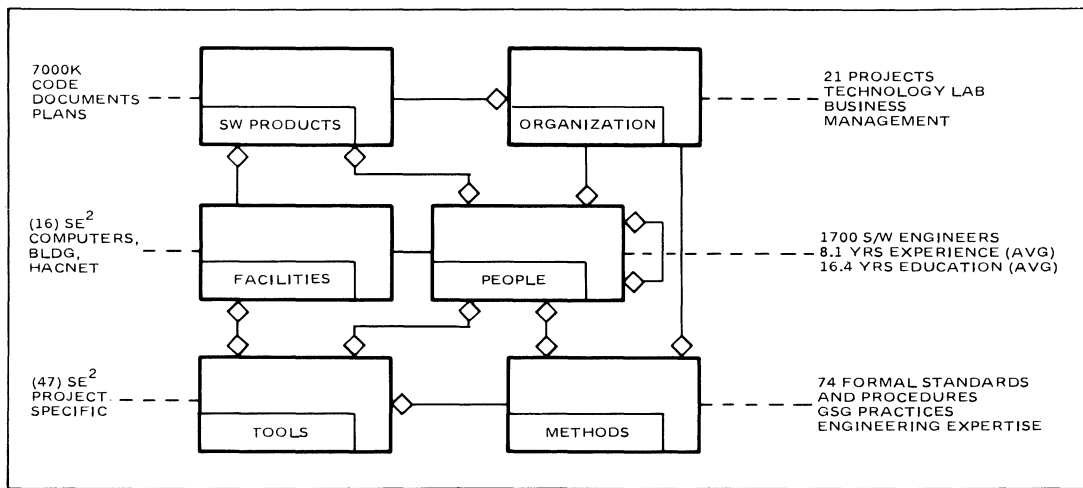


Figure 1: The Hughes SE² is more than tools and methods, it is people properly trained in the ways in which the environment elements can best serve them.

By 1977 the target systems were larger, more complex, and distributed. Block diagrams were used in requirements. The Structured Design Methodology was the primary technique used in top level design. A mixture of PDL, flowcharts, and Hierarchical Input Process Output (HIPO) charts was used in detailed design. Structured Walkthroughs and simulation were used in all phases. Secretaries still typed the documents. The corporate mainframes (non-target machines) began to be used to help with the documentation and system analysis efforts. Although most usage was still card-oriented batch, some timeshare usage on teletype terminals had begun in an effort to speed up turnaround time.

Because of the many engineers and the heavy text requirements, an interactive, multiuser environment was needed. Although the corporate mainframes might have been used for such an environment, greater support was needed than they could provide. It became important to move engineers on and off a system rapidly, to obtain resources (e.g., printers/plotters) and listings rapidly, and to avoid the slowdown caused by the month-end corporate financial reports. Therefore, Hughes looked to Digital Equipment Corporation (DEC) and the Western Electric Programmers Work Bench (PWB) to provide a cost-effective software development environment(3).

(3) PWB is a registered trademark of Bell Laboratories

III.2 1977 to 1984 Evolution

Interactive software development began in 1977 with the purchase of three DEC PDP 11/70 computers. However, before the third 11/70 could be installed, the environment evolved to the DEC VAX 11/780. The 11/70 machines were adequate for small projects, but not for larger projects. The VAX machines were selected based on price/performance, the number of interactive users that could be supported, the virtual memory organization, and the availability of the Programmers Work Bench (PWB). The VAX became the standard computer.

The first terminals selected for standard tasks were BEEHIVE and SOROC. Other terminals were selected for experimentation, including the ADM 500 and the Interactive Systems Corporation Intext terminal. Other terminals were selected for special tasks including the HP2647/48, and the GENISCO. By 1980, the DEC VT100 terminal was the standard terminal because of its reasonable price/performance and reliability. The standard graphics terminal was the VT100 with a Selanar graphics card (made by Selanar, Inc). This combination of terminal and graphics card was selected based on price/performance, reliability, and the ability of the graphics card to emulate the (defacto standard) Tektronix 4010 graphics.

In 1977, there were 70 interactive users, 14 terminals, and 3 computers in the software engineering environment. This expanded to 1500 users, 665 terminals, and 16 computers by 1984. The mission of the environment during this time was to support software development for large real-time military systems, i.e., to write code and documentation.

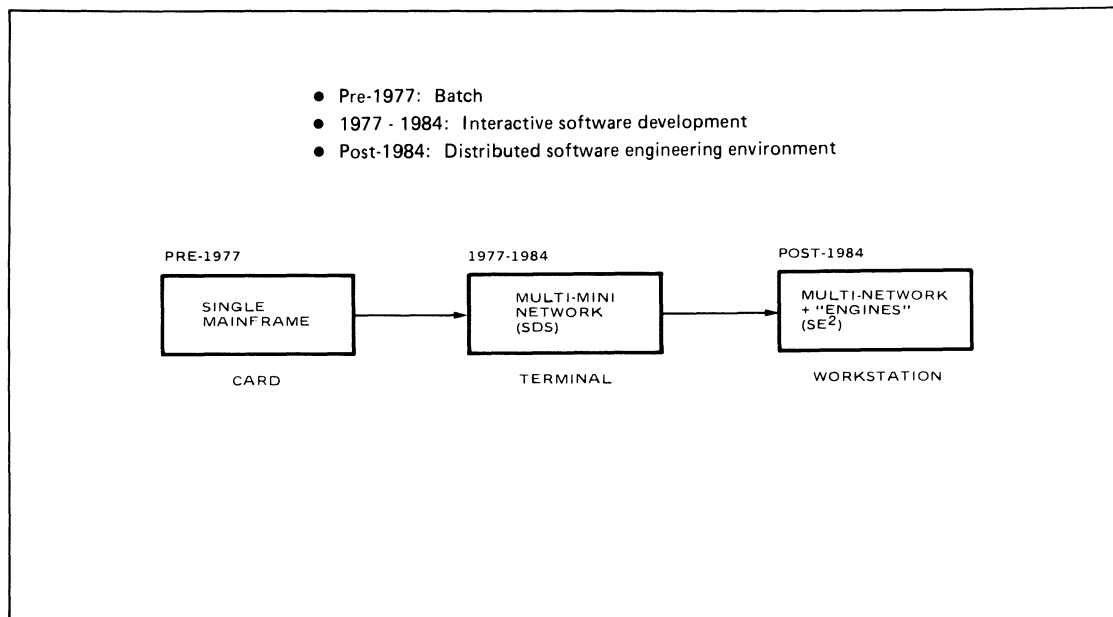


Figure 2: The Hughes SE² has progressed through three stages to satisfy the demands of the users.

During the 1977 to 1984 period, the software development environment evolved at the average rate of two VAX computers, 90 terminals, and 200 users per year. Offices were wired to a communications switch so terminals could be located more conveniently in the software engineer's office (i.e., remote from the VAX center). Because of space problems, whole projects were moved to remote locations along with their VAX. DECNET was installed to provide communications among the local VAXs and the remote VAXs. The disk farm was installed to reduce the effective cost of disk space.

The reason for this growth was to provide high interactivity and fast turnaround to the projects, and to develop and refine the software methodologies and tools. For example, the structure chart (graphics) part of the Structured Design Methodology, while vital to the technique, was difficult to modify and maintain. Therefore, the Structure Chart Graphics (SCG) tool was developed in 1978 to help draw and maintain the charts. Although the tool looked like it would help, it did little because it was only a picture editor and not a design tool. However, SCG did show that graphics could be valuable in software engineering. The Automated Interactive Design and Evaluation System (AIDES) was deployed in 1982 to overcome the problems found with SCG. Although AIDES draws structure charts, it is also a design tool. All structure chart pictures are drawn automatically from the design database. From this experience we learned that tools must reflect the underlying methodology. That is, simple editors (text or graphics) are not, themselves, a methodology any more than a hammer is a methodology.

Requirements issues were becoming more of a problem. This was either a result of design problems coming under control, or the systems getting larger, or both. The Structured Analysis Methodology was selected for the requirements area because it was related to the familiar Structured Design Methodology. It also seemed to be one of the best methodologies available for real-time systems. In developing the System Engineer's Workbench (SEWB) to support requirements development and analysis, it was found that the VT100 terminals with Selanar boards on 2400 baud communication lines were inadequate. The drawings were coarse and the response slow. Therefore, Apollo workstations were brought in to help solve these problems. The Apollo was selected because it was about the only workstation available at the time. The closed architecture and proprietary operating system of the Apollo would eventually cause Hughes to look at other workstations.

By 1984 communication speed problems and raw processor power problems were becoming apparent (an estimated 128 VAX's would be needed by December 1987), only limited support for the DoD classified project environments could be provided (which were getting larger, but not large enough to need an individual VAX); and, most important, more direct support was needed to assist the software engineer in performing his total job. Therefore, in April 1984, the Software Development System (SDS) Architecture Group was formed to define the next generation software engineering environment.

III.3 Post-1984 Evolution

The next generation Hughes SE² was defined by the SDS Architecture Committee Report of August 1984. The report findings focused on improving the productivity of our software engineers in 1987 by 50% over 1983 levels. This process was to be achieved through a combination of techniques including: 100% support for the software engineers job, tool integration, introduction of the workstation form factor, the use of a common operating system (UNIX), the use of a communications infra-structure (ethernet, TCP/IP), and making sure that the new technology was introduced to the user community in an orderly manner(4). A Deployment Plan was established to satisfy the technical plan. Hughes is well into the Deployment Plan with 90 workstations installed, with the objective of having 450 workstations by the end of 1987.

The mission of the pre-1984 software environment was to provide a software development environment for real-time systems. The new environment is composed of software, hardware, courseware (training), and people dedicated to supporting all of the roles of the software engineer (not just software development). The roles include requirements, design, code, and test plus communication, project management, documentation, configuration control, and report and presentation generation. The environment specifically excludes secretarial roles (e.g., general typing), senior management roles (e.g., corporate business forecasting), and supervisory roles (e.g., performance appraisals). Therefore, the mission is complete support of the software engineer in developing real-time military systems.

Implementation of the advanced Hughes SE² began in 1984 with the purchase of four SUN 2/120 workstations. The SUN workstation was selected because of its processing power, display capabilities, open architecture, and adherence to the popular standards. The processing power and high resolution display capabilities are essential to the proper functioning of the more sophisticated graphics tools. The open architecture allows foreign boards to be included in the SUN. For example, Multibus allows a direct connection to our air defense target system.

(4) UNIX is a registered trademark of Bell Laboratories

Adherence to the popular standards is most important. For example, Unix and standard Fortran 77 improve the portability of our tools. Ethernet, with the popular TCP/IP, provides the high speed communications in the workstation network and between the workstation networks, process engines, central VAX system, target systems, and the Hughes corporate network (see Figure 3). The ability to add process engines is limited by the communications protocol, not by the operating system of the engine. This provides the ability to evolve smoothly and to create a virtual network (i.e., network login, not machine login). Therefore, all VAX-based tools are available to the workstation user. For small, classified environments, the MicroVAX (rather than an 11/780) can be used to support the VAX based tools in a more cost effective manner. When and if VAX-based tools are rehosted to another computer, they can look and act the same as they did before rehosting. This greatly reduces the shock of changing hardware.

Most importantly, arguments over whether a single (integrated) database is better than multiple databases go away. In the same way that the user sees a virtual network (although the network is composed of many machines), the user will see a virtual database. That is, the user will see objects (not files) in the virtual network. While it is important that the user see objects and not files, it is not important to know where the object is, or whether it is part of a larger database or simply a single file. Therefore, the environment may evolve to a single database or to many individual databases based on factors other than the user's view of the system.

IV The Current Hughes SE²

This section now takes three slices through the current Hughes SE² to provide visibility into how the collection of resources are administered to satisfy the needs of a large population of software engineers. Concepts to be expanded upon include:

- a. The ability to assign hardware resource sufficient to satisfy requirements.
- b. The ability to introduce technology in an orderly way.
- c. Support for the full life cycle is mandatory.
- d. Integration is necessary.
- e. A common user interface provides an easily learned paradigm.
- f. Training greatly amplifies the correct dissemination and use of the resources.
- g. Established maintenance mechanisms ensure that the environment evolves in an orderly way.

IV.1 Hughes SE² Hardware

The evolving Hughes SE² hardware architecture addresses the needs for optimized assignment of hardware and improved information bandwidth.

Figure 4 provides a diagrammatic view of the hardware architecture. The tailorability of this architecture to satisfy project needs is immediately apparent when one envisions portions of this network resource being optimally assigned to specific projects. The ethernet communication network provides the ability to configure a distributed environment to meet the varied needs of the projects. As opposed to the 1984 situation where a whole computer, typically a VAX 11/78X, was dedicated to a classified project, now a grouping of workstations and special servers can be optimally configured.

The ethernet communication network establishes the basis for tailoring the distributed environment. This network, employing TCP/IP protocol, can be mixed and matched to fulfill the requirements of the user. For example, the Micro-VAX II special purpose server can easily be replaced with a larger, more powerful computer to provide better throughput of the VAX-designated functions. If a JOVIAL compile engine is required for the development environment, the IBM PC AT/370 can be added to the network. Similarly, microprocessor development systems and other target hardware systems can be added as necessary, to the network.

The environment is manageable, since the physical requirements are minimized. That is, facility size, power demands, and environmental needs meet the specific requirements of the project. As project needs change, hardware components can be added or deleted from the distributed system. For example, during the requirements and design phase of the software lifecycle, the classified facility may include the SUN local area network only, because JOVIAL cross compilers are not necessary at that time. Allowing flexibility of this sort makes for a cost effective and manageable distributed computer system.

The improved information bandwidth is primarily achieved through the incorporation of the workstation as the software engineer's window into the network. The high resolution (1000² addressable pixels), large display (19") window provides for the display of much more of the software engineer's work on the screen at once. As opposed to the prior art VT100, which limited the viewing area, the workstation allows for enhanced tradeoffs in a design iteration. The higher speed ethernet eliminates the 2400 baud choke which was a characteristic of the ASCII terminal.

IV.2 Hughes SE² Software

The tools which comprise the Hughes SE² have evolved to a state where full support for the software engineer is provided, tool integration is fact, and a common user interface provides an easily learned paradigm. Full life cycle support is provided, to mimic the DoD standard 2167 phases including: system software requirements, preliminary and detailed design, code and unit test, and system integration and testing. Tool integration allows the tools in the life cycle phases to pass on information into previous or subsequent lifecycle phase tools. This minimizes costly and error-prone re-input of design information. The common user interface is currently under development to provide the user with a standard way of interacting with the tools.

The Hughes SE² supports the software development life cycle, as shown in the Figure 5.

Software development begins with requirements generation. The System Engineer's Workbench (SEWB) is used to generate data flow diagrams (DFD), a data dictionary, and mini-specs. After the data has been analyzed for completeness and consistency, the requirements document is automatically generated and the requirements database is updated using the Automated Specification Analysis Tool (ASAT). The requirements document can be generated in MIL-STD-1679 or DoD-STD-2167 format.

Next, the Analysis and Design Interface Transforms (ADIT) tool is used to provide an initial translation of the requirements database to the design database and the design data dictionary. The Automated Interactive Design and Evaluation System (AIDES) is used to refine the design and to provide summary reports and structure chart drawings. The structure charts are usually produced in a 6 by 4 foot size for design walkthroughs and in an 8-1/2 by 11 inch size for final delivery. In addition, the Data Dictionary (DD) editor is used to refine the design data dictionary information and the AIDES Metrics (AIM) tool measures the completeness and complexity of the design.

When the design has been completed, another ADIT transform is used to provide an initial translation of the design and design data dictionary databases to the detailed design database. The translation generates module prologues, skeletal PDLs in Software Design and Documentation Language (SDDL) format, and exception information based on differences between the design database and the design data dictionary. A standard text editor is used to refine the PDL, and the SDDL tool is used to provide PDL cross-reference and summary reports.

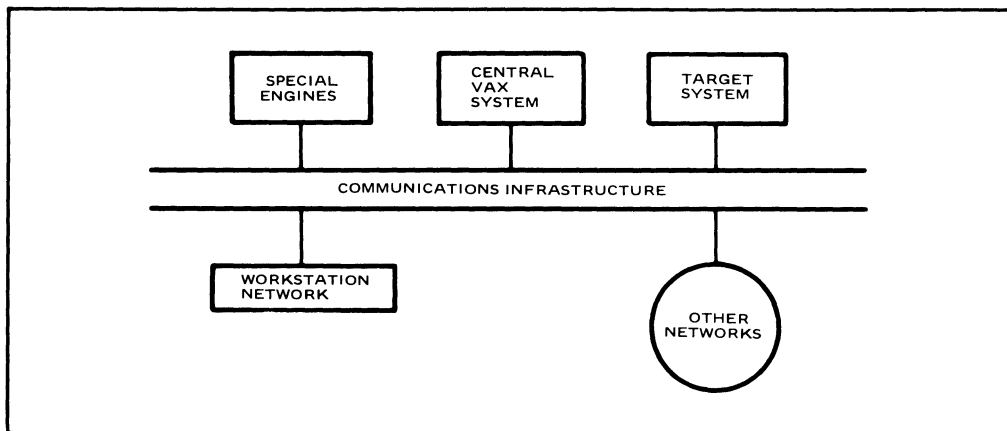


Figure 3: Distributed Software Engineering Environment. An open-architecture, virtual network connects the workstation user to needed resources.

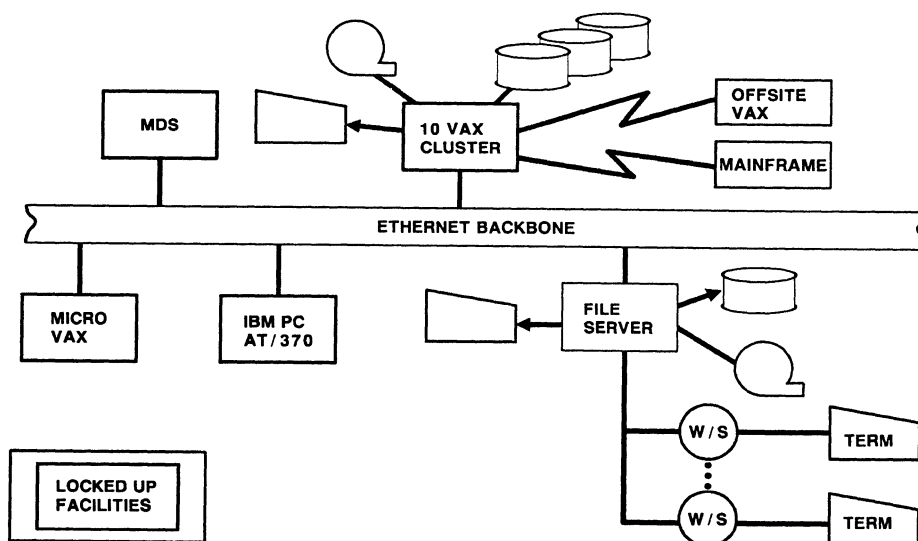


Figure 4: The Hughes SE² Hardware Architecture is designed with tailorability to meet project needs.

The translation of PDL into the programming language of choice is a manual process accentuated through the use of a text editor with macro capability. However, the Design Recovery System (DRS) is used to verify that the code produced matches the design through an automatic process of generating the as built design (in a graphical AIDES context) from the actual source code. A software module is entered into the Change Control And Reporting System (CCARS) after it has been coded and compiled without error. CCARS provides rigid control of the system elements, provides version control, and generates status reports. To generate an executable version of a system, CCARS provides the desired version of the software in source form, the source code is compiled by the applicable compiler, and an executable image is produced.

If the target system is a local system (e.g., VAX), the image is executed immediately. The interactive debug capabilities are used and reports (e.g., table dumps) are generated as necessary to test the system. If the target system is a remote system (e.g., Hughes MX computer or the Navy's AN/UYK series computer), the image is first executed using a target computer simulator. Again, the interactive debug capabilities are used to test the (simulated) system.

The final testing is performed on the remote system by transferring the executable image to magnetic tape, walking the tape to the target facility, and loading the executable image on the remote target system. The data recording and system monitor functions in the target program are usually used to extract table data (i.e., dumps) and to monitor the execution speed of the various functions. In addition, the Debugger for Common ADGE Networks (DECON) for Hughes MX computers or the Remote Access Interactive Debugger (RAID) for Navy AN/UYK computers is used to help debug distributed networks. DECON and RAID are used to set breakpoints, initiate variables, etc., and to synchronize data collection in a distributed system. All testing and maintenance is supported using Library Change Request (LCRs) which report problems encountered with the software. The LCRs are used by CCARS to monitor, control, and report changes to the software.

There are several tools which are currently not integrated into the dynamic view, but nevertheless provide valuable software project checks and balances. The SEER tool implements the Jensen model (Dr. Randy Jensen of Hughes) for software life cycle staffing and costing. Project 2 is an example of a PERT or critical path method tool which is used for project resource tracking. The Management Information and Reporting (MIRG) tool is a management reporting aid which provides the capability to retrieve standard status reports dealing with project productivity statistics. Various modeling tools are available for evaluating target environment budgets in terms of a percentage of resource saturation, such as cpu, memory, or i/o capacity. These tools are being integrated into the environment currently and will greatly enhance management's view of the software development process.

IV.3 Hughes SE² Peopleware

The Hughes SE² is a success because of the commitment which has been made to the proper introduction of the environment to the user community. An evolving set of training classes is presented which covers the entire range of capabilities available in the environment. Classes are constantly being reorganized to make sure that information is transferred to those in need. New forms of information dissemination are being evaluated to make sure that the user is informed in a timely fashion. Examples of these new forms of information dissemination include videotape and computer assisted instruction. The training component of the supporting infrastructure ensures that the environment is used in a productive and conscientious fashion.

The user's first line of support comes from the user consultants. Most of the time (75 percent by survey) the user can obtain the answer to questions/problems directly over the phone. This is a real boon to productivity and also provides a valuable indication of what needs to be fixed or better communicated. The user consultants are the user advocates, smoothing out the path towards productive use of the environment.

The orderly evolution of the Hughes SE² requires that the proper information be available to the management level making the decision. This process has been constructed from several information items with differing frequencies of checks, balances, dissemination, and management involvement. Figure 6 provides a visual representation of the various processes which are executed to guarantee that informed decision making is the rule. On an annual basis a document is prepared which details the proposed tactical plans and their justification. An annual "State of SHARE" (SHARE is the name of the computer facility) presentation is given to the entire user community highlighting the major components of the plan. On a six month interval a workload forecast is constructed, individual project management is interviewed, computer or workstation groups are reviewed, and facility plans are updated. On a monthly basis the Steering Committee meets to review policy statements and workload allocations. Project and facility utilization is formally communicated to management, vendor performance is reviewed, distributed offsite facilities are reviewed, and the software tools are fixed and enhanced by the Software Change Review Board (SCRB). A semi-monthly newspaper is published and distributed to all users. Weekly, system messages are reviewed for information content, and internal facility and project reviews are held. An open door policy is extended to all users and management. This process of informed decision making ensures that technology is introduced into the environment in an orderly fashion, always ensuring that the near-term needs of the project communities are met.

V Summary

The Hughes SE² supports the development of large real-time systems requiring thousands of lines of code and upwards of a couple hundred software engineers per project. The environment has evolved from card-oriented batch using target machines, through an interactive software development capabilities using minicomputers, to a workstation based software engineering environment.

The early environment was primitive (i.e., used coding forms, punch cards, and paper tape), but was adequate for the development of the relatively smaller and less complex systems of the time. However, by 1977 the target systems were larger, more complex, and distributed. Therefore, new methodologies were introduced and the corporate mainframes were used to help develop the systems. Because of the many engineers and the heavy text requirements, Hughes looked to Digital Equipment Corporation (DEC) and the Western Electric Programmers Work Bench (PWB) to provide the needed interactive, multiuser environment.

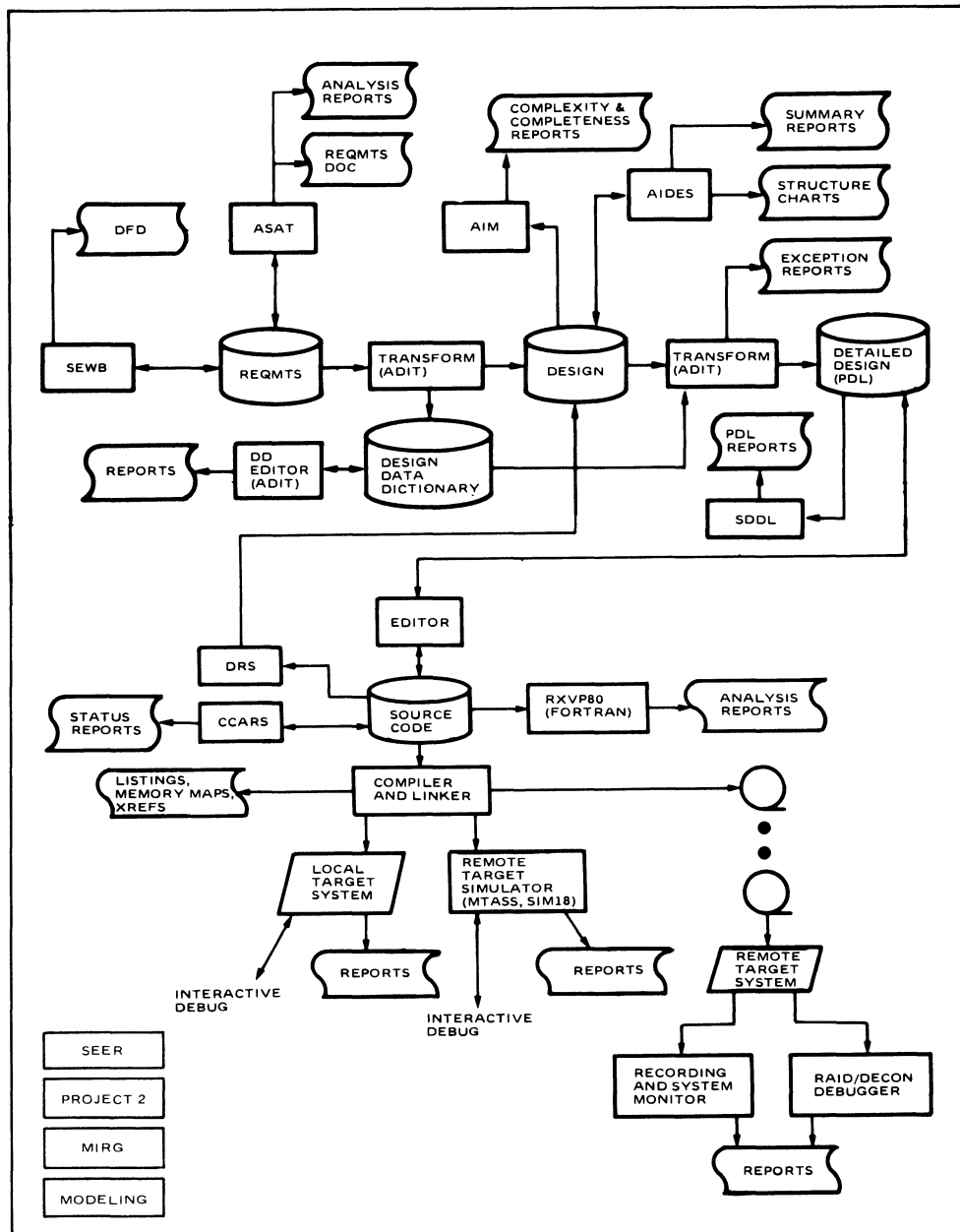


Figure 5: Dynamic View of the Hughes SE². The block diagram illustrates the flow of data and control among environment components which support the development life cycle.

From 1977 to 1984 software engineering exploded, interactive software development began, and software engineering methods were developed and refined. In 1977 there were 70 interactive users, 14 terminals, and 3 computers in the software engineering environment. This expanded to 1500 users, 665 terminals, and 16 computers by 1984. The DEC VAX 11/780 minicomputer and the DEC VT100 terminal became the standard hardware because of price/performance, the number of interactive users that could be supported, and the availability of the PWB. In addition, offices were wired to a communications switch so that terminals could be located more conveniently.

DECNET was installed to provide communications among the VAXs, and a disk farm was installed to reduce the effective cost of disk space. Also, the Automated Interactive Design and Evaluation System (AIDES) tool was developed to support the Structured Design Methodology. The System Engineer's Workbench (SEWB) tool was developed to support the Structured Analysis Methodology. However, the SEWB stressed the VAX/VT100 capabilities and precipitated the step toward workstations in 1984.

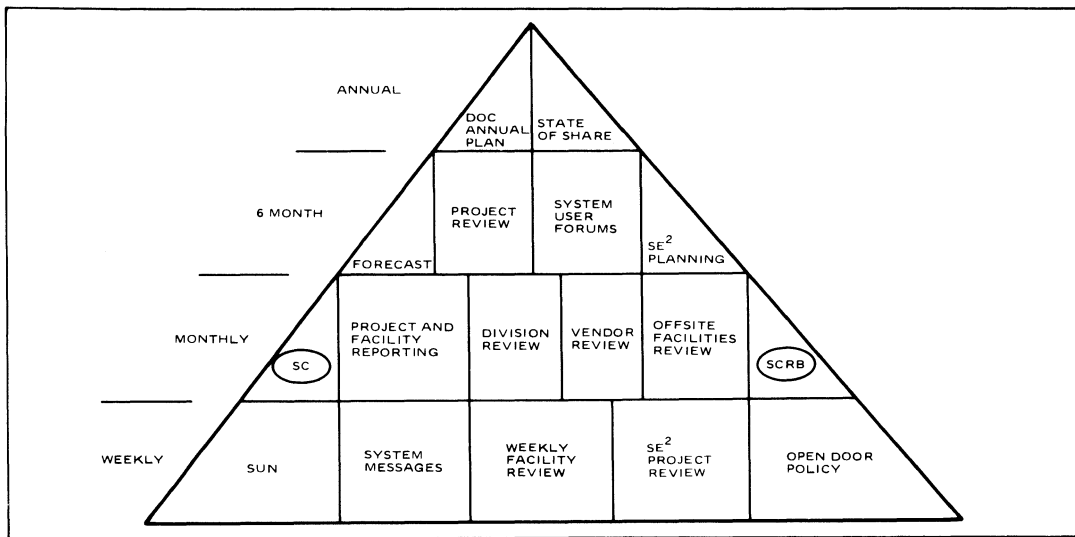


Figure 6: The Hughes SE² Supporting Infra-structure. Layered reviews and constant communication ensure that the required support services are efficiently and effectively provided for project personnel.

The next generation Hughes SE² was defined by the SDS Architecture Committee Report of August 1984. Hughes is currently well into the deployment plan of 450 workstations and a 50 percent productivity improvement (January 1983 basis) by December 1987. While the mission of the pre-1984 software environment was to provide a software development environment for real-time systems, the current mission is complete support of the software engineer. Implementation of the advanced environment began in 1984 with the purchase of four SUN workstations. The SUN workstation was selected because of its processing power, display capabilities, open architecture, and adherence to popular standards. Adherence to popular standards was most important, for example, UNIX and standard Fortran 77 improves tool portability, and ethernet, with the popular TCP/IP, provides the ability to evolve smoothly. Ethernet also provides the high speed communications in the workstation network and between workstation networks, process engines, central VAX system, target systems, and the Hughes corporate network. Therefore, the Hughes SE² is evolving toward an open architecture, virtual network which connects the workstation user to needed resources.

The current, large Hughes SE² is spread over 16 VAX computers and 90 workstations, which support 1700 software engineers. The dynamic view shows that the environment supports the entire software development life cycle using standard tools such as editors, formatters, and compilers, as well as specialized software development tools. In addition to tools which support each phase of software development, the Analysis and Design Interface Transforms (ADIT) tool integrates the requirements, preliminary design, and detailed design phases by providing the ability to automatically transfer engineering data from one phase to the next. The Change Control and Reporting System (CCARS) tool is used to control the source code by providing rigid control of the system elements, providing version control, and generating status reports. Simulators and target system tools are used to debug the system software.

A supporting infra-structure of training, user support, and informed decision making ties together the hardware, software, and methodologies to allow the environment to evolve in an orderly manner to meet the demands of the software engineer. A set of training classes is provided which covers the entire range of capabilities available. Classes are continually being reorganized to make sure that information is transferred to those in need. New forms of information dissemination are constantly being evaluated to ensure that the user is informed in a timely fashion. The user's first line of support comes from the user consultants. Often the user can obtain the answer to questions/problems directly over the phone. The informed decision making process has been constructed of several components with differing frequencies of checks, balances, information dissemination, and management involvement to achieve orderly growth.

The Hughes SE² was a creation of necessity resulting from the ever-increasing complexity of the software required for large scale military systems. Developing a competitive, state-of-the-art environment has not been easy. The fact that the environment must satisfy a variety of military customers has caused the environment's capabilities to be stretched. However, it is the day-to-day use on many projects, the constant assessment of galloping technology, and the mundane solving of user problems which has made the Hughes SE² a fact of life for 1700 users.

DESCRIPTION OF A PORTABLE COMPILER

David M. Barnes
Oregon Software, Inc.
Portland, Oregon

ABSTRACT

This paper describes the development and structure of Oregon Software's Pascal-2 compiler. Originally developed for DEC's PDP-11, Pascal-2 now runs on a variety of hosts under several different operating systems. The compiler is written entirely in Pascal, with machine dependencies and operating system dependencies isolated through parameterization, source control, and creative use of directory structures. Most of the source code is shared among the various implementations, so the compiler is relatively portable and enhancements are quickly propagated.

HISTORY OF PASCAL-2

Oregon Software's first compiler was Pascal-1, at one time the most popular Pascal available for Digital's PDP-11. It was a one-pass compiler written entirely in MACRO-11, and was known for quick compilation time and reliable generated code. Language extensions, such as an ORIGIN declaration for assigning variables to certain addresses, provided access to the hardware at the system level. Pascal-1 found its widest market in academia, but it also is also used in process control industry applications and the like.

Despite the almost immediate success of Pascal-1, it soon became clear to the developers that there was room for improvement. Pascal-1 was written in Macro-11, and so could not be transported to machines other than the PDP-11. Why not write the new Pascal compiler in Pascal, so it would be, to a large extent, portable across various machines? The developers also realized that they could improve the code optimizations of the compiler, but to do so using the existing Pascal-1 sources would be awkward, since Pascal-1 was implemented as a single-pass compiler. Another improvement they hoped to incorporate was improved error reporting.

The first commercial implementation of Pascal-2, the successor to Pascal-1, became available in 1980. Pascal-2 was written in Pascal, and surpassed its ancestor in code optimization and error reporting. It was also portable, as evidenced by the fact that the first implementation ran on the Honeywell Level-6 machine. The PDP-11 release was not far behind, and Digital bought a copy of the sources, from which it developed MicroPower Pascal and RSX Pascal.

Pascal-2 is now well-established as the only Pascal compiler that runs on all Digital machines, from the 11/23 to the VAX 8600. It is also available as a cross-compiler and a native compiler for non-DEC hosts, such as the Motorola 68000 series, and runs under various operating systems, including VMS and ULTRIX.

STRUCTURE OF PASCAL-2

Pascal-2 is a five-pass compiler. This does not mean that the compiler reads the source code five times, but simply that the compiler does its work in five distinct phases.

There are two reasons for implementing the compiler in multiple passes. One is to accommodate the compiler on the PDP-11, whose memory constraints restrict the amount of code that can be loaded into memory at any one time. On the PDP-11, the five passes correspond to five overlays, invoked in sequence. Another reason for multiple passes is the desire to modularize the code with an eye toward isolating machine-dependencies as much as possible. With machine dependencies isolated, porting the compiler to another host requires modifying a few modules rather than an entire program.

The five passes into which the Pascal-2 compiler is divided are:

Pass	Purpose
SCAN	lexical analysis
ANALYS	program parsing, semantic checking
TRAVRS	optimizations
CODE	code generation
LIST	listing and error reporting

Each pass does its work and leaves behind an output file, which serves as input to the next pass. Auxiliary files are also created during compilation, such as the string file, which holds string information. On virtual memory hosts, such as the VAX, the auxiliary files reside in virtual memory, rather than on disk. Figure 1 shows the relationships between compiler passes and their intermediate files.

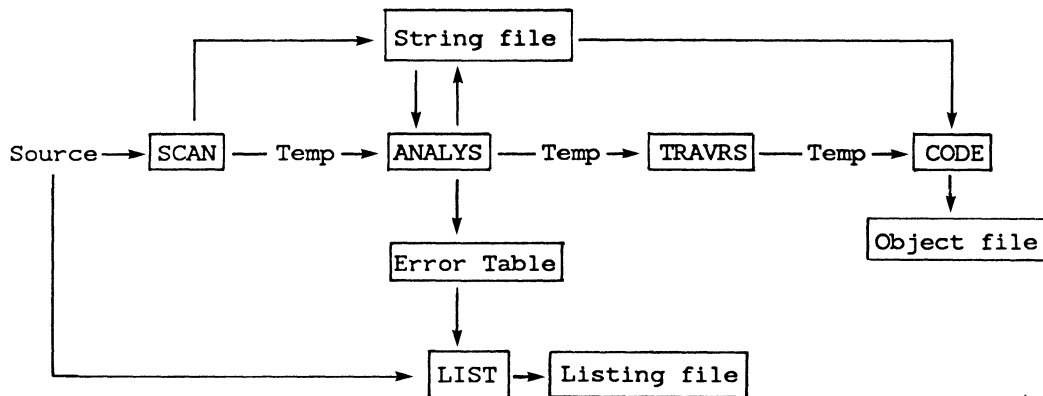


Figure 1: Compiler Structure

Of the five phases, only CODE, the code generator, and a portion of SCAN containing the command string interpreter, are heavily machine dependent. The other sections of the compiler's source code are essentially independent of machine architecture. Occasional operating system dependencies in the remainder of the code are accounted for by means of conditional compilation, which we'll describe later.

First, let's take a tour through the five phases of the compiler.

The SCAN Phase

The SCAN phase reads the source program and separates the source text into lexemes; its output is a stream of tokens (Figure 2). The tokens are fixed-length, encoded forms of the source text. Identifiers and string constants are represented by fixed length tokens that point to the identifiers or strings, which are stored separately in the string file.

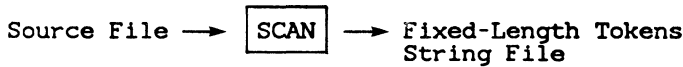


Figure 2: The SCAN Phase

The ANALYS Phase

The workhorse of the compiler's front end is the ANALYS phase (Figure 3). In an "average" compilation (if there is such a thing), about the same amount of time is spent in ANALYS as in code generation - each of these passes consumes about 35 percent, each.

The ANALYS phase reads the token stream from SCAN and parses it using a top-down, recursive-descent parser. The recursive-descent parser occupies less space in memory than other parser designs, an important consideration on some machines, particularly the PDP-11. The parser recognizes correct language constructs and encodes declarations in a symbol table. Incorrect constructs are diagnosed and noted in an error table.

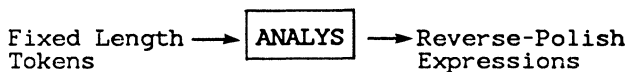


Figure 3: The ANALYS Phase

If ANALYS encounters fatal errors, it passes control directly to LIST, the fifth phase of the compiler, which handles error reporting. This short cut saves time for the user by eliminating the unnecessary code generation step.

Output from ANALYS is a stream of operators and operands in reverse Polish notation, the simplest form of output from a recursive descent parser.

Pascal code:

```

if A + B > B then
  A := A + B
else
  B := A + B;
  
```

Generated code:

```

      movl    B^12(R11),R12      ; get A
      movl    B^8(R11),R10      ; get B
      addl3   (R10),(R12),R9    ; store A + B in R9
      cmpl   R9,(R10)          ; comparison
      bleq   L3                ; branch
      movl   R9,(R12)          ; A := A + B
      brb   L4                 ; branch
L3.:   movl   R9,(R10)          ; B := A + B
  
```

Figure 5: Common Subexpression Elimination

The TRAVRS Phase

TRAVRS transforms the reverse-Polish expressions produced by the ANALYS phase into pseudocode used by the code generator (Figure 4). More than any other phase, TRAVRS follows Pascal formalisms, so it's very reliable from a maintenance standpoint.



Figure 4: The TRAVRS Phase

The compiler only spends about 10 percent of its time in TRAVRS (as compared to about 30 percent for code generation). Nonetheless, its function is quite important, as the output from TRAVRS marks the interface between the machine-independent front end of the compiler, and its machine-dependent back end. Pseudocode output from TRAVRS is a stream of labelled triples that all of our code generators understand. This powerful but concise pseudocode vocabulary makes it possible for us to mix and match front and back ends to produce a variety of native compilers and cross-compilers.

TRAVRS converts its reverse-Polish input stream into pseudocode by building an expression tree from the input stream, one procedure at a time, and then parsing that tree to produce the pseudocode output. As it creates the expression tree, TRAVRS manipulates the tree structure in such a way as to optimize the pseudocode it generates.

Optimizations

Here, let us take a moment and examine in some detail a few of the optimizations performed by the compiler. Many of the most interesting optimizations take place here in the TRAVRS phase.

Common Subexpression Elimination - One very effective optimization performed by TRAVRS is the re-use of repeated expressions. This technique is known as "common subexpression elimination."

In the code fragment shown in Figure 5, for example, the expression "A + B" appears three times. Common subexpression elimination recognizes that it is necessary to calculate the value of the expression only once. The value can be referenced rather than recalculated. This optimization saves not only the execution time required to recalculate the value, but reduces the register usage demands of the procedure in which the common subexpression occurs.

Pointers can be treated as common subexpressions, but only with great care. Because of the difficulty in detecting when the value pointed to has changed, the solution adopted is a conservative one: when comparing two pointers, TRAVRS checks to see if the types pointed to are the same. If so, it assumes that the value may have been modified and regenerates the value.

Constant Folding - Constant folding is another optimization, which is actually performed in the ANALYS phase. Constant folding involves calculating simple constant expressions at compile time. In the example in Figure 6, the expression "MaxLength - 1" is assigned at compile time a value of nine.

Pascal code:

```
const
  MaxLength = 10;

var
  a: packed array [1..MaxLength] of 0..10000;
  ...
  for i := 0 to MaxLength - 1 do
    a[i + 1] := i;
```

Generated code:

```
L2.:   clr1    R12
      movw   R12,L^Var[R12]
      aobleq S^#9,R12,L2.
      ; MaxLength - 1 => 9
```

Figure 6: Constant Folding

Dead Code Elimination - An optimization on which we rely heavily in our own development work is "dead code elimination." Dead code is code that will never be executed.

```
while false do
  A := A + 1;
```

Figure 7: Dead Code Elimination

In the simple example of Figure 7, the statement "A := A + 1" is never executed. Similar cases that are not as obvious occur in real programs, often unintentionally. Dead code elimination deletes such passages during the TRAVRS phase, so no unnecessary output code is generated.

We use dead code elimination at Oregon Software to implement a form of conditional compilation. For example, suppose a module that is shared by all implementations of the compiler contains a small machine dependency. It may not be necessary to rewrite the code to isolate the dependency in another module. We can create an enumerated type and use it as the basis for a CASE statement, where each case element specifies a processor-specific action (Figure 8). By editing the source file to set the value of the constant HOSTOPSYS, we can rely on the compiler to delete code not related to that operating system. The resulting software contains only the code which is used.

```
type
  OPSYS = (VMS, ULTRIX, RSX, MSDOS ...);

const
  HOSTOPSYS = VMS;

case HOSTOPSYS of
  VMS, ULTRIX : processor := VAX;
  RSX         : processor := PDP11;
  MSDOS      : processor := IAPX86;
  .
  .
  .
```

Figure 8: Conditional Compilation

Short Circuit Evaluation Of Booleans - Sometimes the value of a boolean expression can be ascertained without evaluating all of its terms.

Unless the value of one of the later terms is needed for subsequent reference, it's more elegant, and less costly in terms of instructions, to use "short-circuit" evaluation. The Pascal-2 compiler generates code that performs short-circuit evaluation where it's safe to do so, that is, in any situation in which the value of the skipped expression is not needed later by the program.

The example in Figure 9, while unlikely, illustrates short-circuit evaluation. The compiler determines that all of the constant terms are true, and that the result of the test depends solely on the value of the variable B. Thus, in the generated code, only B is tested.

Pascal code:

```
if b and (5 > 3) and ((2 = 7)
  or (7 div 3 = 2)) then
  b := false;
```

Generated code:

```
      movb   S^#1,R12    ; set true
      beql  L3.         ; test b
      clrb  R12         ; ..set false
L3.:   ...
```

Figure 9: Short-Circuit Evaluation of Booleans

Loop Invariant Removal - Loop invariant removal is an optimization in which unchanged expressions occurring within the body of a loop are evaluated once outside the loop and referenced from within the loop, instead of being recalculated on every pass. In Figure 10, for example, the expressions "k * j" and "k + j" remain unchanged throughout execution of the loop. The compiler generates code that calculates these expressions only once and stores them in registers R8 and R7, respectively, for easy access.

Pascal code:

```
for i := 1 to 10 do
  a[i + k * j] := k + j;
```

Generated code:

```
      mull3   R10,R9,R8           ; k * j stored in R8
      addl3   R10,R9,R7           ; k + j stored in R7
      movl    S^#1,R12            ; i := 1

; Loop is 3 instructions, beginning at L2.

L2.:   addl3   R8,R12,R6           ; i + k * j stored in R6
      movl    R7,L^Var+8[R6]       ; R7 assigned to array
      aobleq  S^#10,R12,L2.        ; (loop index)
```

Figure 10: Loop Invariant Removal

Expression Targeting - Expression targeting is a register optimization that takes into account the ultimate destination of an expression to guide register allocation during the expression's evaluation, thus avoiding register usage conflicts. In Figure 11, for example, if the compiler recognizes that it is possible to put the value of the computation $A + 1$ directly into the target location A, it will generate one instruction to increment A in place, thus overwriting the old value with the new one.

Pascal code:

```
A := A + 1
```

Generated code:
(without expression targeting)

```
movl A, RO
incl RO
movl RO, A
```

(with targeting)

```
incl RO
```

Figure 11: Expression Targeting

In those cases where changing the value of A directly might interfere with evaluation of the expression, the compiler must generate an intermediate value to avoid prematurely overwriting the target.

The CODE Phase

The CODE phase reads the pseudocode produced by TRAVRS (Figure 12). CODE maintains a record of the object machine state, and as each pseudo-op is read, it generates the minimum number of machine instructions to achieve the desired state change. CODE also includes a peephole optimizer that performs simple optimizations, such as branch/jump resolution and assigning constants or frequently-used addresses to unused registers for faster access. CODE then writes the generated code directly to an object file.

```
Pseudo-Code → [CODE] → Object Code
```

Figure 12: The CODE Phase

The LIST Phase

LIST is the final phase of the compiler. It performs error reporting, if necessary, and generates a source listing with embedded diagnostics from the error table (Figure 13).

```
Source File → [LIST] → Listing File
Error Table
```

Figure 13: The LIST Phase

COMMON FRONT END

Pascal-2 was well-received, and we soon offered the product on the entire DEC line, as well as systems based on the Motorola-68000, Intel 80186, and National Semiconductor 32000 processors. and as stand-alone systems. The rapidity with which these applications were developed led to multiple copies of the Pascal-2 sources, each of which differed ever-so-slightly from the others. Maintenance became a problem. Figure 14 shows the distribution of compiler phases with respect to machine dependence or independence.

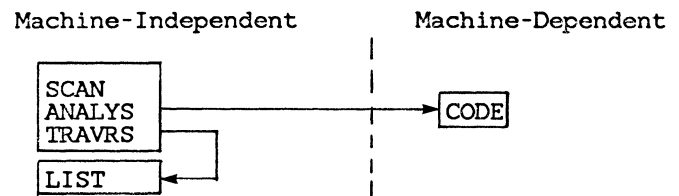


Figure 14: Machine Dependence/Independence

We undertook to re-integrate similar portions of the source code into one set of sources, in which conditional compilation was used to segregate the slight differences between versions. The result of this effort, known as the "Common Front-End" was completed in 1985.

The Common Front End has the advantage of propagating improvements to the software throughout subsequent versions, regardless of the host for which the improvements were made. Of course, there is always the risk that a bug introduced in the Common Front End will find its way into other versions of the compiler. But, through judicious testing and the use of source control, we are able to easily restore earlier versions of the code and weed out introduced errors.

What is involved in a typical compiler port? The time involved to create a new compiler is however long it takes to write a new code generator and command string interpreter. A stranger who had never seen the compiler before ported it to the 80186, under UNIX in six months. The ULTRIX compiler took about four months.

USING VMS DIRECTORY STRUCTURES

Development of the Common Front End was quite a boost to our maintenance and enhancement efforts. Separating the machine-independent modules from the machine-dependent ones is not always as simple as it sounds, however. Some modules, for example, are used for compilers that run on the VAX, regardless of operating system (VMS, ULTRIX, or UNIX). Other modules contain code for a certain operating system, such as UNIX, regardless of the type of the host machine. So, while most components of the compiler fall clearly into one classification or the other, machine-independent or machine-dependent, others lie in the gray area in between.

We do most of our development work under VMS. Our solution to this problem was to organize the directory structure of the Common Front End files to allow us to build any compiler in any configuration from a related set of directories. We rely heavily on the MAKE component of SourceTools to track the locations of various components in the VMS directory structure.

The organization of the files is a tree, where the root of the tree contains the machine-independent modules. Proceeding from the root to the branches, the files become increasingly version-specific. Files located in the terminal nodes of the directory tree are completely version-specific. The MAKE files that generate each version of the compiler are located in terminal nodes, for example. In general, the level below the machine-independent modules is processor specific, and the level below that one is operating system specific. Certain files are specific to an operating system, but not different between various processors. These modules are stored in a processor branch, called "ALL." Some modules can be integrated into a single module by means of dead code elimination. These modules are also stored in the ALL node.

There may be functionally equivalent modules lower in the tree. If such modules exist, they apply; otherwise the generic module applies. For cross-compilers in the structure, a distinction must be made between host and target, both for processor and operating system. Target processor and operating system are higher in the tree, followed by branches for host processor and host operating system. Figure 15 shows a portion of the Common Front End directory tree.

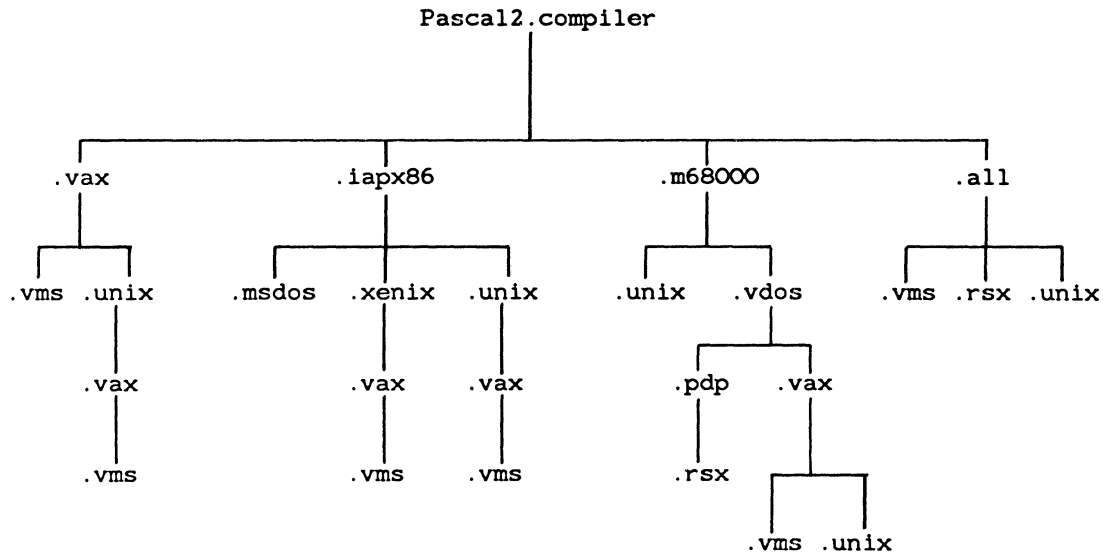


Figure 15: Common Front End Directory Structure

Distribution of the files over the various nodes of this structure is not permanently fixed. The goal is to move as many files as possible high in the tree, so that only one (or a few) copies of a module exist. This approach minimizes our maintenance effort and improves the quality of the product.

One way of pushing a module upward in the tree is by splitting it up into small machine-dependent parts and a larger machine-independent module. The machine-independent module "floats up" to a higher level in the tree. Another way is to parameterize code, using dead code elimination to handle all cases, thus making the module more generalized.

SOURCE CONTROL

A key component of the Common Front-End is a source control system developed in-house, known as SourceTools. The source control system allows us to do development work on the common set of sources without stepping on each other's toes, and maintains an audit trail of modifications to the sources.

SourceTools maintains centralized control over the source code for each module in the compiler and maintains an audit history. We can easily review the changes made to a given module: what the change was, who made it, and when. We can also recreate any version of the compiler when necessary to back out a misguided bug fix or investigate a customer inquiry.

A typical source control record of a compiler source module looks like the one in Figure 16.

Sourcecon Module: ANALYS

Owner: [2,26] File: ANALYS.PAS Module name: ANALYS

descr: Pascal-2 module

PROCESSOR: ALL

SYSTEM: ALL

50: 2.1F.2 made on 8-Nov-1985 at 04:23:48 by JANR Next delta: 49

purpose:

Added an errorcheck to handle the case where there are more array elements than 'maxaddr', the maximum we can index.

49: 2.1F.1 made on 7-Nov-1985 at 07:35:03 by DON Next delta: 48

purpose: Update to revision 2.1F

48: 2.1E.23 made on 23-Oct-1985 at 12:15:40 by JANR Next delta: 47

purpose: Initialized the variable "nullboundindex".

47: 2.1E.22 made on 23-Oct-1985 at 01:21:33 by RICK Next delta: 46

purpose: Added initialization of the boolean "standardfilesreferenced".

...

Figure 16: Source Control Record

Another nice feature of SourceTools is automatic keyword substitution. We use it in a number of ways: to insert revision information into the source code, for example.

Last modified by: ~name~ on ~update~

Last modified by: JANR on 8-Nov-1985

Figure 17: Keyword Substitution

COMPILER CERTIFICATION

Our efforts to centralize our compiler sources paid off in July of 1985 when Oregon Software became the first American vendor to certify not just one Pascal compiler, but eleven Pascal compilers. A twelfth was certified shortly afterward. All twelve certified compilers (Figure 18) were generated using the common front-end technology.

VAX/VMS native
VAX/UNIX (BSD 4.2) native
VAX/ULTRIX-32 native
SCI-1000 80186/UNIX (IN/ix) native
VAX/VMS to 80186/UNIX (IN/ix) cross
VAX/UNIX (BSD 4.2) to 80186/UNIX (IN/ix) cross
VAX/ULTRIX-32 to 80186/UNIX (IN/ix) cross
VAX/VMS to 68000/UNIX (RTU) cross
VAX/UNIX (BSD 4.2) to 68000/UNIX (RTU) cross
VAX/ULTRIX-32 to 68000/UNIX (RTU) cross
Masscomp MC500 (68000)/UNIX (RTU) native
SUN Workstation II (68000)/UNIX native

Figure 18: Certified Compilers

MODULA-2

Our latest challenge is a different sort of compiler port - a Modula-2 compiler whose development begins with Pascal-2 code generators and optimization logic.

To parse a different syntax, changes to the front-end are required. For a language such as Modula-2, whose syntax is very similar to that of Pascal, the changes should be minimal. Existing code generators will change somewhat, too, as the front-end redesign introduces additional pseudo-ops.

CONCLUSION

Our experience in developing and porting the Pascal-2 compiler verifies that a systematic, modular approach to software development yields the expected benefits. In our case, we developed a compiler entirely in a high-level language, and the resulting compiler is, indeed, highly portable and efficiently maintained. Source control tools and methodical isolation of machine dependencies via modularity and carefully designed directory structures also improve the transportability of our compiler.

ACKNOWLEDGEMENTS

Pascal-2 and SourceTools are trademarks of Oregon Software, Inc.

VAX, VMS, PDP-11, and ULTRIX are trademarks of Digital Equipment Corporation.

M68000 is a trademark of Motorola, Inc.

UNIX is a trademark of AT & T Bell Laboratories, Incorporated.

AN APPROACH TO BUILDING LARGE,
MODULAR SOFTWARE PROGRAMS

F.E. Cross
Naval Weapons Center
China Lake, California

ABSTRACT

The paper suggests an approach for defining and building large, but modular, and configurable software programs. The areas addressed by this approach are goals, requirements, concepts, tools, and environment. Included are some architectural features such as control of program flow and access to (automated) debug data. In addition, module standardization, ground rules for the software environment, and an effective, yet lenient, configuration control system are included.

This approach is best suited to those programmers who want to follow the software engineering principles of modularity and hierarchical decomposition, but who must also integrate their work with others' to incrementally build large programs. This approach was applied on a VAX/VMS system to build a Fortran avionic simulation of more than 2000 subroutines.

INTRODUCTION

Rarely does one find software programs that are designed to grow, or are even configurable enough to permit growth. Instead, growth or modification is a painful procedure requiring "patches" to baseline modules (and resulting configuration control of those patches). Patches corrupt small, singular-concept modules by making them into larger, multiple-concept ones. At best, some growth in capabilities occurs. At worst, some baseline capabilities are lost. In any case, the documentation (if it ever existed or was of sufficient quality) is outdated by the change. After a few layers of patching, the program becomes very fragile to any change. The net result is that discouraged programmers move on to new projects where a "fresh start" means getting out of the "patch-it-and-see-how-it-works" mode.

There is no reason why programs cannot be designed to grow and even to accommodate multiple functions if one discards old habits that are limitations to cleaner, more modular programs. Old habits are those things that programmers do because they've always done them, and do not really trust tools or a new approach. Some examples are manual coding of module's data declarations (In FORTRAN, these are COMMONs, type, and DIMENSION statements.) and manual insertion of debug or diagnostic code. Being freed of these manual tasks would allow programmers to be more creative, therefore, more productive.

This paper presents some lofty goals for software development, but a reasonable approach for attaining them. The foundation for this approach are the three keys to successful programming architectures: modularity, standardization, and genericness. These are concepts that programmers should look for in all software architectures. This approach will be described under five general areas: goals, requirements, concepts, tools, and environment that are related as follows:

1. Goals provide purpose and perspective,
2. Requirements are those necessary to meet the goals,
3. Concepts represent the approach taken to satisfy the requirements,
4. Tools are those needed to support the concepts, and
5. Software environment prescribes the use of the tools to accomplish the goals.

Figure 1 shows these five general areas along with the subtopics presented under each.

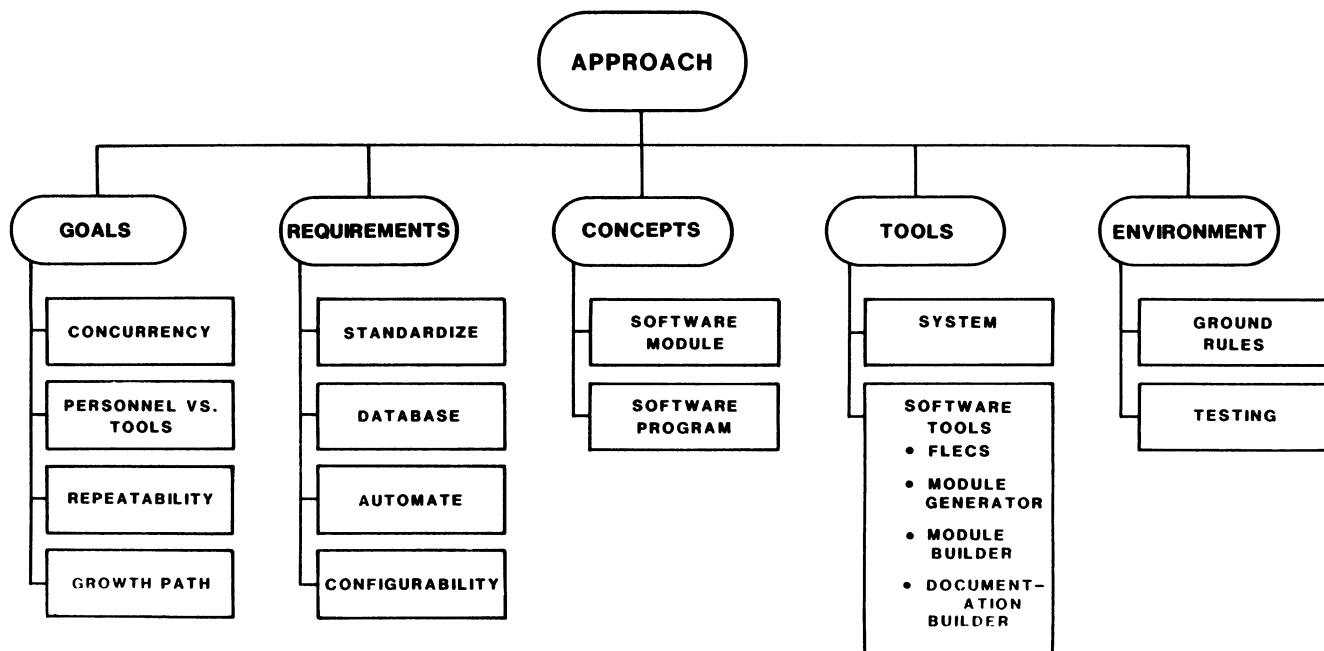


Figure 1. Approach Tree.

GOALS

Before any software project is started, there should be some clearly stated goals. The goals provide technical people with the best possible "big picture." They should include some indications of the project's purpose, scope, and rationale. While this information may seem extraneous (especially to management), it provides programmers with an understanding of their function in the project, and is probably the best motivating factor for them. The following goals are a recommended subset of any project's goals. The qualifications that a project adopt these goals are that it is "big enough" to require more than one programmer, that it last more than one year, and that it experience at least one change in user requirements sometime in its lifetime. The goals are:

1. Support concurrent development and use,
2. Consider personnel versus tools,
3. Ensure test repeatability, and
4. Provide a growth path.

Support Concurrent Development and Use

Software is often in use and under development at the same time. This occurs because the user usually cannot wait for the finished product and is willing to use a preliminary version. For this reason, the user tasks and development tasks must be accomplished concurrently, but not at each other's expense. The software design should take this into account in the early project phases, and should anticipate that

it will continue throughout the later phases.

Consider Personnel Versus Tools

The software development approach should consider automation in the software development process. The computer-aiding tools considered should address reduction of manpower requirements. Neither the software development environment nor the program developed for the project should require much repetition of mundane activity by the programmers. Experience has shown that such repetition leads to boredom that results in errors. An environment that fosters errors often has a high personnel turnover; therefore, the software development approach should allow the programmers time to automate less creative activities.

Ensure Test Repeatability

Any integration test must be run in such a manner that the results can be duplicated later. This appears to be self-evident, but many engineers have conducted tests and reported results that could not be reproduced. This usually happens in situations where there are too many variables over which the testers have too little control (such as hardware changes). The purpose of this goal is to create an awareness of the problem so that these situations can be avoided.

Provide a Growth Path

The approach taken to accomplish the early phases of the project should allow a natural growth into later phases with little or no redevelopment of previous

capabilities. In other words, developers should design and build a flexible program so that code will not have to be thrown away later when user requirements change.

REQUIREMENTS

Having set the goals above, let's identify some general software requirements to try to meet those goals. Some requirements that should be incorporated into any large software project are:

1. Standardize the software development procedures,
2. Data base the information,
3. Automate the procedures, and
4. Create a configurable set of software that can grow and adapt easily to changing requirements.

Standardize

Of all the software development requirements, the degree of standardization is the factor that probably most influences the coding activity. Without standardization, configurability is difficult, but automation and data basing are nearly impossible. More on those later. The topics to be covered under standardization are;

1. Review the benefits of standardization,
2. Formalize the approach to implementing software,
3. Adopt a single format for modules (coded routines), and
4. Standardize documentation forms.

Benefits of Standardization

Programmers sometimes resist standardization of the software development process. This resistance usually increases with the programmer's age and/or past volume of written code. If programmers are to accept standards, this resistance must be outweighed by the benefits. The benefits of standardization are improved communication, increased control over the code, and decreased time needed to turn ideas into code.

Communication is improved by standardization because programmers will "speak the same language" when reviewing each other's modules. This occurs because the concepts are structured in the same format and are easily recognized.

Standardization can increase the programmers' control over the code. This can be both a dynamic or static control. Dynamic control, which occurs while the program executes, is attainable because access and interfaces to the code can also be standardized. We will see later how standardizing the access (subroutine calls) can simplify configuration management. Static control, which is control over the source code, is attainable through automated procedures because the information stored in each software module is found in the same format.

Standardization decreases the time and effort required to turn ideas into code. This can happen because many of the manual tasks (such as insertion of module debug code, testing, and generation of documentation sets) can be automated.

Formalize Approach to Implementing Software

Most organizations that generate software like to think that they have an efficient, formal approach to coding. In reality, few do. Those that have formalized, have benefited from it because, in the process, they also standardized their approach. To formalize one's approach is to define a target for others to criticize; so be open to feedback. It is better to formalize one's approach to developing software, offer it for criticism, and use the feedback to improve the approach, than to never address it at all.

Single Format for Modules

Most programmers will agree that it is best to have a single format for all software modules. Each, of course, thinks that his/her format is best. Some programmers will compromise more than others on the module's format; but, in the end, there should be only one who decides. This one person should address both coding and documentation formats for the module.

Coding standards and formats are criticized loudest by those programmers who most need them. Therefore, the best coding standards are those that standardize the format of coded concepts or ideas and leave the implementation details to the programmer's style. This creates an environment where the walkthroughs are high enough in level to be productive and not sessions of haggling over variable names or "look-how-you-can-code-this-in-3-lines-of-code-instead-of-4," and other such trivia.

The module format should also include a documentation format. Without a format, the documentation will (if it gets done at all) be even more stylized than the code it documents. So, if the documentation is

not in a recognizable standard format for other programmers, it and the code it documents will be meaningful only to its author.

Standard Documentation Forms

Every software project should have some standard documentation forms. These forms serve two purposes. They provide an overview of the project software and its progress and provide access to the various levels of software information. Some of the standard documentation forms are listed below.

1. Structure trees - show how one concept module breaks down into others.
2. Status reports - show the stage of development of each module.
3. Module listings - provide details to the implementation of the module.
4. Symbol dictionaries - list all the program's symbol names with their description and specifications.
5. Cross references - show which modules reference the symbols listed in the symbol dictionary.
6. Keyword searches - list the modules that contain specified keywords.
7. Test input/output files - provide records of the test performed and their results.

Data Base

The next general software requirement that every project should consider is that of data basing its information. A data base of the software work can require a standardized and dedicated approach, but it can also provide an accessible, permanent storehouse of the project's corporate knowledge. The data base should be a part of the software development system and not some separate, therefore neglected, activity. The two data base topics to be covered are: the software module as a unit of the data base and the data base being expandable and modifiable.

Software Module Units

One way to ensure that the data basing of software information does not become a neglected activity (and cease to represent the program as implemented) is to make the coding and the data basing the same activity. This can be assured by requiring that each software module of the

program be a unit of the data base. While this collection of software modules is not a formal data base, it does ensure that its data is current and accurate. It also provides a starting point for a later conversion to a formal data base.

Expandable and Modifiable Data Base

The data base of software modules should be expandable and modifiable. The number of software modules in a project can grow rapidly, and usually does when the early phases of the project are successful. Therefore, the method for accessing the data base modules should not be hindered by increasing the number of modules. The format of the modules will also be subject to change, so the data base should not be restricted, or introduce restrictions, to the module's format.

Automate

The third general software requirement that every project should consider is that of automating as much of its methodology as possible. The degree to which any activity can be automated is directly influenced by the availability and regularity of the data input. For software development, input data consists of the information needed to build the software modules (such as module inputs, outputs, and algorithms). If the modules are all standardized to one format, and that format identifies the module's inputs and outputs, then a great deal of automation can be achieved. Two areas where this automation can be accomplished are:

1. Generation of module-related functions of code, and
2. Updating of standard documentation forms.

Module-related Functions of Code

There are module-related functions whose code can be easily generated automatically (i.e., by tools). Some examples of these functions are the trace of program flow at the module level, the display of the values of the module's inputs and outputs, and timing of the execution of a module. A tool that would generate the code to perform the above functions would require only the list of the module's input and output symbol names and their data type specifications. That same information is required to build the module, and would be available in the module format.

Update Standard Documentation Forms

Some of the standard documentation forms, such as structure trees, status

report, symbol dictionary, cross reference, and keyword search, were described earlier under REQUIREMENTS. These documentation forms all make use of data needed to build modules so that data is available in the standard module's format. If the requirement to treat all the software modules collectively as a data base is satisfied, then all the data necessary to automate the update of the standard documentation forms is available.

Configurability

The last general software requirement that every project should include in its approach to software development is configurability. Configurability, without defining what it is yet, allows programmers to ignore the rest of the program while they concentrate their attention on their task but still remain an integrated part of the total program. Configurability does not just happen nor can it be easily "added on" later. Configurability must be addressed as a desired requirement and planned into the software development approach. To better understand this concept, consider:

1. Definition of configuration,
2. Uses for configurations, and
3. Benefits of configurability.

Definition of Configuration

There are ways to refer to individual modules. Usually this is done by referencing the module's name or the module's number. One can also refer to a collection of modules. Usually this is done by calling it "the program." However, there is no common way to refer to a subset of modules within a program. To do this, we have selected the term "configuration." For the purposes of this paper, the word "configuration" refers to a specific set of modules that are a subset of the entire program.

The approach set forth in this paper recommends that software modules be uniquely identified by number instead of the more traditional title or name. In practice, we have found that programmers accept this policy, if the number is always followed by the module's title (For example, AV1234 ! AIRFRAME MODEL where all module names begin with "AV" and the in-line comment delimiter "!" is used to precede the title). This lets one pre-name all the modules in a program (allowing for as much growth as desired), create skeletons of those modules, and fill them in as the program grows. Naming a module with a number permits the creation of the generic subroutine call because a number can be associated easily

with an address (whereas, names do not associate well). Identification by number also makes configuration modules possible. A configuration module is a software module whose sole purpose is to specify a set of numbers. That set of numbers identifies a group of modules whose uses are discussed below.

Uses for Configurations

How are configurations to be used? They will be used to activate a specific capability that is a subset of the total capability. Consider the following examples. Suppose that the call to every routine (software module) in the program is preceded by a test on a logical that is uniquely associated with the module, such as;

```
IF (ACTIVEMODULE(1234)) CALL AV1234
```

where ACTIVEMODULE is a logical array and AV1234 is a subroutine name. Then, a configuration can be used to activate the set of logicals (ACTIVEMODULE) and "switch on" the desired set of modules for execution. This provides effective control of program flow at the module level.

As another example, suppose that some modules serve only diagnostic or test purposes in the program. They can still be designed into and left in the program because they are always "switched off" or activated only when desired.

Benefits of Configurability

In general, configurability gives programmers and users a large degree of control and flexibility over their program and its operating environment.

Consider, for example, what the impact of configurability is on program versions. Since any module can be activated or deactivated on demand, there is never a need to "patch" a baselined module. A new module, by a different number, can be copied from the old one instead. Thus, no baseline code is ever thrown away or changed to meet new requirements. This means that conceptually, program versions are replaced by configurations that are activated by an input file to the program. There is only one linked version of the program and all "versions," "baselines," etc., exist as configurations within the program.

Consider, too, the working environment. Code that is baselined and code that is under development can coexist in the same executable program. The code under development will not impact the users of baselined code because those users will not activate any modules that are under development.

Configurability can eliminate the conflict among programmers that occurs in situations where one programmer changes another's code. Now, programmers can copy that code into a new module and change that, instead. With the support of configurability, all modules can be treated as the "private property" of each programmer. No one need worry that someone might change his/her module and, consequently, introduce problems.

The quality of code also increases with configurability. Tests and diagnostics are no longer lost in the development rush but remain as a part of the program, ready to be activated when demonstrations are desired.

Configurability, along with modularity, increases the portability of the program's code. While it is quite unlikely that any program is entirely portable from one system to another, the ability to deactivate modules that are not portable increases the likelihood that the remainder can be run on another system.

One last benefit of configurability is the flexibility it gives the program user to consider use of multi-processors for a program that originally ran on a single processor. This can be possible by running the same program on both processors (having shared data) and activating the set of modules in one processor that are deactivated in the other processor. Distributed processing is not this simple, of course; but there are situations where this type of processor loading is feasible, and configurability allows it to be considered.

There can be some disadvantages with configurability and modularity to consider as well. First, there are longer link times to handle many modules. Second, if a VMS system is not used, lots of memory will be needed to hold the "dead" code. Third, modules may tend to get too small (fragmentation) as developers seek greater control, or configurability, over their code.

CONCEPTS

Now that we have identified the requirements for this software development approach, let's look at some concepts that satisfy those requirements. These concepts are addressed at two levels: the software module level and the software program level.

Until now, the ideas presented have been general enough that they could apply to most languages and computer systems. From this point on, however, the details for these ideas become less general as they become more application oriented. As greater detail and specifications are required, the author has resorted to the FORTRAN language. This is done for con-

venience and is not meant as a language dependence or restriction. In addition, a program is referenced as an example that uses this software development approach. That program is the avionic simulation (AVSIM)[1], developed for the A4/AV8 project at the Naval Weapons Center in China Lake, California.

Software Module

The software module is the smallest configurable unit of a program. It addresses a single concept and completely implements that concept so that the module is self-contained. The software module topics are:

1. Types of software modules,
2. Module standard, and
3. Context of the module.

Types of Software Modules

There are basically four types of software modules as listed below. While each has a different function, all four have the same standard format.

1. Computation module - does number crunching.
2. Flow control module - performs an executive function in that it selects and calls the desired modules for execution.
3. Interface module - combination of the computation and flow control modules that feeds the program's data into some general purpose module, calls the module, and then saves the results of that module for use by the program.
4. Configuration module - specifies a set of module numbers as described in the configurability subsection of the REQUIREMENTS section.

Modules do not always fall neatly into one of the four types listed above. For instance, a module can perform both computation and flow control functions. In any case, a neat classification of each module is not necessary to this approach at this time.

Module Standard

The standard for the software module selected for this approach is basically that described in DoD MIL-STD-1679(Navy)[2]. This standard interprets the well-known structured programming concepts and produces a list of do's and don'ts. For example, it limits modules to

a single entry/single exit, 200 lines of code, and standard programming control structures. It prohibits the use of GOTOs and self-modifying code.

All inputs and outputs (I/O) for the standard module are passed via central dictionary COMMONs or argument lists. Since all modules' I/O symbol names appear in the central dictionary, and the module identifies that dictionary, all module I/O code generation can be automated. In most cases, then, programmers need not be concerned about coding a module's data declaration or module-related functions. In addition, programmers do not need to code module-related functions. These functions, such as trace, debug of outputs, and timing, are coded automatically by a tool, upon request.

The format of the module (see Figure 2) consists of:

1. I/O header section includes, as a minimum:
 - a. module's title and number;
 - b. brief description of the module's purpose;
 - c. list of its input, output, and local variables;
 - d. list of external modules;
 - e. indicators for generating module-related code;
 - f. name of the central dictionary; and
 - g. status indicator for the level of module completion.
2. Code section - contains the user's executable code.
3. Documentation trailer section - provides explanations for the code and follows the format in "STRUCTURED DOCUMENTATION"[3].

Context of the Module

Now that we have seen what an individual software module looks like, let's consider the context of that module in the total software development approach. First, each module is uniquely numbered and a total number of modules is decided. For the AVSIM example, the modules are named AV0000 to AV9999. These 10,000 modules are then created in individual, permanent files as skeletons or stubs to be filled in as needed. This large number was selected on the basis that the modules be small (and therefore numerous) and the AVSIM program design be expandable over a long lifetime.

Second, there is a calling convention for each module. That convention is to precede the call with a test on a logical that is uniquely associated with that module. Following this convention ensures configurability of the total program.

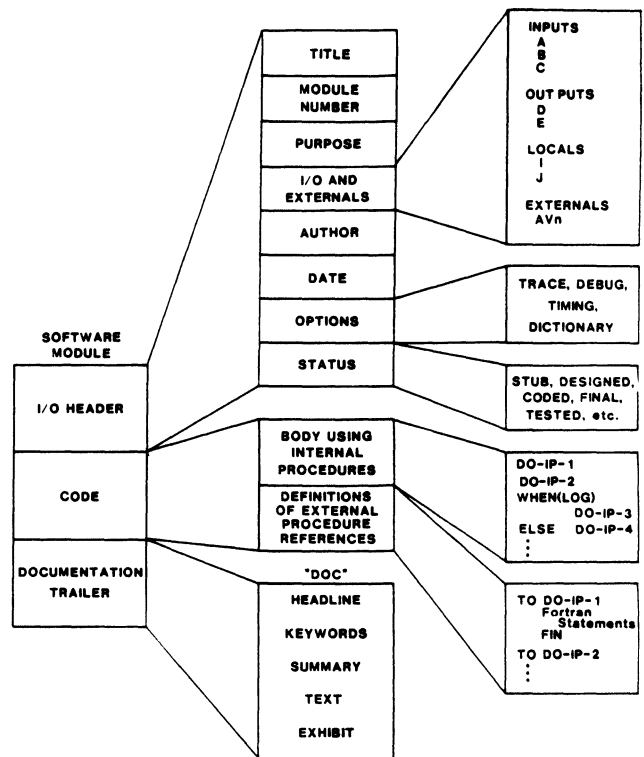


Figure 2. Software Module Expansion.

It is also useful to consider use of a generic subroutine-calling module. This module passes one argument, which is the number of the module to be called. For example, CALL AV_CALL (1234) produces the same result as CALL AV1234, where AV_CALL(n) is the generic subroutine-calling module.

Software Program

Having discussed the software module, we will now look at the recommended features of software programs. The topics to be covered are:

1. All modules are linked into one executable,
2. Program flow is controlled by the input file, and
3. The selective control of module debug functions.

All Modules are Linked into One Executable

All of the software modules should be linked into a large, single executable, producing a single version of the program. In this way, no modules are ever lost and anyone can access the modules developed by everyone else. Use of the object library lets the developers write a link command that need not be updated every time a new module is written.

Program Flow Controlled by the Input File

The program flow desired by user is specified by a program's input file which identifies only those modules the user wants activated for a run. By default, all modules are initially "switched off" until the user turns them "on." The program designers may, however, have some baseline or startup modules that are activated by a bootstrapping portion of the program. In any case, time-consuming compiles and links can be reduced by designing a configurable program where sections of code are activated via input files.

The input file not only activates the modules desired by the users, but also allows them to effectively replace one subroutine with another (without re-linking). This is accomplished through the use of the generic subroutine-calling module and its call list. In the example seen earlier, we showed that CALL AV_CALL(1234) was equivalent to CALL AV1234. Use of the call list, REF_LIST, allows the user to reassign the address of the module called. For example if an input file uses AV5678 instead of AV1234, the user needs only to put REF_LIST(1234)=5678 in the input file and AV5678 will be called every place in the program where there is a CALL AV_CALL(1234).

Selective Control of Module Debug Functions

The software program has some modules whose function is to control the section of module-related debug functions. These functions, such as trace, debug outputs, and timing, are selectable by function and by module or configuration number. In the cases where the program executes in numbered cycles, the module-related function(s) are also selectable for a beginning cycle number, a terminating cycle number, and a frequency rate (activated every second, third cycle, etc.).

TOOLS

Now that we have the above software development concepts in mind, let's identify some tools that can support them. Tools can be generally divided into two categories, system and software. For the purposes of this paper, the system tools are the computer-dependent support programs and the system that runs them. The software tools are those programs that are generally computer independent.

System Tools

The hardware and computer-dependent support software needed for a program developed under this approach are listed below. The VAX/VMS series computers satisfy these requirements.

1. Virtual memory system (VMS) - a VMS is necessary to free the programmers of memory constraints so that the program can grow as large as required.
2. Shared memory - shared memory provides communications between processors in systems having distributed processing.
3. Versatile linker program - the operating system must provide a linker that can handle large numbers of subroutines (modules) and large numbers of labeled COMMONs, since there may be as many labeled COMMONs as there are central dictionary symbols.
4. Permanent file directories - the operating system must provide directory structures for accessing large numbers of permanent files because each software module is stored in a separate permanent file.
5. High resolution clock - a high resolution clock is necessary for timing small modules that run on the order of milliseconds. System clocks usually do not have this resolution. DEC's KW11-P programmable clock satisfies this requirement.

Software Tools

The software tools listed in this section are those required to support the proposed software development approach. They are mostly custom-built programs; and so, all are not available in computer system libraries. They are, however, all in the public domain. These software tools are;

1. Fortran Language Extended Control Structures (FLECS)[4],
2. Module Generator[5],
3. Model Builder[6],
4. Standard documentation forms tool.

Fortran Language Extended Control Structures (FLECS)

Since the language selected for the AVSIM program was FORTRAN, a tool to extend the language's control structure was desired. That tool, FLECS, is a FORTRAN preprocessor that extends the FORTRAN 77 control structures and generates FORTRAN code. The two primary control structures (and not provided in ANSI 77) are the internal procedure and the CASE statements. The internal procedures appear as pseudocode. Pseudocode allows the programmer to write modules in English-like

sentences and show the details of implementation elsewhere in the module.

Module Generator

The Module Generator is used for design and coding of the programmer's software module. At the design stage, the Module Generator format helps the programmer identify the module's I/O data and external modules. Because the Module Generator accepts FLECS coding, it helps the programmer design a module's code through the use of internal procedures and control structures.

At the coding stage, the Module Generator generates two different kinds of code for the programmer. First, it uses the program's central symbol dictionary to generate the module's data declarations. These statements include the labeled COMMONs, type, and DIMENSION statements (FORTRAN). Second, it generates the code for the module-related functions, such as trace, debug outputs, and timing. These code generation functions free the programmer to concentrate on a module's function and not spend time handling the module's data declarations or debug code.

Model Builder

The Model Builder generates real-time code from specified software modules. It does this by combining the small modules called by an executive routine into a single, large subroutine. In the process, it leaves out all debug code and checks for conflicts in the use of local names. The Model Builder does not affect the source modules, but only creates objects that are linked into the main program.

Standard Documentation Forms Tool

The standard documentation forms tool generates and updates the various documentation forms. This documentation program uses the standard module source files as input, treating that dictionary of permanent files like a data base. The output of the documentation program are structure trees, status reports, symbol dictionary listings, cross references, keyword search results, and reference manuals.

The structure trees represent the conceptual decomposition of each module. They show the modules called from each module. The listing contains each module's number and title and indicates the level of the module in relationship to the other modules. The example in Figure 3 illustrates an AVSIM structure tree for AV1510, a module that controls the sampling of global point data.

The status report is a list of all the modules in the user's program. It

lists the module number, title, module-related functions implemented, and development status. The example in Figure 4 illustrates a portion of the AVSIM status report. Note that the unused modules, or stubs, are indicated by "0/0" in the TITLE column or "S" for STUB under the STATUS column.

The symbol dictionary list provides an alphabetical listing of the user's central dictionary. The format of a dictionary item is:

```
symbol_name=      textual      description
$shared_memory_indicator
  [ data_declarations ]label_common_name
```

where the \$shared_memory_indicator is optional (\$STATE indicates a state variable to be located in shared memory.), and the data_declarations consist of data type and dimension specifications. For examples of elements in the dictionary, see the cross reference sample that follows.

The cross reference shows which modules reference the program's I/O data. The cross reference is generated from the symbol dictionary listing and a file of the I/O headers from every module in the program. The example in Figure 5 illustrates a few items from the AVSIM cross reference. Note that the type of usage is indicated by *OUT for output, *INP for input, *LOC for local, and *REF for referenced (meaning that the variable may be either input, output, or both).

The keyword search lists all of the modules containing specified keywords. To generate the keyword search results, the program uses the keyword portions of the documentation trailer of all the software module source files. The result is a list of all the modules containing the specified keywords and a list of each module's keywords as well. The example in Figure 6 illustrates an AVSIM keyword search list for the keyword "EARTH." Note that for many modules the default list of keywords is simply the title.

The reference manuals are sets of "docs" grouped by subject matter, model, author, etc. The "doc" is the documentation trailer found at the end of each module (see Figure 2). The user specifies the keywords and/or other module criteria, and the tool searches the data base of documentation for the appropriate "docs" to compile into a reference manual (as an example, the list of modules produced into a reference manual). The list of modules produced by the keyword search in Figure 6 could also be used as a list of "docs" that would be compiled into an "earth model" reference manual.

```

AV1510 ! SAMPLE GLOBAL POINT DATA
      AV1520 ! SAMPLE EARTH DATA
            AV1525 ! I/O INTERFACE TO AV1526, GENERIC EARTH DATA
                  AV1526 ! GENERIC EARTH DATA

AV1530 ! SAMPLE OCEAN DATA
AV1540 ! SAMPLE ATMOSPHERE DATA
      AV1550 ! I/O A/C INTERFACE TO AV1555, SAMPLE WIND DATA
            AV1555 ! GENERIC WIND DATA
                  AV0600 ! FIRST ORDER LAG FUNCTION

AV1560 ! I/O INTERFACE TO AV1565, SAMPLE AIR DATA
      AV1565 ! GENERIC AIR DATA

```

Figure 3. AVSIM Structure Tree.

NAME	CODED	OPTIONS	TITLE	STATUS
	T D T	D		STATUS
	R E I	I		QUALITY
	A B M	C		CLASSES;
	C U E	T		S = STUB
	E G			F = FINAL
				D C Q T
				E O U E
				S D A S
				I E L T
				G D I E
				N T D
				E Y
				D
AV1519	X X X	X %		S
AV1520	X X X	X	SAMPLE EARTH DATA	X X F
AV1521	X X X	X %		S
AV1522	X X X	X %		S
AV1523	X X X	X %		S
AV1524	X X X	X %		S
AV1525	X X X	X	I/O INTERFACE TO AV1526 - GENERIC EARTH	X X X
AV1526	X X X	X	GENERIC EARTH DATA	X X F
AV1527	X X X	X %		S

Figure 4. AVSIM Routine Status List.

```

GRAVITY= STANDARD GRAVITATIONAL ACCELERATION OF AN OBJECT (FT/SEC2)
$STATE [ REAL (7) ]GR9248
      AV1525*OUT AV1725*INP
GRAVITY_G= STANDARD GRAVITATIONAL ACCELERATION (GENERIC) (FT/SEC2)
[ REAL ]GR3895
      AV1525*REF AV1526*OUT AV1725*REF AV1726*INP

RADIUS_GEOCEN+ GEOCENTRIC EARTH RADIUS FOR AN OBJECT (FT) $STATE
[ REAL*8 (7) ]RA4970
      AV0538*INP AV1525*OUT AV2200*INP AV2320*INP

RADIUS_GEOCEN_G= GEOCENTRIC EARTH RADIUS (GENERIC) (FT)
[ REAL*8 ]RA7002
      AV1525*REF AV1526*OUT

```

Figure 5. AVSIM System Cross Reference.

```

:
0459 ! EARTH MODEL GRAVITY INITIALIZATION
*KEYWORDS* AVSIM AV_INIT EARTH MODEL GRAVITY INITIALIZATION

1520 ! SAMPLE EARTH DATA
*KEYWORDS* AVSIM AV_CYCLE SAMPLE EARTH DATA

1525 ! I/O INTERFACE TO AV1526 - GENERIC EARTH DATA
*KEYWORDS* AVSIM AV_CYCLE I/O INTERFACE TO AV1526 - GENERIC EARTH

1526 ! GENERIC EARTH DATA
*KEYWORDS* AVSIM AV_CYCLE GENERIC EARTH DATA
:

```

Figure 6. AVSIM Keyword Search Results

ENVIRONMENT

Having identified the tools above, let's look at the environment in which to use them in this approach to software development. How a tool is used is almost as important as what the tool does. This section describes the proper environment for using these tools. The two areas of the environment discussed are the ground rules for coding and the testing philosophy.

Ground Rules

The ground rules represent agreements among program developers and users on day-to-day coding practices. For this approach, the agreements are:

1. Each module is edited by only one programmer. Each module is unique and "owned" by its writer. No one modifies anyone else's modules or creates multiple versions of the same numbered module.
2. Baseline modules are never changed. Once a module has passed its tests, it has satisfied some set of requirements. If the requirements change, then a new module by a different number is created. In this way, no baseline software capabilities are lost.
3. All modules follow the standard module format. Any deviations from the standard should be considered high risks and carefully documented.
4. All calls to external modules follow the standard calling convention, i.e., test on the module's logical first. Failure to follow this convention results in a loss of configurability.

5. All modules linked into the large executable are selectively activated via the program's input file.

Adherence to these ground rules is instrumental in satisfying the goals and requirements described earlier. One of the interesting side effects of these rules is that they tend to make programmers think more and do less trial-and-error coding. Maybe this is because programmers find it more difficult to change a module they SAID was done, instead of doing a more complete job on it BEFORE saying it is done.

Testing

Now that we have looked at some ground rules for the generation of code, let's consider a test philosophy supported by the tools. The following are some levels of testing:

1. Standard testbed configuration,
2. Demonstration of program flow,
3. Demonstration of module results, and
4. Determination of module timing requirements.

Standard Testbed Configuration

There is a standard testbed configuration built into the AVSIM program that allows programmers to test their code in a "stand-alone" mode and still be part of the total program. All the programmers have to do is activate these "testbed" modules and any others that they want to support the test. These stand-alone configurations usually become diagnostics and frequently establish baseline capabilities simply because they are documented tests that are part of the finished product.

When programmers have completed their modules and off-line testing, they are ready to test the modules' integration with the user program.

The skeleton testbed input file is shown in Figure 7. It illustrates the generic interface, AV0511, AV0512, AV0513, to which a user can assign test modules. Note that comment lines in the input file begin with "*."

Demonstration of Program Flow

After generating a test file to activate their module(s) and any other supporting modules, programmers demonstrate

that their modules are being executed in the desired context of the total program. This is usually done by requesting a TRACE (module-related function) of all modules. This will tell the programmers the order in which their modules are executed. Next, the programmers demonstrate that the program flow within their model is correct. This is usually provided by the same TRACE data. Shown in Figure 8 is part of a trace file for the tenth cycle of a simulation run. It showed the tester that the earth and atmosphere routines were called in the desired order.

```

*      Enable modules that might be required.
*
*      Activate the AV CALL modules
DEFINE_MOD(CONF(9307))=.TRUE.
ACTIV_MOD(ALL.CONF(9307))=.TRUE.
*

*****
*      Tailor this section to suit your test.
*
*      Activate generic diagnostic Initialization
ACTIVE_LIST(0511)=TRUE
*
*      Activate generic diagnostic Cycling
ACTIVE_LIST(0512)=TRUE
*
*      Activate generic diagnostic Post-Processing
ACTIVE_LIST_(0513)=TRUE
*
*
*      Diagnostic Initialization
REF_LIST(0511)= fill-in
*
*      Diagnostic Cycling
REF_LIST(0512)= fill-in
*
*      Diagnostic Post Processing
REF_LIST(0513)= fill-in
*
*****
*
* Object #1 is a fill-in diagnostic or test
N_OBJECT=1      OBJECT=DIAGNOSTIC

```

Figure 7. Testbed Input File.

```

ICYCLE= 10
.
.
1510 ! SAMPLE GLOBAL POINT DATA
1520 ! SAMPLE EARTH DATA
1525 ! I/O INTERFACE TO AV1526 - GENERIC EARTH DATA
1526 ! GENERIC EARTH DATA
1540 ! SAMPLE ATMOSPHERE DATA
1560 ! I/O INTERFACE TO AV1565 - GENERIC AIR DATA
1565 ! GENERIC AIR DATA
.
.

```

Figure 8. Trace of Program Flow at the Module Level.

Demonstration of Module Results

After the programmers have verified that their program flow is correct, they are ready to demonstrate that the outputs from their modules are valid. The programmers can accomplish this by requesting a DEBUG (of outputs) of selected modules in their program. After the results have been verified, the programmers are ready to time their modules. Shown in Figure 9 is part of a debug file for the tenth cycle of a simulation run. It shows the values output by the GENERIC EARTH DATA module, AV1526.

Determine Module Timing Requirements

Once the outputs of a module have been validated, the programmer can determine the module's timing requirements by requesting a TIMING of that module in the real-time environment. If the module is an executive, then the recorded time reflects the timing requirements of all the called modules (i.e., model). If the model is too slow, then the programmer can time the called modules individually to determine which module's implementation is too slow. Shown in Figure 10 is part of a timing output file for the avionic simulation. It shows the times required to execute the GENERIC EARTH DATA module, AV1526, every tenth cycle. Note that the

times vary, indicating some system overhead such as interrupt handling.

FUTURE DEVELOPMENTS

One of the requirements stated at the beginning was the adaptability of the product (software program) to change. The program and approach have already changed several times to become what they are today. More changes are planned or are in progress. These changes may formalize some process to make it more efficient, or they may add new tools or languages, or they may address areas that need more attention.

One of the areas of current development is the formalization of the software data base. With the increase in the number of modules, we have found that opening individual permanent files is too time consuming and are transitioning to the formal data base, DATATRIEVE. Since we want the developers to still feel like they are working on individual modules, we are also building an interface program to hide the data base from them.

```
ICYCLE= 10
1526 ! GENERIC EARTH DATA
GRAVITY_G      = 0.32081070E+02
RADIUS_EW_G    = 0.20925742E+08
RADIUS_GEOCEN_G = 0.20925742E+08
RADIUS_NS_G    = 0.20785654E+08
RNS_REQ_G     = 0.99330546E+00
```

Figure 9. Debug of Program Values at the Module Level.

```
ICYCLE= 10
1526 ! GENERIC EARTH DATA
END_TIME= 3346.20 - START_TIME= 3332.00 CYCLE_TIME= 14.20 MSEC

ICYCLE= 20
1526 ! GENERIC EARTH DATA
END_TIME= 5381.60 - START_TIME= 5380.40 CYCLE_TIME= 1.20 MSEC

ICYCLE= 30
1526 ! GENERIC EARTH DATA
END_TIME= 1030.60 - START_TIME= 1029.40 CYCLE_TIME= 1.30 MSEC
.
.

ICYCLE= 60
1526 ! GENERIC EARTH DATA
END_TIME= 560.00 - START_TIME= 557.20 CYCLE_TIME= 2.80 MSEC

ICYCLE= 70
1526 ! GENERIC EARTH DATA
END_TIME=2585.60 - START_TIME= 2584.40 CYCLE_TIME= 1.20 MSEC
.
.
```

Figure 10. Timing of Program Performance at the Module Level.

An issue that is being addressed now has to do with training new personnel and the centralization of tools. In the past, new people were taught by example and given manuals to read. As a result, their learning was haphazard and tended to be incomplete. Now, we are using FMS, with the data base task above, to build an interactive, menu-driven program. By having an extensive HELP feature and a tutorial mode in this program, we hope to remedy this problem.

One area that is not addressed by this approach is the requirements generation process. As such, some methodologies (like Yourdon) and tools (like SREM and PSA/PSL) are being studied. These requirements tools are still very young, though, and many programmers are not yet used to these tools.

Some consideration is also being given to incorporating the language Ada into the system. Since this is another tool lacking maturity (especially in view of real-time run requirements), it may be some time before Ada is fully incorporated into our software development methodology.

An interesting problem that arises from carrying modularity and configurability to these lengths is that of keeping track of all the modules and when they are supposed to be used. This problem usually shows up when someone wants to activate someone else's work (modules) in addition to their own. This information is usually passed by word of mouth, resulting in needed modules being left out, and unnecessary ones being activated. We are currently looking at tools (similar to DEC's XCON for configuring hardware components) for configuring our software modules.

SUMMARY

By addressing the areas of goals, requirements, concepts, tools, and environment, an approach for defining and building large, modular software programs was proposed. The approach included some concepts not usually found in other approaches, such as configurability, and a built-in configuration control system. It also included some tools for automating the generation of software modules and the updating of standard documentation forms.

This approach was adopted by the A4/AV8 simulation project at the Naval Weapons Center for its avionic simulation (AVSIM) in 1979. The AVSIM program currently consists of more than 200 modules and can be configured to satisfy both A4 and AV8 requirements. The program is run on a multi-VAX-11/780 system using shared memory and real-time peripheral equipment. The approach has been found suitable for the project's needs and will continue to be modified and expanded to adapt to new requirements.

This work was supported by the A4/AV8 Program Office and the A4/AV8 Facility Branch at the Naval Weapons Center, China Lake, California. Many thanks to all the members of the team who contributed to the effort and were patient during its development.

REFERENCES

- (1) Cross, F. E. AVSIM MANUAL for the Avionic Simulation Program. China Lake, California (1981).
- (2) Chief of Naval Material Weapon System Software Development, MIL-STD-1679 [Navy], AMSC No. 23033. Washington, D. C. (1978).
- (3) Weiss, Dr. E. H. STRUCTURED DOCUMENTATION PROGRAM I: Software Manuals for Users and Customers (CDR-01-V1). Madison: Carnegie Press (1980).
- (4) Beyer, T. FLECS: USER'S MANUAL. Oregon: University of Oregon (1981).
- (5) Cross, F. E. "Module Generator: A Practical Definition for a Software Module." In Proceedings of the Computer Software Applications Conference 1983. November 7, 1983, Chicago.
- (6) Cross, F. E. and Vigmostand, E. B. "USER MANUAL for the Model Builder" (working paper).

Module Generator Format

TITLE: Module Title

LIBRARY NUMBER: 0001

PURPOSE:
(Brief description of the module's purpose and method)

DESIGN REQUIREMENTS:

INPUT

OUTPUT

LOCAL

EXTERNAL

AV000X ! Module Title

END

AUTHOR: Last, First MI

DATE: MM-DD-YR

OPTIONS: TRACE, DEBUG, TIMING,
DICTIONARY=AV_DICT:MASTRAV.DIC

STATUS: Stub
(Stub, Designed, Coded, Final or Incomplete, Tested)

METHOD:

 SUBROUTINE AV0001 ! Module Title

C

C Format for calling external modules via the generic AV_CALL

C CALL AV_CALL(000X) ! Module Title

C

 RETURN

 END

DOCUMENTATION: Reference "STRUCTURED DOCUMENTATION" by Dr. Weiss

HEADLINE

AV0001 Title of Documentation

KEYWORDS keyword1, keyword2, ...

SUMMARY

 (100 - 150 WORDS AVERAGE)

TEXT

 (300 - 700 WORDS AVERAGE)

EXHIBITS

 (GRAPHICAL DISPLAY)

 END

G. Cort and R. O. Nelson
 Los Alamos National Laboratory
 Los Alamos, New Mexico 87545

We present the details of a strategy for acquiring a comprehensive set of software tools to support a life-cycle-based software development methodology. Specific criteria for evaluating software tools are presented with examples from the Los Alamos tool kit. The importance of an environment to support both the tools and the methodology is discussed.

Introduction

The environment in which modern software development projects must operate has become increasingly more complex as the capabilities of computing hardware have advanced, and as more functionality and performance have been required of the associated software systems. The earliest attempts at addressing these problems led naturally to the first practical software tools: text editors, assemblers, compilers and linkage-editors. Each of these tools rapidly proved to be of enormous utility. To this day these tools comprise the basic tool kit of every software developer.

As applications continued to increase in complexity the simple tools were supplemented with methodologies designed to organize the overall software development effort. These methodologies identify and emphasize activities which are not directly associated with code generation, and as such the simple tool kit is of little use in supporting them. For this, and a variety of other reasons, most comprehensive software development methodologies have evolved virtually without supporting software tools. As a result, the implementation of these methodologies (and by extension the entire software development process) can be a tedious, time-consuming, labor-intensive activity.

This situation prevailed until approximately the beginning of the current decade. Previously, the tools which existed to support software development activities generally were developed in-house by individual programmers to address a specific need or activity. The use of such tools was limited almost exclusively to their authors, and these tools were not supported or maintained in any organized fashion.

Within the past five years, however, many tools supporting software development activities have become generally available. The most casual market survey reveals literally hundreds of commercially available software products designed to address some aspect of the software development activity. In addition, many larger organizations develop proprietary tools for universal in-house utilization.

Moreover, there has been a fundamental metamorpho-

sis in the attitudes of software managers and developers regarding the use of software tools. Managers widely perceive software tools as vehicles through which more powerful development techniques and methodologies can be implemented without introducing the stifling overheads which previously were a necessary accompaniment. Tools are perceived as a means of improving programmer effectiveness, increasing productivity and enhancing the overall quality of the delivered software product.

Developers, on the other hand, expect their tools to free them from many of the tedious, time-intensive and boring activities associated with many aspects of modern software development methodologies. Tools are expected to simplify the routine of implementing the methodology and to identify (and often to mitigate) errors which might otherwise remain undiscovered until a subsequent phase of the development project. In this capacity, tools should support the development, by iterative refinement, of the final deliverable software product. In this respect, software tools may be considered the analogs of computer-aided design systems which are already being employed so successfully to develop electronic hardware.

Although the expectations described above for both groups seem eminently reasonable, it is often the case that developers and/or managers become extremely dissatisfied with the operation or performance of their newly acquired software tools. It is often reported that a tool does not perform the function for which it was acquired, or that the task performed is too trivial to be useful.

Many tools are difficult to use and require a greater commitment of resources to training than either managers or developers are willing to absorb. Most often, however, it is discovered that a suite of tools (often acquired from different sources) cannot be made to work together. Consequently, developers and managers must devote inordinately large blocks of time to interfacing the tools together. Often this requires extensive manual effort on an ongoing basis.

The ultimate result of all of these factors can be a significant degradation of productivity, product quality and project morale. Project personnel at all levels perceive that utilization of the tool(s)

significantly complicates the performance of their duties, and a considerable negative backlash may develop.

The LANSCE Approach

For the past three years at the Los Alamos Neutron Scattering Center (LANSCE), we have been involved in the development of a moderately large (150K lines) real-time data acquisition system to support condensed matter and nuclear physics research. The size of the project, as well as the mission-critical nature of the software, dictated the use of a formal software development methodology. Owing to stringent time constraints and the very limited resources available to the project, we decided from the outset to utilize software tools extensively, thereby to automate many aspects of the methodology and to reduce significantly the corresponding overheads.

Also at an early stage of the LANSCE project, it was recognized that most of the tool utilization problems described in the preceding section are byproducts of the tool evaluation and acquisition processes and do not necessarily reflect deficiencies of the tools themselves. We therefore resolved to determine, before the fact, those characteristics of a software tool which are necessary for the tool to be useful. In addition, we decided that, whenever possible, tools would be acquired from commercial sources. However, any tool which could not satisfy all of our criteria would be automatically excluded from consideration. It should be noted that this policy resulted in the vast majority of our tools being developed in-house.

The four criteria employed by the LANSCE software development project to judge the efficacy of a software tool are utility, scope, interface and integration. These concepts are discussed in detail in the following sections.

Tool Utility

Fundamental to the ultimate usefulness of any software tool is whether it performs the task(s) for which it is acquired. Although this statement may seem obvious to the point of being trite, it is misapplied or overlooked in a surprising number of instances.

The most common error which is committed in evaluating the utility of a software tool is to mistake a tool which performs a job for a tool which does the job. Almost all tools perform some function, but whether that function addresses precisely the job which you want done makes all the difference. Tools which are not carefully matched to the task at hand create bottlenecks and generate frustration by forcing users and managers to compensate for the inadequacies or idiosyncrasies of the tool.

The LANSCE system employs two different classes of tools: configuration management (CM) tools and developer support tools. The utility required for each class and for each tool within a class was formally specified and reviewed before the acquisition process was begun.

The CM tools were specified to perform all transactions between project participants and the local secure repository of software baselines. Constraints were identified to define all aspects of the operation of these tools.

Additional CM tools were specified to perform automated rebuilds of all or part of the software system. Other tools were specified to provide a comprehensive, automated, configuration accounting system to track the changes to, and to record the status of, every software component in the system.

Developer support tools were specified to assist in the implementation of various aspects of the local software development methodology. Tools were specified to support development of each of the five software baselines which must be met for each deliverable product. These tools provide extensive automated support for the evolutionary development of comprehensive inline documentation for each software baseline. This includes multiple levels of procedural documentation (including structured pseudocode), a comprehensive data dictionary, and module structure charts. In addition, these tools enforce facility coding and documentation standards, promote a high degree of uniformity, and ensure completeness of the final product.

Other support tools simplify and extend the developers interface to standard compilers and linkage editors. These tools virtually eliminate the need for developers to create and maintain special command files to compile and/or link computer programs.

An additional tool provides users the run-time option of choosing a particular version of an application for execution. This is particularly useful when an uncertified version (perhaps under maintenance) has a feature which is required by some subset of the user community, but which is not generally available in the certified version. In this situation, any user requiring the uncertified software version may choose to execute it without affecting the environment of any other user on the system.

Because the utility of each of these tools was specified well in advance of acquisition, the acquisition and evaluation processes were significantly simplified and the utility of each tool was guaranteed. As a result, each tool was integrated into the existing environment without undue disruption.

Tool Scope

The issue of tool scope requires that a software tool address the targeted problem at a level which is appropriate to the context in which the problem is encountered by the user. Scope and utility are closely related, owing to the fact that inappropriate scope can often nullify a tool's utility. The issue of tool scope often reduces to whether the tool performs the expected function without requiring the user to extensively reconfigure either the tool or the environment in which it functions.

The scope which is appropriate for a specific tool often depends strongly upon the purpose, desired versatility and audience which the tool must serve. The LANSCE documentation tools, for example, exhibit a relatively narrow scope. These tools are designed to perform a particular function at a well-defined stage in the development life cycle. Their inputs and outputs are standardized and simplified to accommodate the LANSCE methodology and environment. The user expects these tools to perform a specific documentation task expeditiously and without requiring special configuration. From the user's perspective, the inability of the tool to deal with the documentation needs of a wide range of methodologies and project organizations is unimportant.

Tools that exhibit a narrow scope are therefore generally preferred in situations that are characterized by a narrow class of problems and/or a large community of relatively unsophisticated users. The narrow-scope tool satisfies the desire of most individuals to be tool users -- not tool builders.

The LANSCE CM tools are characteristic of the wide-scope alternative. In order to meet our CM requirements, an environment consisting of many primitive tools was acquired. The environment provides a procedural command language which allows the primitives to be combined into highly customized applications. The CM tools which are thereby constructed are general purpose in nature and collectively exhibit a very wide scope. That special knowledge and experience is required to utilize the CM tools is unimportant in light of the very restricted user base and the wide variety of problems which can be addressed.

Tool Interface

By virtue of being the most visible feature of any software tool, the user interface plays the most important role in determining whether a tool will be accepted by its intended audience. As the component which interacts most strongly with a subjective human being, the tool interface should be at least as carefully designed, tested (and evaluated!) as the tool's functionality. Yet it is often obvious that a tool's interface has been hastily thrown together and tacked onto a "working" tool just prior to release.

An effective interface must at once be simple, appropriate, uniform and consistent. Simplicity avoids presenting the tool user with a confusing plethora of choices or options. The interface should be tailored to the level of sophistication of the user. Oversimplified interfaces engender feelings of impatience and inefficiency on the part of their users, whereas interfaces which are too complex cause confusion and frustration.

Consistent interface syntax and semantics are absolutely necessary for a user to gain confidence in the associated tool. In many instances, problems assessed as failures of tool functionality can be traced to the (unintentional) misapplication of an inadequate or inconsistent interface.

The importance of uniformity among tool interfaces

cannot be overemphasized. Particularly in cases in which a suite of tools is employed, it is of enormous advantage if a uniform interface is employed by all tools. Uniformity results in the minimization of the effort required of a user to become familiar with the operation of the tool set. Because all tool interfaces are semantically and syntactically equivalent, moving between tools becomes much more comfortable.

All LANSCE tools employ a standardized interface by utilizing the VMS Command Definition Utility (CDU) to define a DCL command to invoke each tool. This guarantees that the interfaces are uniform across the entire spectrum of tools. The syntax and semantics associated with DCL commands, parameters and qualifiers are familiar to even our most casual users and therefore impose no additional overhead to assimilate. Combined with online help and user documentation in the VMS style, the interface is appropriate for users at essentially any level of sophistication. A derivative, but very important, benefit of this approach is the extreme ease with which interfaces can be written -- the need to develop parsers, menu drivers or command language interpreters is completely eliminated.

Tool Integration

The last criterion for software tool suitability concerns the issue of tool integration. This criterion expresses how well a tool fits into the environment in which it is expected to function. The integration issue accounts for the most insidious cases of tool failure -- instances in which a tool (or suite of tools) can be demonstrated to have the appropriate utility and scope, as well as an effective interface, yet still contributes negatively to a project.

A successful software development tool must integrate effectively with three external entities: other tools, the software environment, and the development methodology. It should be realized that these external entities cannot generally be considered independently.

For example, the choice of a development methodology will to some extent determine the environment and tools required to support it. However, the environment (operating system, local data bases, etc.) may play an important role in determining the appropriate methodology as well as in determining which tools can be effectively supported. A powerful set of tools, on the other hand, can significantly enhance the environment.

The point of identifying these mutual dependencies is to establish the importance of promoting and nurturing a healthy symbiotic relationship among the tools, the environment and the methodology embraced by a software development project. Any attempt to consider one of the three in a vacuum and without regard for the others, must necessarily have an adverse effect upon the collective system.

That tools must integrate with each other is a necessary consequence of the size and complexity of the tasks to we apply them. Text editors must produce source files which can be processed by compilers, and the resulting object modules must be

compatible with linkage-editors which use them as input. Within the context of these simple, familiar tools the concept of tool integration seems natural and obvious.

Within the larger context of the typical software development project, however, tool integration becomes a far more subtle problem. This problem can be compounded when tools are acquired from multiple sources.

Often, a tool employed during one phase of a software development project produces information which is inappropriate or incompatible for use with another tool in a subsequent phase. In other cases the information is appropriate but the format in which it is presented is not. Consequently, project managers and developers are forced to take remedial action (often in the form of additional manual procedures) to transform the information which is produced by one tool into a form or format which is acceptable to another. In addition to draining precious resources from the project, these activities lead to the tools being (correctly) perceived as encumbrances by project participants. Project managers rapidly become alienated when expensive tools produce no appreciable benefits or even detract from overall productivity.

The problems generated by lack of integration of most software tools could be ameliorated to some extent if the tools were integrable into a common software environment. Here, the term software environment refers to a common body of information to which all tools have access. An operating system is a simple example of such an environment, although modern software development projects already utilize environments comprised of extensive data bases and other complex information structures.

Unfortunately, very few software development tools (particularly commercially available tools) have the flexibility to integrate into customized software environments. Some tools define their own environments, within which they can function rather effectively, but with only rare exceptions, they cannot integrate effectively with any external environment.

Finally, the lack of integration of software tools with standard software development methodologies is a most serious deficiency. Indeed, it was this problem which primarily influenced the LANSCE project to develop its own methodology (1) and to support it with locally developed tools. These statements should not be misconstrued to mean that there are no software tools available to support the various phases of a software development project, but merely that there is no tool, set of tools or combination thereof which lends automated support to the entire software development activity without encountering severe integration problems between its components and the underlying environment.

The LANSCE approach to addressing the integration problem has been to define the methodology, environment and tools in terms of each other. Our methodology is therefore designed to be implemented with, and supported by, a comprehensive set of software tools. The tools and methodology together

define a unique software environment which encompasses the methodology (software baselines, standards, tools) as well as extensive information structures used to meld the methodology and tools into a cohesive, cooperating structure. The degree of integration achieved in this manner has been extremely high and provides a stable foundation upon which new tools, more powerful methodological features and environmental enhancements can be added.

Conclusions

In the preceding sections we have attempted to present our philosophy for acquiring and implementing a comprehensive suite of software development tools to support a life-cycle-based development methodology. The philosophy is based upon four criteria which we believe to be essential characteristics of successful software tools.

It has not been our intention to provide detailed descriptions of the purpose and functionality of each tool in our system, or to expound upon the details of our software development methodology. These topics have been discussed elsewhere (2,3). It was our purpose to present a rational framework within which a comprehensive set of software tools can be specified and ultimately acquired.

Finally, it must be stressed that the criteria which we have advanced for useful tools are quite stringent. Very few commercial tools have been able to satisfy them. Regardless, careful and thorough specification of software tools is crucial to any successful program of acquisition. In the final analysis, it is far better to develop your own tool, or to forego acquisition altogether, than to introduce an inadequate, frustrating and disruptive influence into your software development project.

References

1. G. Cort, J. A. Goldstone, R. O. Nelson, R. V. Poore, L. Miller and D. M. Barrus, IEEE Trans. Nuclear Science, NS-32, 1439, 1985.
2. G. Cort, in Conference on Software Tools, John Manning, ed., IEEE Computer Society Press, 1985.
3. G. Cort and D. M. Barrus, Proceedings of the First Meeting of the Softool Users Group, September, 1984.

Acknowledgement

This work was performed under the auspices of the U. S. Department of Energy.

G. Cort and R. O. Nelson

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

We present the details of a software development methodology which addresses all phases of the software development life cycle, yet is well suited for application by small projects with limited resources. The methodology has been developed at the Los Alamos Neutron Scattering Center (LANSCE) and was utilized during the recent development of the LANSCE Data Acquisition Command Language. The methodology employs a comprehensive set of software tools to support development and maintenance of exhaustive documentation for all software components. The impact of the methodology upon software quality and programmer productivity is assessed.

Introduction

With the rapid advance of hardware technology, and the increasing complexity of software applications, modern software systems are becoming much larger and significantly more difficult to manage. For numerous reasons, these factors have had a particularly serious impact on the quality of scientific software. Whereas large engineering organizations can devote considerable resources to software management, most scientific software development projects are characterized by small staffs with very limited resources. That most scientific programmers have little or no familiarity or experience with software management issues exacerbates the problem. As a result, software developed in this environment is often fragile, haphazardly designed, difficult to use and impossible to maintain. Documentation, if it exists at all, is frequently inconsistent and inaccurate.

In order to address these problems, and with the ultimate goal of improving software reliability and maintainability, the Computer Group at the Los Alamos Neutron Scattering Center (LANSCE) has established a comprehensive development methodology for scientific software. This methodology employs many of the strategies and techniques utilized by large projects to manage the software development process, but significantly reduces the associated overheads.

Assumptions and Implementation Strategy

The software systems that are developed and maintained by the LANSCE staff can be characterized as mission-critical. (This is particularly true of the real-time systems.) The operational lifetime of most of these systems is expected to be rather long--approximately ten years. The LANSCE methodology must, therefore, promote the development of highly reliable software that can be maintained by a small staff. We base this strategy upon three assumptions.

1. Large-scale methodologies are not suitable for use by projects of small or intermediate size. Very effective methodologies already exist for managing software development projects. Historically, these strategies have been pioneered by, and perfected for, large-scale software development projects which can devote considerable resources to applying them. These methodologies usually require a dedicated staff to perform time intensive activities within a highly stratified management structure. A commitment such as this usually represents only a small fraction of the total resources available to a large project. The LANSCE staff, however, consists of approximately four full-time programmers and a single manager. As would most small or intermediate size projects, the LANSCE effort would be overwhelmed by the institution of a large scale development methodology.

2. Coding is the least important activity associated with any development or maintenance operation. Coding should correspond to merely the translation of a sound design into an implementation. The most critical development activities are performed either before (specification and design) or after (testing) the coding phase. Most small projects (especially those of a scientific nature) emphasize the coding activity, often to the complete exclusion of the other phases. Reliability, consistency and maintainability are very difficult to introduce into an application during the coding phase, so many scientific applications are characterized by very low quality.

3. Documentation is fundamentally more important than code. Every software professional has been exposed to inadequately documented code, usually during a maintenance operation. The precedence of documentation over code, however, has far wider implications: over the entire software life cycle, complete documentation is crucial to the successful

operation of any software system or component. In addition to adequate maintenance documentation, requirements must be fully documented (for comparison with validation test results). A carefully documented design is invaluable when the software requires enhancement or repair. Exhaustive documentation of test coverage, specific test cases and all test results is essential for validation and verification of subsequent versions. For these reasons, the LANSCE methodology emphasizes the development of complete, consistent and uniform documentation for every phase of the life cycle.

The LANSCE methodology implements these assumptions by applying policies of universal standardization and strict modularity. Standardization of all development/maintenance activities (and the results thereof) is the simplest means for promoting uniformity and completeness in the final product. This can be accomplished without adding appreciably to the management overhead. Standards application and verification can often be accomplished with automated tools, further reducing the overhead.

In order for standards to be effective, however, a certain level of management commitment is required. Of primary importance is enforcement of existing standards. Standards must be compulsory and universally applied. Voluntary standards or selective application doom the effort to failure at its inception. In the very rare instances in which exceptions are made and standards violations are permitted, there must be significant other advantages to be gained (improved functionality, clarity of expression, etc.) other than mere convenience.

Standards, to be effective, must exist in written form. Unrecorded conventions that are presumed to be understood by all participants are often worse than no standards at all. Interpretations vary with individuals, and standards application subtly changes with time. Hence, the content and uniformity of resulting software varies radically with author and date of development.

The LANSCE facility standards address all aspects of the software development project. They establish development procedures, specify the components to be produced during each phase and define acceptance criteria. At a lower level, they promote complete and uniform documentation by defining a partitioned documentation template and rigidly specify the information required in each partition. Coding and style standards are provided to specify the implementation language, to identify illegal control structures, to establish formatting rules (indentation levels and case rules), to set naming conventions and to define modularity constraints.

The policy of strict modularity is defined in the facility standards and specifies limitations for the size and contents of a software module, as well as the allowed forms of communication between modules. The goal of this policy is to compartmentalize the software, thereby reducing side effects which are attributable to pathological intermodule connections.

The LANSCE methodology limits the number of software routines (programs or subprograms) per source file (module) to one. Module length is not to exceed 100 executable lines. Communication between modules must be performed exclusively through calling parameters. Global common data structures are explicitly forbidden. These rules promote the development of small, highly functional modules with simple interfaces. Reuse of existing modules is therefore encouraged and maintenance can be performed on individual modules without jeopardizing other unrelated components.

The Life Cycle Approach

The LANSCE methodology is based upon a software life cycle model. The model partitions a development or maintenance operation into three phases: specification, design and implementation. The implementation phase is further subdivided into coding and testing activities.

The activities that may be performed during a particular phase of the life cycle are rigidly specified in the facility standards. During the specification phase, functional requirements are analyzed, general algorithms are established, and top level constraints are identified. The design phase addresses the detailed algorithmic and procedural aspects of the software product. The coding activity translates the design information into the appropriate programming language. The testing activity is required to generate a formal test plan which specifies the test coverage and which describes each test case in detail (purpose, inputs and results required to pass the test). The testing activity must also generate a test report that contains the actual results of every test case.

Work on a particular development/maintenance cycle proceeds sequentially through each of the phases described above. Generally, all work specified for a particular phase or activity must be completed prior to beginning the subsequent phase. (The development of the test plan may, however, proceed concurrently with the coding activity.)

Peer reviews are employed to verify the completeness, correctness and appropriateness of all work performed during a particular phase of activity. Work is subjected to a mandatory review at the conclusion of each phase. For extremely complex projects, intermediate reviews may also be required (e.g. a test plan review).

The reviewing body is composed of the entire programming staff, the section leader, and a representative of the user community. Collectively identified as the Configuration Control Board (CCB), these individuals determine the completeness of submittals by comparison with standard baselines. A baseline is defined in the facility standards for every phase and activity. Each baseline details specific components that must be completed to satisfy the requirements of the corresponding phase or activity.

Submittals are also reviewed for compliance with general facility standards (coding and documen-

tation). Algorithms are critically evaluated and required changes are identified. The reviews therefore provide a means for identifying and eliminating errors at the earliest opportunity. This guarantees that each phase of the life cycle is addressed in the appropriate order and in a complete and consistent manner.

Documentation

Documentation is the cornerstone of any successful development methodology. The LANSCE strategy emphasizes the development of exhaustive documentation for each system component during each phase of the software life cycle. The software review process promotes the generation of complete, uniform, accurate and current documentation. Of equal importance to the small project is the requirement that the documentation be easily managed.

One of the problems associated with large-scale methodologies is that although they produce very comprehensive documentation for each life cycle phase, this documentation exists as many separate components (requirements documents, statements of work, design documents, user's guides, reference guides, etc.) and in many different forms (text, graphics or binary). The sheer number of components makes cataloging and tracking the various versions extremely difficult. Some entities (e.g. graphical data flow diagrams and structure charts) are tedious and time consuming to produce and very difficult to maintain. As a result many components become obsolete as the project evolves, thereby undermining confidence in all existing documentation. For small projects with limited resources, the effort required to support a documentation strategy such as this is prohibitive.

The LANSCE methodology is designed to drastically reduce both the number of distinct documentation components and their manifestations, without severely restricting the documentation scope. Only one manifestation is permitted: all documentation is required to be electronically readable and modifiable with a text editor. Documentation which does not meet this criterion (structure charts, for example) is reorganized to comply, or eliminated. Unwieldy media (e.g. paper-only copies) are thereby eliminated and the update procedure is simplified considerably. Because all documentation is computer readable, automated software tools can be employed extensively to streamline much of the effort.

The number of documentation components is reduced to three. The first component consists of all specification, design and maintenance documentation. This component is generated and maintained inline within the source code module. The second documentation component consists of a users' guide that describes the operation of the software and contains no design or maintenance information. The last documentation component contains all testing information and consists of the test plan (test description) and the test report (test results).

The inline documentation component consists of a

standard documentation template that resides at the beginning of every software module. The template is divided into ten sections, each of which is dedicated to a particular category of documentation. These categories are organized into narrative documentation and tabular documentation classes.

The narrative documentation categories are formatted as paragraphs, and are divided into sections for module purpose, history, functional description, assumptions and limitations, special comments, references and pseudocode. These sections contain information documenting the requirements, previous maintenance activities, algorithm, procedure, general data structures and special features of the associated module. Automated tools are provided to support the generation and update of narrative documentation.

Several special components of the narrative documentation class are notable. The pseudocode category contains a concise, but detailed description of the module procedure. Pseudocode is expressed as structured English statements associated with keywords that denote control information. As such, the pseudocode comprises a very high level, structured program-design language with fully bracketed syntax, and is used to express the detailed design of a module. Pseudocode is easily translated into code at implementation time, yet it is far simpler to comprehend for either the designer or the maintenance programmer. Tools are provided that format and verify the syntax of this documentation.

The second special construct provided as part of the narrative documentation is a module structure chart. This documentation uses a tree structure to represent the hierarchical organization of the module and all subordinate modules. This allows a maintenance programmer to determine the precise calling structure of a system or subsystem. Tools are provided to build the structure chart automatically and to format the structure chart in any of several ways.

The tabular documentation class is dedicated to describing specific data items (symbolic constants and variables), data types and subprograms. There are three tabular categories: interfaces (for documenting the routine's formal parameters and external files), global identifiers (for documenting objects that are visible outside the declaring routine) and local identifiers (for objects of local scope).

Each documentation table is organized into a series of columns. The names of the objects being documented are listed alphabetically in the leftmost column. Succeeding columns contain information which specifies base type, attributes, parameter passage mechanisms and other information. The rightmost column contains a definition of the object. Every identifier that appears in a module must be documented in one of the tabular categories. Automated tools are available to construct the tabular portion of the template and complete most of the entries in each table (including, in many instances, the description). These tools guarantee that all identifiers are indeed included in the tabular documentation. In addition, the

tools delete tabular documentation for objects that no longer appear in the module, and thereby contribute to keeping the documentation current. In this manner a complete, uniform and accurate data dictionary is maintained within the source code.

The LANSCE methodology is designed to produce progressive and cumulative evolutionary documentation. The LANSCE philosophy organizes all documentation according to its intended audience. Documentation that is of principal use to designers, programmers and maintenance personnel is placed inline with the appropriate source code. Thus, all specification, design and maintenance documentation is maintained within the corresponding source module and the test plan is contained within the testing software. This close association of documentation with source code has the advantage of making all documentation immediately available with the associated sources. This enhanced access to the documentation makes documentation updates simpler to perform and less likely to be forgotten.

The users' guide documentation is targeted for individuals who do not require the detailed knowledge of a developer. Users' guides are therefore organized to contain practical information that is required to integrate the software into an application. Detailed information regarding the design and implementation of the software is deliberately omitted as irrelevant to the users' needs. As such, the users' guide documentation is maintained separately from the source code.

The above organization for documentation allows the various software baselines to be expressed entirely in terms of documentation components. In this manner, as the development progresses through each stage of the life cycle, each baseline contributes to completing the documentation for a particular system or module. This approach produces source modules that evolve from their documentation. Every module begins as an empty documentation template that is completed in stages until, as the final step, code is generated.

The specification baseline, for example, is comprised of specific categories of narrative and tabular documentation that define the requirements and constraints for a software system. The modules in which the specification baseline is built eventually evolve into the system's executive routines. Requirements and constraints are specified in narrative format in the purpose, assumptions and limitations and special comments categories. References are provided as needed. Top level interfaces are defined in the interfaces documentation table. The general algorithm is described in the functional description narrative category.

The design baseline is also constructed principally within the documentation template. Executive logic design is appended to the specification baseline. This corresponds to pseudocode that details the top level procedural flow as well as a preliminary structure chart to document the system calling hierarchy. Detailed subprogram design is then accomplished by completing all narrative documentation and the interface tabular documentation for every module in the system (as derived from

the structure chart). This includes a functional description and pseudocode for each module. The users' guide is then composed for the software system. It should be noted that through this point in the development no code has been written.

Upon completion of the design baseline virtually all documentation which is required to operate and maintain the system is complete. Only the data dictionary documentation and the testing documentation remain to be developed. The former can be (to a large degree) automatically generated after the coding is completed. The latter is developed concurrently with the coding activity and as a result of executing the test procedures.

The advantages of this approach are numerous. Detailed documentation is available to guide the programmer in translating the design into code. The resulting code is therefore much more likely to reflect the requirements and design than if no such guidance were available. Several levels of procedural documentation (functional description and the more detailed pseudocode) are provided, thereby simplifying the implementation and future maintenance operations. Problems of omission and inaccuracy which are normally associated with retrofitting documentation to existing code are avoided entirely.

The test plan baseline contains documentation for each test case to be executed. For each test case, this information includes a unique identifying number, a description of the function tested, the inputs required and a description of the expected outputs (for comparison with test results). The test data pack is also specified as part of the test plan baseline. The test procedure is ultimately derived from (and will reside in the same module as) the test test plan baseline.

The implementation baseline consists of all source modules (with completed documentation), the test plan and test procedure module, the users' guide and the test report. In particular, all tabular documentation is completed. The implementation baseline thus emerges as a fully documented, progressively developed software product.

Configuration Management

The last element of the LANSCE methodology is a strategy for controlling access and tracking changes to the components of each software system. This configuration management strategy is based upon a philosophy of modular maintenance: during the conduct of any maintenance operation, only those modules that actually require modification are made available to the maintenance programmer. All maintenance operations, including a list of modules to be modified, must be certified by the CCB prior to the initiation of any maintenance activity.

These very strict policies derive from the mission-critical nature of the LANSCE software: failures in system components can result in total disruption of the facility, with serious economic, political and scientific consequences. By rigidly controlling access to system sources, errors introduced through inadvertent or unauthorized access are eliminated. Backup configurations are

provided for use in the event of the failure of a primary system component, thereby allowing the system to remain operational (although at reduced capability) while the primary component is under repair.

The LANSCE configuration management system is built around an automated tool that provides a secure repository for controlled modules as well as a command language for automating configuration management activities. Only one individual, the Configuration Manager (CM), has access to controlled modules. This individual is responsible for moving software (implementation baselines) into the controlled environment after certification by the CCB, and for releasing modules which have been approved for maintenance to the designated maintenance programmer.

In order to request maintenance (either enhancement or repair) on a particular system or component, a work request must be submitted to the CCB. The request is reviewed for accuracy, relevance and importance. If approved, it is assigned to a programmer for determination of the work required and the modules affected. The analyst's recommendations are reviewed by the CCB and, if approved, the designated modules are released to a programmer for modification. The standard LANSCE methodology is utilized to perform the maintenance activities and review the results. Upon completion and subsequent certification by the CCB, the new implementation baseline is admitted to the controlled environment.

Logs of work requests are maintained to track the status of all software products. Automated tools are provided to simplify the process of completing, submitting and cataloging these requests. Other tools are provided to automate the process of moving modules into and out of the controlled environment as well as for rebuilding systems and libraries affected by maintenance operations.

Conclusions

In the preceding sections we have detailed the features of a powerful software development methodology that is suitable for use by projects of small or intermediate size. The methodology is designed to maximize the reliability and maintainability of software components developed from it. This is done by emphasizing phases of the development life cycle that are generally ignored by small projects: specification, design and testing. Exhaustive documentation is progressively generated for each phase, and the coding activity is demoted to consume minimal project resources. A suite of automated tools is provided to support all phases of the development effort.

Although the LANSCE methodology is designed to minimize the overheads imposed upon programmers and software managers, application of the methodology undoubtedly increases the development time for a software product. Estimates of the additional time required for software development are necessarily subjective, but a conservative estimate for this methodology indicates a programmer overhead of

approximately 200%. Utilization of the configuration management strategy described is expected to place a burden of an additional 15% upon one participant.

The predicted overheads are based upon the increased effort that must be expended by each participant during the specification, design and testing phases (the time required for coding should actually decrease significantly). Peer reviews also consume a significant amount of time, both in preparation and execution. In order to maintain a proper perspective, however, it must be noted that the resulting software product is of significantly higher quality than a comparable system developed with traditional methods. Documentation is complete and accurate. Formal testing introduces a level of reliability that cannot be attained through the ad hoc exercising that might otherwise be performed. In summary, although the development time is tripled, the resulting software is exceedingly more reliable, maintainable and robust. This enhanced quality is expected to manifest itself in much longer mean times between failures as well as simplification of enhancement/repair activities.

The LANSCE methodology may also be tailored to the requirements of individual projects, particularly in terms of the degree of implementation and accompanying overhead. The simplest subset to implement retains the policy of evolutionary documentation and combines the specification and design phases. No peer reviews or formal testing are performed. This implementation significantly reduces the overheads associated with the methodology, although the quality of resulting software can also be expected to be much lower. The addition of peer reviews provides a very powerful means for improving software quality at the expense of increased overhead. A formal testing program and configuration management procedures then provide the full benefit (at maximum cost).

Whatever the degree of implementation, the LANSCE methodology promotes enhanced software quality by shifting a major portion of the development effort to the early stages of the life cycle. Regardless of project size, increased attention to specification and design issues will always produce a better result.

Acknowledgement

This work was performed under the auspices of the U. S. Department of Energy.

NETWORKS SIG

A HIGH SPEED LOCAL AREA NETWORK OUTPUT NODE

Robert J. Aiken
Networks Division, 8234
Sandia National Laboratories, Livermore, Ca.

Prepared by Sandia National Laboratories
Albuquerque, New Mexico 87185
and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-76DP00789

ABSTRACT

Sandia National Laboratories, Livermore has implemented a high speed local area network (LAN) Output Node in order to provide the user community with high quality non-interactive output in a fair and efficient manner. This report will discuss the major ideas and issues that were encountered during the implementation cycle of the output node.

SNLL provides for the computational needs of its scientific community with a high speed local area network which is comprised of two CRAY 1Ss running the Cray Time Sharing System (CTSS), two IBM 4341s running the common file storage (CFS) system on top of IBM's MVS operating system, and seven VAX 11/780s running DEC's VMS operating system. The backbone for this network is the 50 megabit/sec Hyperchannel from Network Systems Corporation (NSC) and the 70 megabit/sec Computer Interconnect (CI) from DEC. Two of the VAX 11/780s provide a special service to the network. A gateway VAX 11/780 is used to facilitate the transfer of files from a worker VAX 11/780 to either a CRAY 1S or the CFS system. The output node VAX 11/780 provides network wide access to non-interactive graphics devices and laser printers. See Figure 1.1 for a representation of SNLL's network.

The implementation of the output node was performed in stages, with the first phase being the installation of two XEROX 8700 laser printers. Subsequent phases of the output node added QMS laser printers and DICOMED film recorders. Many of the design concepts and implementation techniques that were used during the first phase were also applicable to subsequent phases. Figure 1.2 shows the current output node configuration with a more detailed view of the XEROX to VAX interface found in Figure 1.3.

There are five different conceptual areas of a network that must be addressed when implementing an output node. The user interface, the worker node, the network protocols, the output node and the output devices must be integrated in such a manner as to provide the user with access to high quality output devices from any node in the network. These conceptual areas exist in all LANs that provide a "server" function for the whole network regardless of the networks' respective bandwidth.

The first area is the user interface which allows the user to invoke the utility(s) needed to have a file processed by the output node. The utility must be designed for ease of use by providing acceptable default parameters and a friendly invocation format that is forgiving of user errors. This interface must be consistent on every worker node regardless of machine or system type. The ability to have a file processed by the output node also infers the need for a status reporting tool that the users can invoke in order to follow the progress of their jobs through the system.

The second area constitutes the system type functions of the worker node where the user is located. The main concerns of this area are what kinds of system software and languages are available for the implementation of the user and network interfaces. These will not always be the same for each node due to the heterogeneity of the network; therefore each participating nodes' software must be tailored to provide a consistent interface at both the user and network area. This area is not as concerned with what the users sees at the interface level or how the user invokes the utility but how to transform that users' request into the proper system/network requests to produce the expected results.

The third area can be categorized as the network area due to the fact that its primary function is related to the actual error free transfer of a file from one node to another. In order to accomplish this goal all worker nodes must agree upon a system independent network protocol such that they can properly communicate with each other. There must also exist some common file definition that all nodes agree upon, which defines the type of file being transferred and any other pertinent information needed to properly process that file once it reaches its destination.

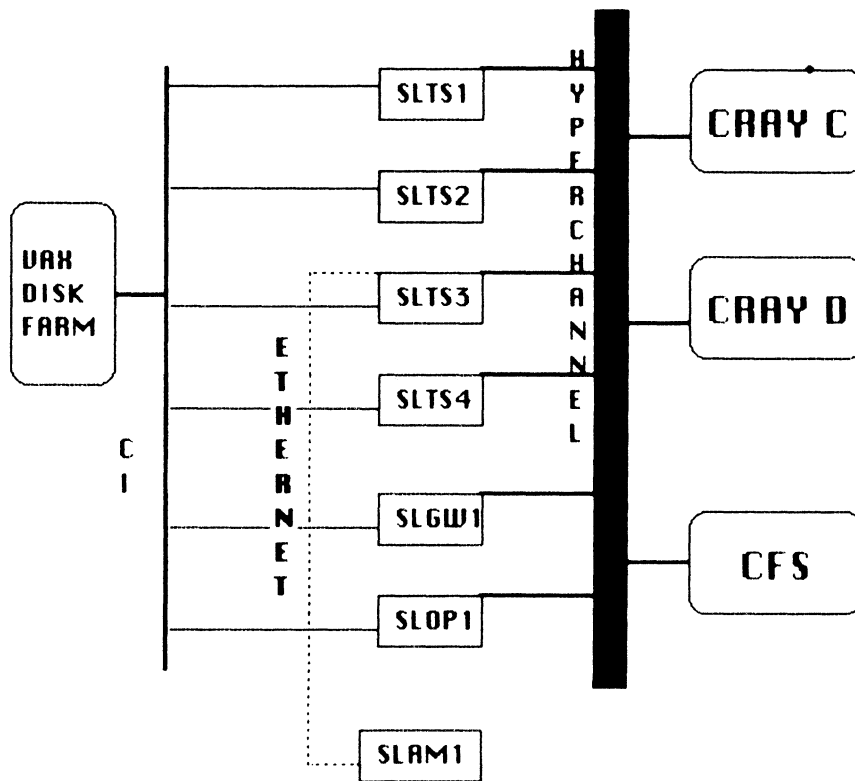
The fourth area is the output node itself. This machine is dedicated to providing fair and reliable access to the different devices it supports. The main concerns of this area follow. Temporary disk storage must be provided for jobs until they have been successfully processed. The output node must provide an interface to the network in order that it can receive the files that need processing on its devices. Once a job has been received from the network it must be scheduled for the destination device in a manner that is both fair and efficient. After a job is chosen as being next in line for processing the output node must provide a device interface that allows for the proper use and control of that device even if the target device is not host compatible. At all times the output node must be able to detect errors and recover from them if at all possible.

The final area is the actual output device itself. There are certain devices, such as the XEROX 8700 laser printers, that are microprocessor controlled and need to be programmed to perform their tasks properly. This class of machine is starting to replace the older "dumb" devices and will make this interface that much more complicated due to the fact that the communications between the output node and the output device becomes a point to point network connection.

There exist some problems that span all five of the areas discussed above. Error handling and recovery become quite complex when one has to decide which of the five areas is responsible for detection, user notification, retry and cleanup. This problem becomes even more complicated in a network that is based on loosely coupled third generation operating systems which do not offer the same system services and file management schemes. Revision control of worker node software is also very difficult to handle in a consistent and proper manner in a heterogeneous network environment.

Homogeneous network environments may not appear to suffer from the same problems as a highly diversified heterogeneous network but there are lessons to be learned that may well save time and effort in the future. Whenever a server node, such as the output node, is introduced into a network it behooves the implementor to examine the five areas listed above and decide in advance upon well defined interfaces from the user to the output device. A server node is of value to the network only if it can be easily and reliably used from any worker node, which means that time and effort will have to be invested in each of the five different areas.

FIGURE 1.1



- > **SLTS1,SLTS2,SLTS3,SLTS4** are worker VAX 11/780s
- > **SLGW1** is the gateway VAX 11/780
- > **SLOP1** is the output node VAX 11/780
- > **SLAM1** is the applied mechanics VAX 11/730
- > **CRAY C** and **CRAY D** are CRAY 1Ss
- > **CFS** is two IBM 43141s with a cartridge store device

Figure 1.2

OUTPUT NODE

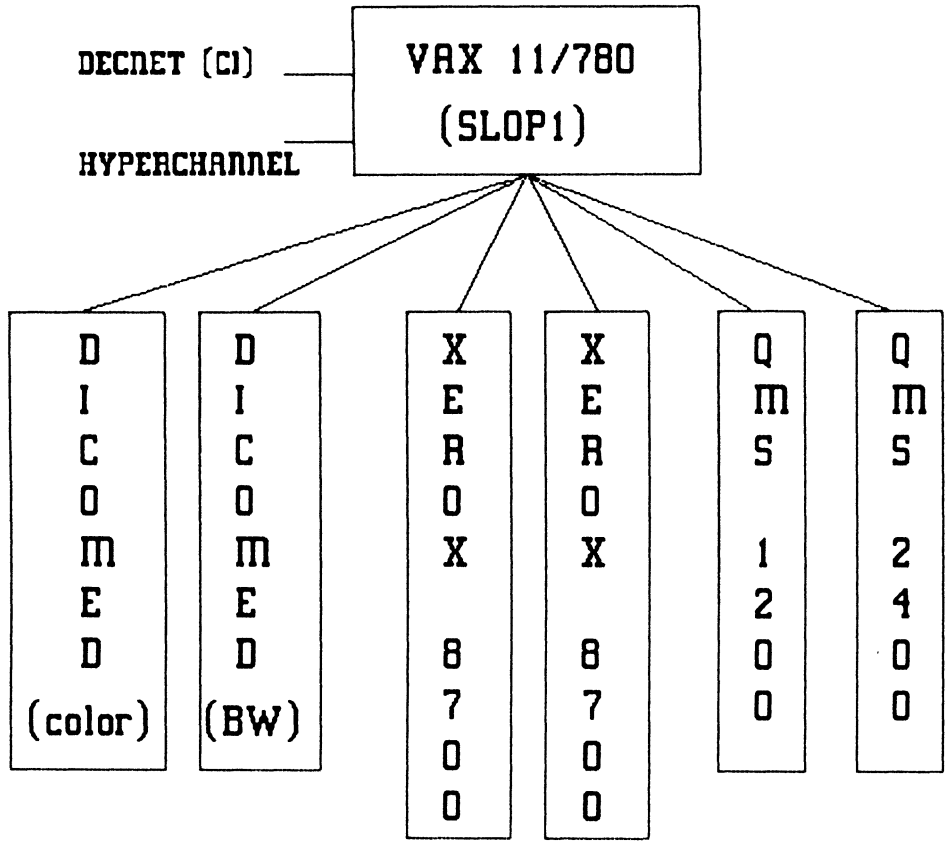


Figure 1.3

XEROX - VAX INTERFACE

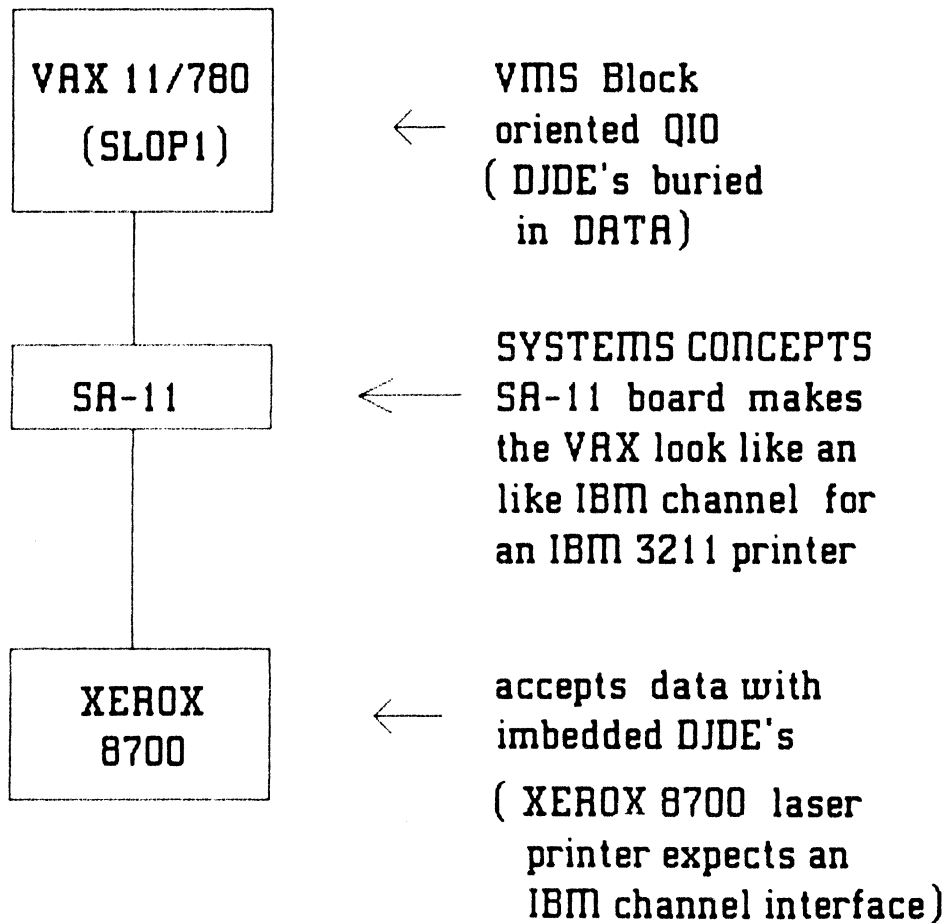


Figure 1.4

VMS LPRINT

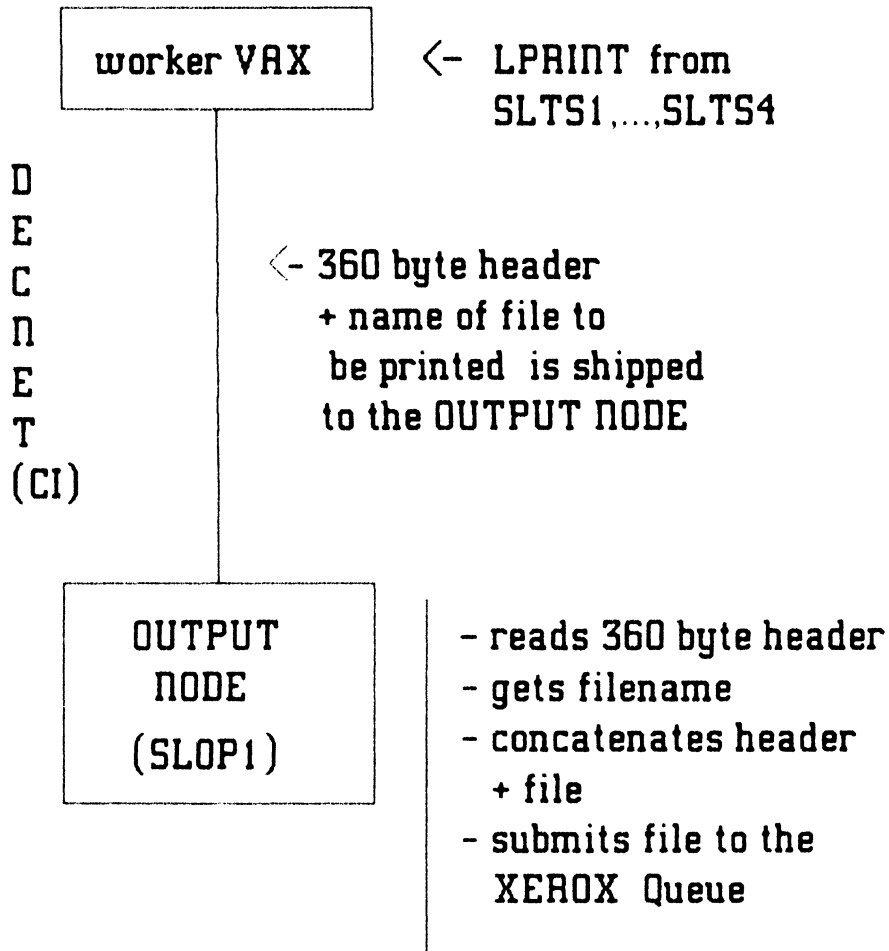
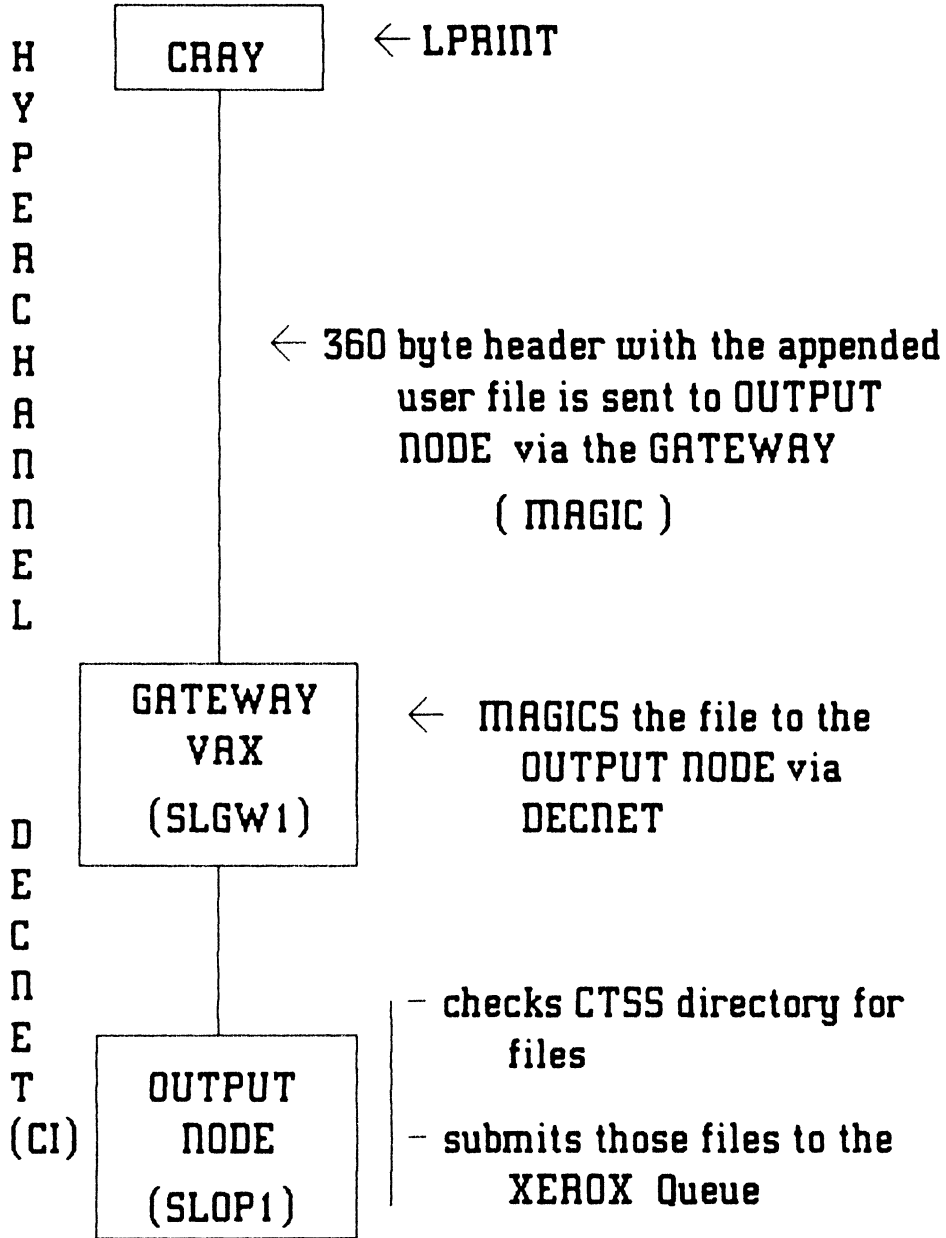


Figure 1.5

CTSS LPRINT



INTERACTIVE FORMAT CONVERSION SYSTEM (IFCS)

Steven J. Kempler
NASA/Goddard Space Flight Center
Laboratory of Extraterrestrial Physics
Greenbelt, Maryland

ABSTRACT

The Interactive Format Conversion System (IFCS) is a package designed to facilitate the transfer of data between heterogeneous computers. The system has the generalized capability of: 1) accepting input data from a number of devices (disk, tape, data line); 2) performing useful data conversions, and; 3) producing output on a variety of devices. The structure of the data conversion subsystem simulates a subset of the presentation layer in a network communications link by converting input data into an internal machine independent format and then converting the internal data to the output format. This conversion subsystem is derived by the inputs of the application. The Transportable Applications Executive (TAE) is used to provide a consistent user interface and tie the various subsystems together.

1.0 INTRODUCTION

1.1 BACKGROUND

The transfer of space-derived data between computers, both heterogeneous and homogeneous, have become more common and increasingly desirable. The International Organization of Standardization (ISO) has established a seven layer model for networking. (Tanenbaum describes this in detail in his book *Computer Networks*.) See Figure 1. The five lower layers of the networking model are being addressed by various organizations, including the National Bureau of Standards (NBS), International Standards Organization (ISO), and the IEEE. Currently, the sixth layer, the Presentation layer, is being developed on a case-by-case basis many times over for a variety of space-related data. This layer specifically performs transformations on data, such as text compression, format conversions, encryption, etc.

1.2 CURRENT EFFORT

Problem: Currently, Presentation layer, specifically format conversion software, is being developed on a case-by-case basis many times over for a variety of space-related science data, leading to much duplication of effort and code.

Solution: The Interactive Format Conversion System (IFCS) is a subset of Presentation Layer Software. It generalizes format converting by interactively generating software that transforms data from and to the desired machine formats. The generated code is transportable and can be generated for a particular application and used on a number of different

computers. Such a system solves the problem for the users of space-derived data that has not been attacked in any general sense up to now.

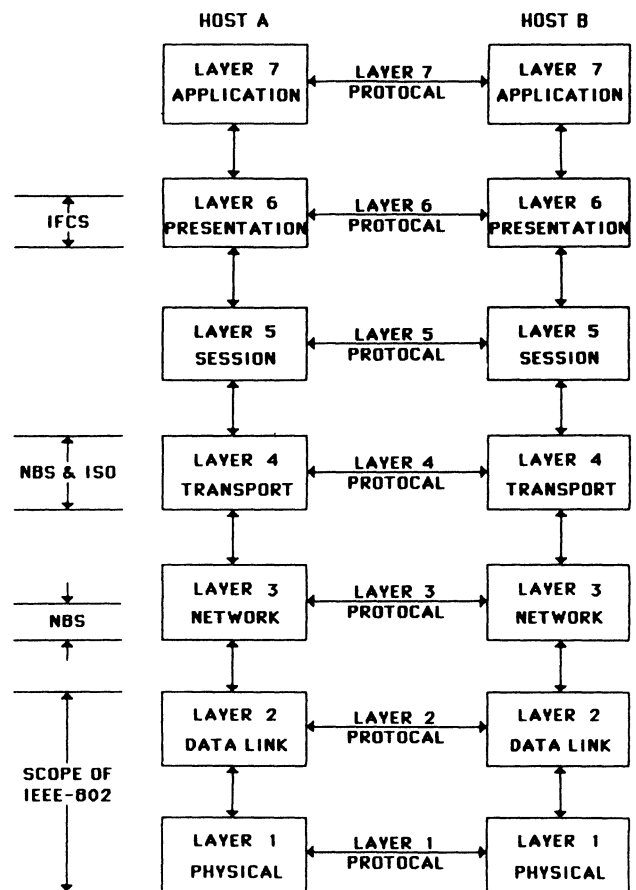


FIGURE 1 - SEVEN LAYER NETWORKING MODEL

User inputs to the system include global variables which are defined and used throughout the IFCS session. After IFCS is launched (Figure 2), the CVTGEN executes by receiving communications (inputs) from the user: data definition files, file and machine names. CVTGEN generates a stand alone conversion routine, that may be linked to a user supplied program or link to the General Format Conversion Utility (CVTLINK). To execute the General Format Conversion Utility (GFCUTIL), the data input devices (containing the data to be converted) and data output device, and device attributes must be communicated (input) to the utility, as well as the number of records to convert. The result is converted data.

IFCS presently resides on the Laboratory of Extraterrestrial Physics (LEP) (Code 690) VAX 11/780, in Building 2 at Goddard Space Flight Center. This computer supports a wide range of scientific space-related data and the analysis of this data. Included are data from Mariner, Voyager, ISEE and IMP satellites. In addition, the LEP VAX supports many data analysis packages and numerical libraries. Therefore, the need has grown to transfer data on the LEP VAX to other computers as well as visa versa, to support the needs of scientific data analysis in familiar environments. This need is not limited by any means.

USER INPUTS FOR IFCS

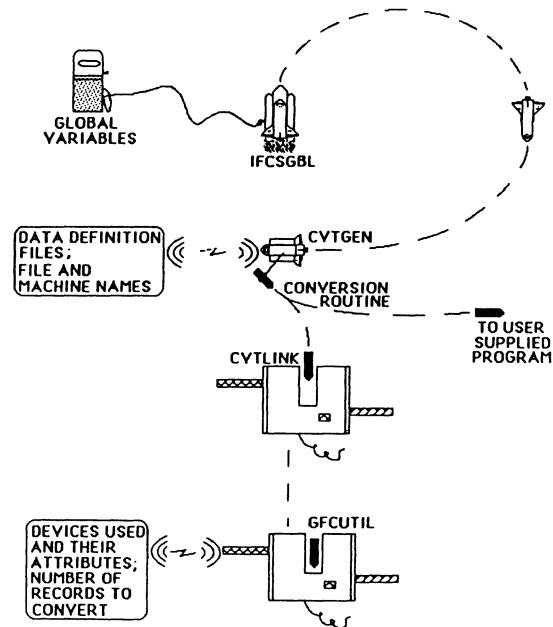


FIGURE 2

1.3 DESIGN CONSIDERATIONS

The primary objective in designing and implementing IFCS was to develop a system that allows characteristics of the source and target data streams, along with identifier information, to be easily specified interactively. IFCS utilizes this input to produce transportable computer code that maintains the semantics of the data as they are transformed from one computer to another. In addition, IFCS was developed to be friendly and flexible. The user need not supply more information than is absolutely necessary for the function to be performed. Also, interactive input requirements must be unambiguous and check for invalid inputs. IFCS is made flexible enough to handle a wide spectrum of possible data inputs. Finally, the system was designed to isolate host specific code so that it may be transported with minimal change.

2.0 CAPABILITIES

2.1 FUNCTIONAL CAPABILITIES

The first step of IFCS is the conversion generator program. Required inputs include input and output record definition and optional namelist file names, machine format associated with the data and the name of the output conversion routine. The conversion generator analyzes the input and output record definitions and builds a file (a subroutine) that contains all the actual field conversion routines in exact order as defined by the record definition. The library containing these lower level conversion routines is the heart of

IFCS. Each routine performs a different function. (i.e. convert DEC R⁴, convert to IBM R⁸, etc. See PERFORMANCE CAPABILITIES for a further discussion on converting.) The conversion generator creates the routine which in turn accesses these pre-existing routines when run. The second step in IFCS is to compile and link the conversion routine. It may be linked to a user developed program or it may utilize the General Format Conversion Utility (GFCU). This utility will perform all general input of data, convert the data using the conversion routine and output the results. Finally, GFCU or the user application program is executed to perform the format conversion.

In addition, IFCS includes a Machine Definition program for when it becomes desirable to add new machines to the system (IFCS presently supports DEC VAX, IBM and SIGMA 9 computers). New conversion library routines will also need to be implemented.

IFCS utilizes Transportable Applications Executive (TAE) to enhance its functional capabilities as well as fulfill design objectives. Inputs are entered using TAE standards. This provides ease for the experienced user and support for the less experienced user. Menus and help files provide information for the first time user. In addition, IFCS can be easily transported to any installation that maintains TAE.

2.2 PERFORMANCE CAPABILITIES

IFCS has several important performance capabilities. A primary capability is its use of namelists. This provides the user with the ability to convert only

certain fields of data from the input record. GFCU is capable of inputting data from up to three input sources, converting the data and outputting to a single sink. Also, GFCU can input and output to tape or disk. Most important is that IFCS uses an intermediate data format when converting data. That is, every field transformed is actually converted twice (i.e. IBM -> intermediate form -> VAX). This design was implemented so that when additional machines are added, source code will increase at a much smaller rate (Figure 3). (All new routines will convert to or from intermediate format.) Finally, IFCS is capable of checking for precision loss and overflow when it is desirable to convert values to utilize less space (i.e. R*4 to I*2). A value, of the users choice, representing BAD data is inserted. Also, the user may choose a tolerance level in which IFCS will stop if BAD must be inserted too many times.

3.0 OPERATION

IFCS is able to perform six basic operations through the use of the TAE menus (Figure 4). IFCSGBL is a global procedure that allows the user to define certain variables. CVTGEN, CVTLINK and GFCUTIL are the three steps for developing and executing a data conversion program (Figure 5). The procedure, IFCS, combines the previous three PDF's in one procedure.

NUMBER OF CONVERSION ROUTINES REQUIRED AS A FUNCTION OF THE NUMBER OF MACHINES IN IFCS

CONVERTING ONE DATA TYPE TO THE SAME DATA TYPE:

NUMBER OF MACHINES (M)	ONE ROUTINE DOES CONVERSION (M*(M-1))	WITH IFCS INTERMEDIATE FORMAT (M*2)
2	2 (1 IN EACH DIRECTION)	4 (2 IN EACH DIRECTION)
3	6	6
4	12	8
5	20	10

CONVERTING ONE DATA TYPE TO ANY OF 4 DATA TYPES:

M	16 * M * (M-1)	4 * M * 2
2	32 (16 IN EACH DIRECTION)	16 (8 IN EACH DIRECTION)
3	96	24
4	192	32
5	320	40

ORDINARILLY, IT TAKES 32 ROUTINES TO BE ABLE TO CONVERT ANY 6 DATA TYPES (R*4, R*8, R*16, I*2, I*4, C*8) FROM ONE MACHINE TO ANY OF THOSE DATA TYPES ON ONE OTHER MACHINE.

USING IFCS INTERMEDIATE FORMAT IT TAKES ONLY 16.

FIGURE 3

ROOT", Library "DISK\$USER3:(YSJFY.PLS.DEMO)"

INTERACTIVE FORMAT CONVERSION SYSTEM

- 1) TO ALTER IFCS GLOBAL VARIABLES (IFCSGBL)
- 2) TO GENERATE AN IFCS CONVERSION ROUTINE (CVTGEN)
- 3) TO LINK THE CONVERSION ROUTINE TO THE GENERAL FORMAT CONVERSION UTILITY (CVTLINK)
- 4) TO EXECUTE THE GENERAL FORMAT CONVERSION UTILITY (GFCUTIL)
- 5) TO EXECUTE PROC CVTGEN, CVTLINK AND GFCUTIL (IFCS)
- 6) TO IMPLEMENT A NEW MACHINE'S ATTRIBUTES (MACHDEF)

Enter: selection number, HELP, BACK, TOP, MENU, COMMAND, or LOGOFF. ?

TAE MAIN MENU FOR IFCS - FIGURE 4

Finally, MACHDEF allows the programmer to implement additional machines.

3.1 IFCSGBL

Two variables set in this global are used throughout IFCS.

- the name of the IFCS generated conversion routine to be linked and executed.
- the number of input sources (up to three).

3.2 CVTGEN

CVTGEN represents the first step if IFCS (Figure 6). Using information recieved from user created internal files and user input, CVTGEN creates a FORTRAN routine that, when executed, will receive data according to the specified format, convert the data to the desired machine and output only the

INTERACTIVE FORMAT CONVERSION SYSTEM (IFCS)

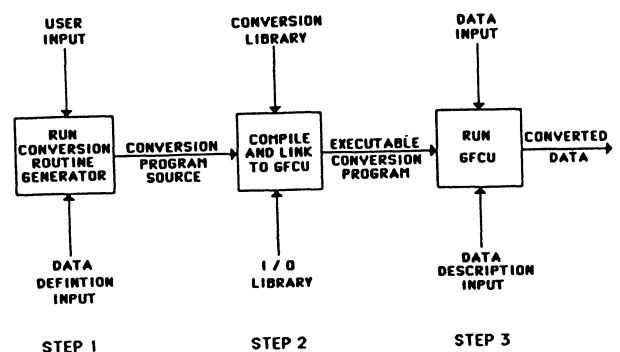


FIGURE 5

CONVERSION ROUTINE GENERATOR

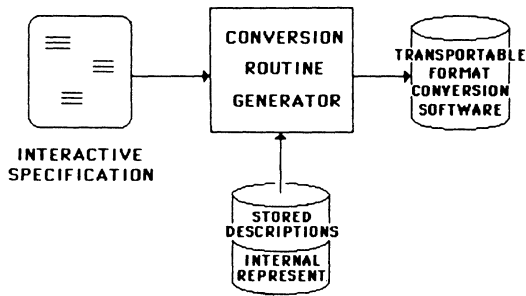


FIGURE 6

GENERAL FORMAT CONVERSION UTILITY (GFCU)

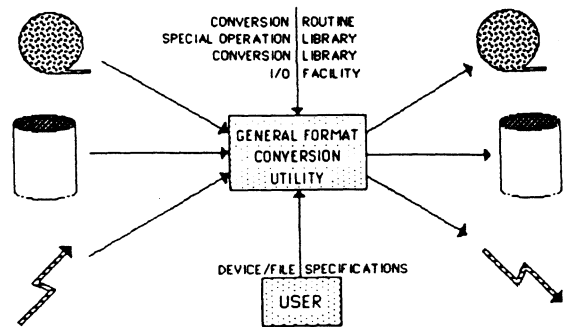


FIGURE 7

data fields within the record that are of interest. The conversion routine is made up of a series of calls to pre-existing lower level routines. For each data field, a lower level routine is accessed to perform the correct bit manipulations to move that field into and out of the intermediate format (as described earlier).

3.2.1 Inputs

The required internal files include:

- the record definition files which contain the exact field format of the data to be input (one is required for each input source).
- the record definition file which contains the exact field format of the data to be output.
- the optional namelist files which contain a name that corresponds to each field in the record definition files (one for each input source) of the data to be input.
- the optional namelist file which contains only the names of the fields that are to be converted and output. If namelist files are not specified or the input data and output data namelists are exact, then all fields are converted.

The user inputs include:

- the name of the file that contains the record definitions (one for each of up to three input sources).
- the name of the file that contains the input namelists (up to three). This is optional.
- the name of each machine which the input data was generated on (up to three).
- the name of the file that contains the record definition for the output.
- the name of the file that contains the output namelist. This is optional.
- the name of the machine which the output data is generated for.
- the name to be given to the file that will contain the newly generated conversion routine.

3.2.2 Outputs

The CVTGEN output is the reusable conversion routine created to user specification.

3.3 CVTLINK

At this point the conversion routine may be utilized with a user application or it may be linked to the General Format Conversion Utility (GFCU). This procedure compiles and links a conversion routine to the GFCU. The name of the object and load modules will be the same as the source file, which are defined in IFCSGBL. No interactive inputs are required for this step.

3.4 GFCUTIL

The third step, GFCUTIL, actually converts the specified data (Figure 7). This procedure, using the tape I/O library, provides a means for reading tapes created on and writing tapes for other machines in any format, as well as reading and writing to disk. Generally, GFCU acquires the input data, performs the specified conversions, and outputs the results.

3.3.1 Inputs

To operate GFCU, the following inputs are requested:

- the type of device in which the input is received from (TAPE or DISK). One for each input source.
- the type of device in which the output is to be sent (TAPE or DISK).

For each TAPE used, the following must also be provided:

- the name of the tape drive.
- the tape label if the tape is labeled.
- the machine format of the tape (presently, IBM, VAX or SI9).
- the record type of the tape file.
- the logical record size.

- the tape block size.
- the file name of the data (for input data tapes and output data tapes), or the file number of the data (for input data tapes).
- the starting record number in the file of the tape where data conversion is to commence (for input data tapes).

For each DISK file used, the following must also be provided.

- the organization of the disk.
- the disk access method.
- the record size of the disk.
- the name of the disk file.
- the starting record in the file where data conversion is to commence (for input data).

In addition, these parameters are also required:

- the number of data records to be converted.
- whether GFCUTIL is to be executed in interactive mode or batch mode.
- a BAD value to be inserted when precision is lost in converting to real numbers. (Default = -99.9)
- a BAD value to be inserted when precision is lost in converting to integer numbers. (Default = -9999)
- an integer representing the number of BAD values that may be inserted before IFCS will stop processing data.

3.3.2 Outputs

The output generated is a file containing the desired converted data.

3.5 IFCS

The purpose of this operation is to combine the three steps of IFCS into one procedure. This provides much convenience when it is desired to create, link and execute IFCS all at once.

3.6 MACHDEF

This operation is primarily used by the IFCS manager. When a new machine is implemented into IFCS, the manager must: create lower level conversion algorithms that convert data to and from the intermediate format; provide for any tape formatting dissimilarities that the new machine has to the existing machines (in the tape I/O library) and; execute MACHDEF. MACHDEF is a software maintenance program that implements the characteristics of any newly added machine to IFCS. As mentioned, only the IBM, VAX and SIGMA 9 are presently supported. This software receives the machine characteristics and places them in a machine definition file.

3.6.1 Inputs

The inputs include:

- the name that identifies the machine whose attributes are being entered (ex. VAX).
- a two character machine identifier (ex. VX for VAX).
- the number of bits per byte of this machine.
- the default Hollerith code to be assigned to this machine (ASCII or EBCDIC).
- up to 20, two character data format identifiers (ex. R4 for REAL * 4).
- up to 20, integers describing the length in-bytes of each data format identifier entered.
- up to 20, one character data type identifier for each data format identifier entered (I, F, H or Z).

3.6.2 Outputs

The output of this process is the addition of the new machine specifications in the machine definition table.

4.0 FUTURE CONSIDERATIONS

Plans exist for IFCS on all fronts. Enhancements to the system include adding a provision for special data types (for example, spacecraft telemetry). Enhancements for I/O include implementing electronic communication into GFCU. Adding other machines to the system is another immediate consideration, as well as implementing IFCS on other machines. From an operational point of view, optimizing the speed of IFCS is being addressed.

ACKNOWLEDGEMENTS

I wish to acknowledge William Mish, Thurston Carleton and Jack Yambor for their design and implementation of IFCS.

REFERENCES

- Carlson, Patricia A., et al., Primer for the Transportable Applications Executive, NASA/GSFC, January, 1984.
- Century Computing, Inc., Application Programmer's Reference Manual for the Transportable Applications Executive, March, 1984.
- Century Computing, Inc., User's Reference Manual for the Transportable Applications Executive, March, 1984.
- Computing Surveys, Vol. 13, No. 4, December, 1981.
- Tanenbaum, Andrew S., Computer Networks, Prentice-Hall, 1981.

VAX COMMUNICATION CONTROLLERS

Roger Russ
Advanced Computer Communications
Santa Barbara, California

ABSTRACT

This paper discusses design problems encountered in developing microprocessor-based front-end communication controllers for the VAX product family. As network data rates increase, these problems become more acute. Two example connections to networks with high bit rates are described, and a design that solves potential data-overflow problems is discussed. Performance graphs show the correlations between VAX load and network throughput.

INTRODUCTION

The essential elements of a point-to-point link between two VAX processors across a network are the two host systems and the network. As shown in figure 1, each host system consists of a VAX and its UNIBUS. Also shown is a front-end communication processor making the connection to a network (any of various physical transmission media including public networks, broadband or baseband networks, local area networks, or other commonly known networks). Our discussion focuses on the design problems that arise when the individual components are integrated into a typical-design front-end communication processor; optimal component performance typically cannot be maintained during worst-case front-end loading. We describe two example network connections (one to the Ethernet and one to an RS-422 high-speed serial (HSIO) link at T1 rates) and our unconventional front-end design solution to the network-data overflow problem and other limitations imposed by typical front-end designs.

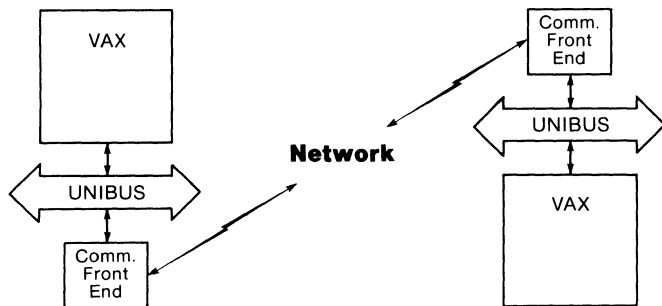


Figure 1. Typical Network Configuration

DATA TRANSMISSION RATES (BANDWIDTHS)

The key parameters influencing front-end design are data transmission rates: network bandwidth, microprocessor bandwidth, and UNIBUS bandwidth. Ethernet bandwidth is 10 million bits/sec (Mbps), about six times the T1 rate of 1.544 Mbps.

The bandwidth of the microprocessor bus in the front-end is particularly important. The 16-bit 68000 microprocessor unit (MPU) operating at 12 megahertz (MHz) is a commonly used microprocessor that has excellent performance characteristics. The MPU requires a minimum of four clocks to complete a transfer on the microprocessor bus, that is, 333 nanoseconds (nsec); the consequent bandwidth is 3 million words/second (*Motorola Microprocessors Data Manual, 1981*).

UNIBUS bandwidth is another important factor in the data transmission rate. We use VAX-11/780 UNIBUS values in this discussion (see the *VAX-11/780 Architecture Handbook, 1977*). The data rate depends on which data path is used through the UNIBUS adapter. The direct data-path rate is 425 thousand words/second (425K) for a write, and 316K for a read. The buffered data-path rate is 695K for both reads and writes. These rates are summarized in table 1, where they have been converted to kilobytes/second for ease of comparison.

Table 1. Data Path Bandwidths

DATA PATH	BANDWIDTH
T1	193 Kbps
Ethernet	1250 Kbps
68000 bus	6000 Kbps
UNIBUS:	
direct data path	850 Kbps (write) 632 Kbps (read)
buffered data path	1390 Kbps

TYPICAL FRONT-END DESIGN

The design of a typical front end is shown in figure 2. The key element is the microprocessor and microprocessor bus. The other elements either support program execution or facilitate data flow. Program execution elements include RAM, EPROM and counter-timer. Data flow elements include host interface, network interface, and DMA controller.

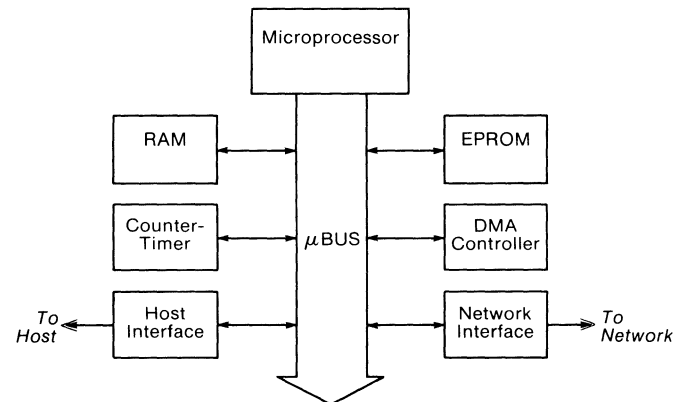


Figure 2. Typical Front-End Design

Data Transfer Rates

When we examine data transfer within this structure, and survey the rates of various elements, particularly under worst-case high-traffic conditions, several design problems emerge.

MPU (12 MHz) execution times are listed in table 2. These numbers are valid for instructions without additional memory cycles. Execution times range from 333 nsec (0 wait states) for execution from fast-access EPROM to 667 nsec (4 wait states) for execution from a large, slower-access dynamic RAM.

Table 2. MPU (12MHz) Execution Times

Wait States	Time (nsec)
0	333
1	417
2	500
3	583
4	667

In comparison, DMA transfer times for the currently available 68450 DMA Controller (DMAC) are given in table 3. This four-channel device is available with 8 MHz or 10 MHz clock. Table 3 lists the times for a single transfer into a device or memory with no wait states. These times are similar to those for the MPU.

Table 3. DMAC Transfer Times (no wait states)

DMAC Clock (MHz)	Transfer Time (nsec)
8	500
10	400

However, brief analysis of a worse-case high-traffic situation points out a problem. With all four DMAC channels active and with one DMA transfer interleaved with one MPU cycle, four cycles or 4.67 microseconds ($4.67 \mu\text{sec} = 4 \times (500 + 667 \text{ nsec})$) would elapse before the lowest priority channel could transfer a word; this word would be transferred an order of magnitude slower than shown in table 3.

Next we consider a realistic UNIBUS access time for the front end in figure 2. Because the 256-kilobyte UNIBUS comprises only 1.6% of the addresses available on a 68000 bus, a likely method of interfacing with the host is to memory-map the UNIBUS into a segment of the MPU and 68000 bus's 16-megabyte address range. To perform a memory-mapped cycle, the DMA controller puts a UNIBUS-mapped address on the 68000 bus. This address is recognized by the host-interface logic which then requests a UNIBUS cycle (NPR). The data transfer (requiring about 0.6 μsec) occurs when the bus is granted (NPG). However, because bus acquisition depends on the hardware configuration of the UNIBUS, a worst-case high-traffic time cannot be determined. Higher priority DMA devices, particularly those doing burst transfers, might use the bus repeatedly or for long periods. With a few such higher priority DMA devices, bus grant delays longer than 40 to 50 μsec might occur.

The final element in this front-end design is the network interface. We consider two examples of network hardware: the AMD 7990 Local Area Network Controller for Ethernet (LANCE) and a high-speed RS-422 serial interface using the Rockwell 68561 Multi-Protocol Communications Controller (MPCC).

Figure 3 diagrams how the LANCE is attached to the 68000 bus. Because the LANCE is not 68000 compatible, bus conversion logic (providing control timing and data and address multiplexing, not shown) is necessary. One example converter is a state machine implemented in PLAs that has a worst-case time of 900 nsec. The LANCE has an internal 48-byte FIFO; at the Ethernet data rate (10 Mbps), it overflows in 38.4 μsec unless data are removed.

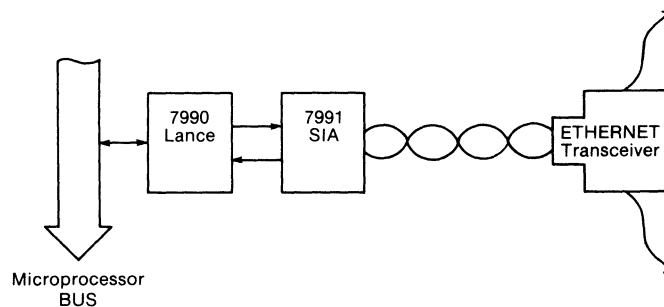


Figure 3. LANCE Attachment

Figure 4 diagrams the HSIO interface. The MPCC is 68000 compatible; its worst-case access time is 440 nsec. The MPCC has an internal 8-byte FIFO; at the T1 data rate (1.544 Mbps), it overflows in 31 μsec unless data are removed.

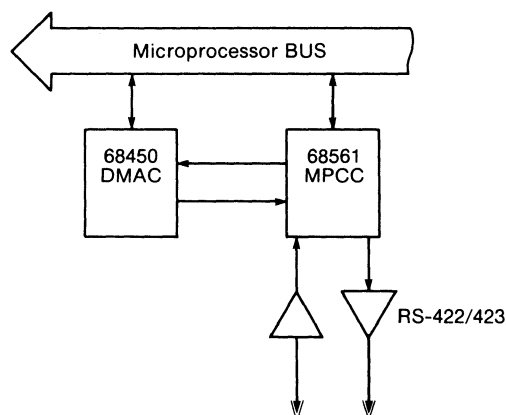


Figure 4. MPCC Attachment

Handling Network Data Overflows

How does the front-end operate when the network interface is about to overflow and the host interface owns the microprocessor bus for an indefinite period of time? The simplest solution is simply to ignore it and let the network interface overflow and post an interrupt to the MPU. A retransmission would then be initiated. This is not a very elegant solution, and throughput is seriously reduced if retransmissions occur frequently. A variety of other mechanisms also could be used with this type of front-end design, for example isolation of the host interface, a dual port RAM for the network interface, or an abort mechanism for the host interface. Alternatively, a substantially different front-end design could be implemented, as described in the following sections.

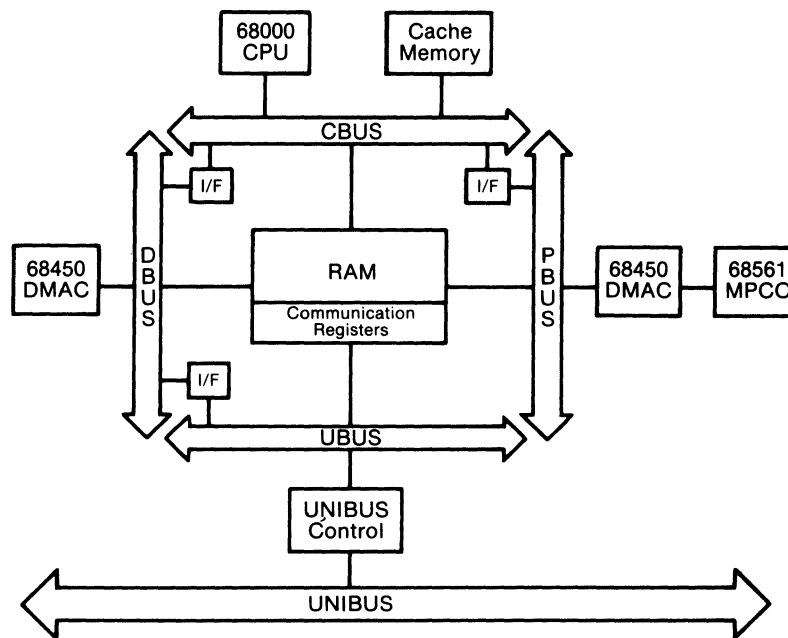


Figure 5. ACP 6000 Architecture

A UNIQUE DESIGN SOLUTION

A design that elegantly handles the potential conflicts discussed in the preceding paragraphs is shown in figure 5. The depicted architecture constitutes the core of the ACP 6000 communication front-end processor. Its central feature is a four-port RAM; each port accessed by a segment of the 68000 bus. The segments, named the CBUS, DBUS, PBUS and UBUS attach or detach in accord with the activity taking place.

By supporting only one DMA device on each bus segment, this four-bus architecture guarantees that each DMA device has access to its local bus, virtually eliminating bus contention. The functional result is that generally RAM arbitration occurs rather than bus arbitration and contention, and because RAM arbitration is an order of magnitude faster, overall throughput is enhanced. An additional advantage is that different operations (described below) can occur simultaneously on separate buses.

Typical Front-End Operations

Typical ACP 6000 operations include microprocessor instruction execution, host communication, DMA transfers with the host, and network transfers. Microprocessor instruction execution involves only the MPU, the CBUS and local cache memory. Host communication occurs via the Communication Registers (actually located in RAM) and the hard-wired, dual-port UNIBUS Control and Status Register (UCSR). When the host uses the Communication Registers, only RAM, the UBUS and UNIBUS-control logic are used. DMA transfers with the host are examples of two buses connecting to perform a transfer. During such transfers, the DBUS DMAC transfers data into its internal data register using only the DBUS, UBUS and UNIBUS-control logic. During network transfers, the PBUS DMAC either reads a byte from RAM and writes it into the MPCC's FIFO, or reads from the FIFO and writes to RAM.

The following narrative traces the sequence of events by which the ACP 6000 transfers a packet of data from the host to the network. First, the host initializes the Communication Registers with information about the data packet: the size of the packet's data block and the block's UNIBUS address, plus other packet-related information. As the last step of the initialization, the host writes a bit in the UCSR that causes an interrupt to the MPU.

The MPU then reads the packet information in the Communication Registers and initializes the DBUS DMAC with the UNIBUS information and local (ACP 6000) RAM buffer addresses. The MPU's final step in taking the information from the host is to set a start bit in the DMAC, causing the DMAC to transfer the data from the UNIBUS into local RAM. When this data block is transferred, the DMAC sends a completion interrupt to the MPU, which in turn writes ending status in the Communication Registers and interrupts the host by writing a bit in the UCSR. This interrupt informs the host that the transfer is complete. At this point, the ACP 6000 might have further processing to perform on the data packet such as further protocol processing, buffer management and copying. When the packet is ready, the MPU initializes the PBUS DMAC, which then transfers the packet from RAM to the MPCC. The MPCC then transmits the packet to the network.

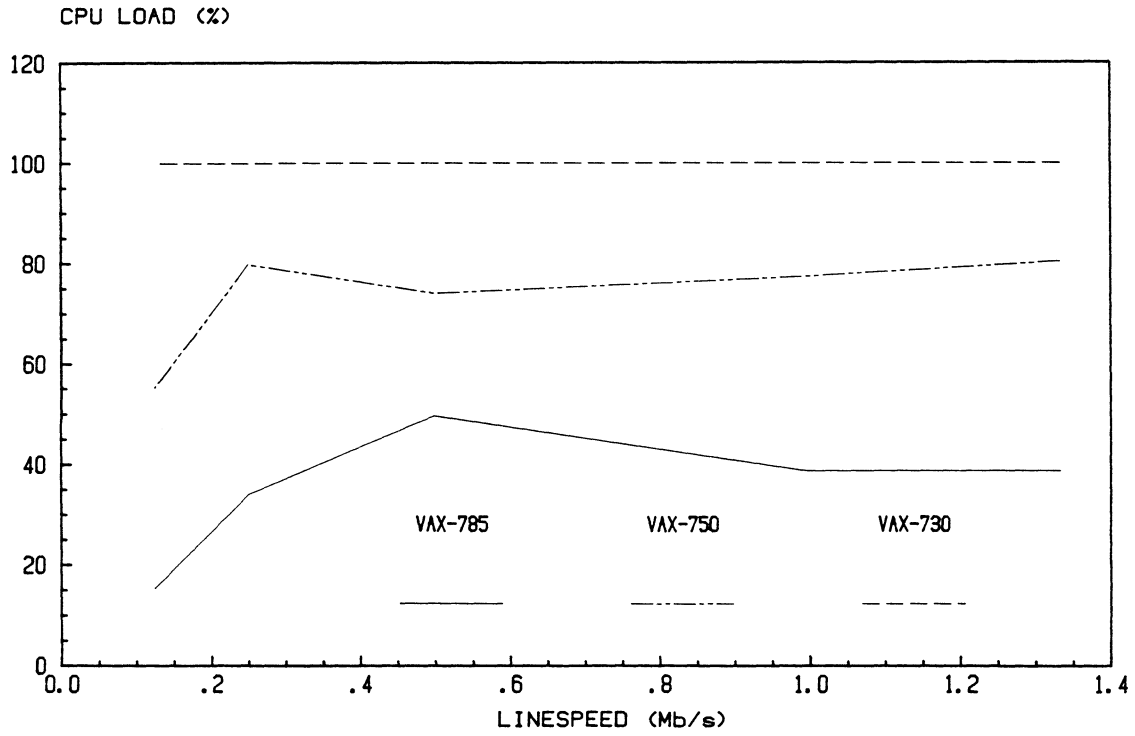
VAX System Performance

When considering the use of a front end, it is important to see how its data rates affect host performance. Graphs 1 through 8 show the relationships between packet size, line speed, aggregate throughput, and VAX CPU loading using an ACP 6000 front end (data collected and graphed by Peggy Cornell of ACC). Each graph depicts the performance of three VAX hosts: the 11/730, 11/750 and 11/785. The ACP 6000 was doing complete HDLC protocol processing; the host was creating data to be transferred, requesting transfer by the ACP 6000, receiving and checking returned data, and logging a running total of (correct) received data.

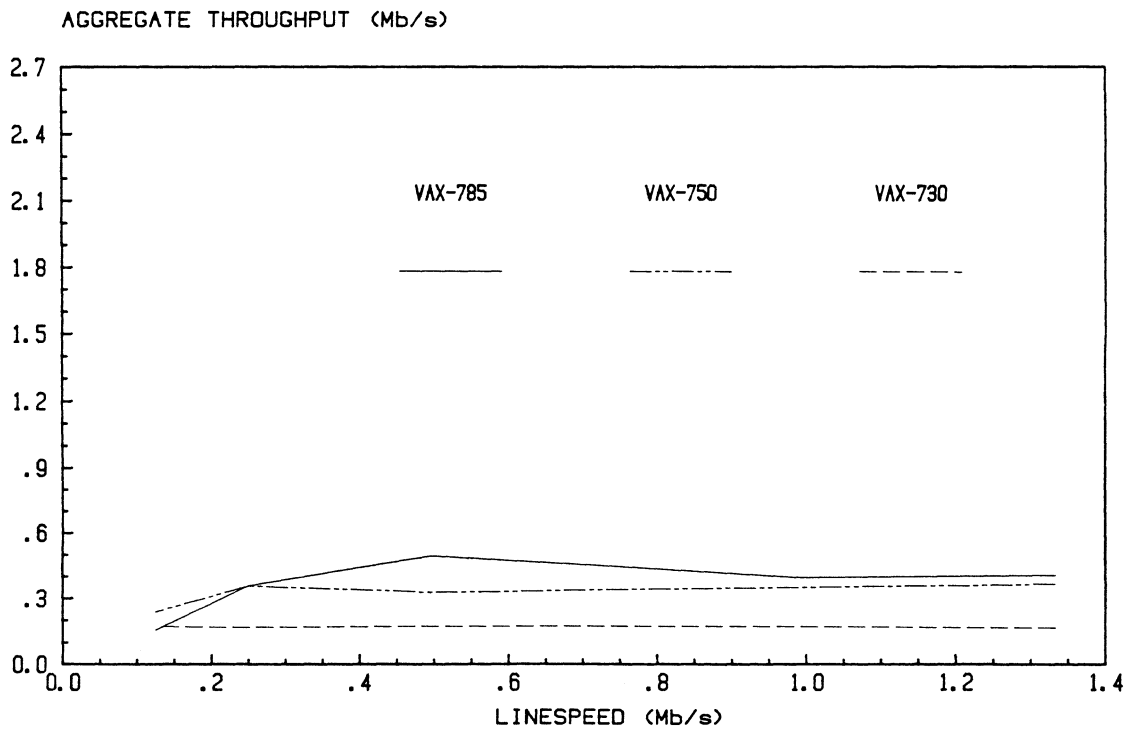
Four sets of graphs are provided, one set each for packets of 256 bytes, 1024 bytes, 2048 bytes, and 4096 bytes. Each set of graphs plots aggregate throughput (input + output) versus linespeed, and VAX CPU load versus linespeed.

Several general comments can be made about these graphs. First, it can be seen that the 730 reaches saturation much faster than the other processors. Interpacket processing on the VAX is the limiting factor, and the 730 reaches a limit of about 200 packets per second. As packet sizes increase this factor becomes secondary, and overall throughput becomes larger. With 2048-byte packets, both the 750 and 785 achieve throughput that approximates the linespeed up to 1 Mbps. With 4096-byte packets, all processors achieve high data throughputs, although the 730 does saturate above 1 Mbps.

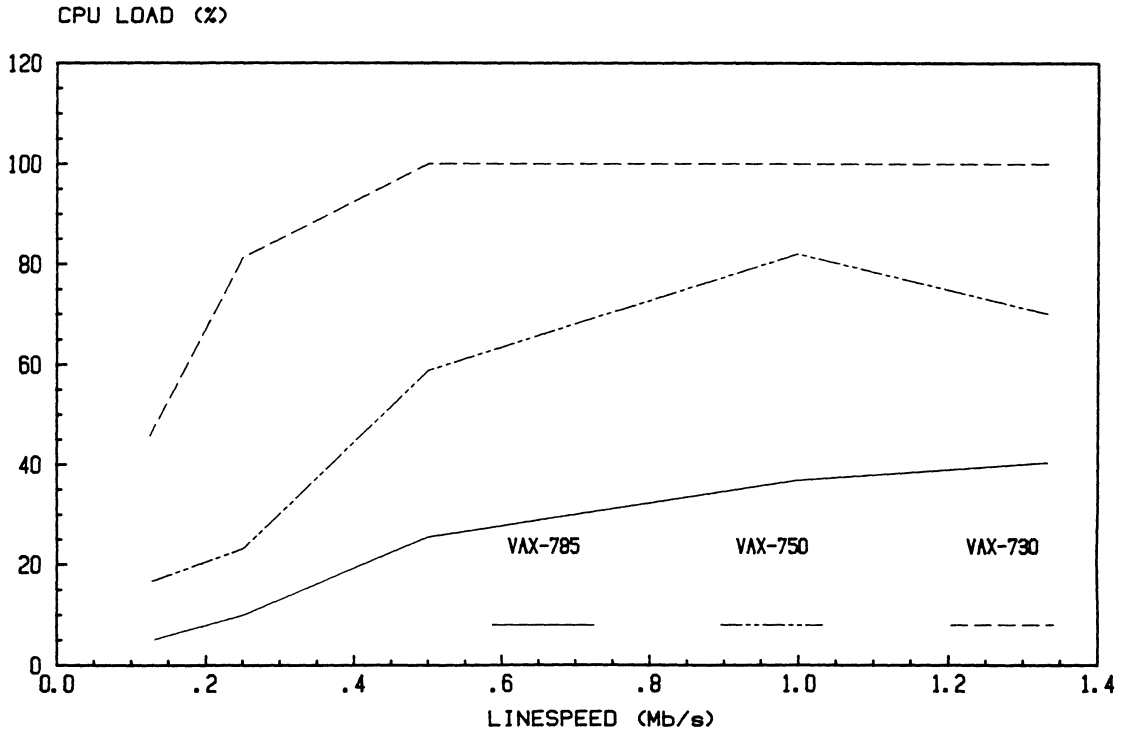
Graph 1. 256-Byte Buffers: VAX Loading vs Linespeed



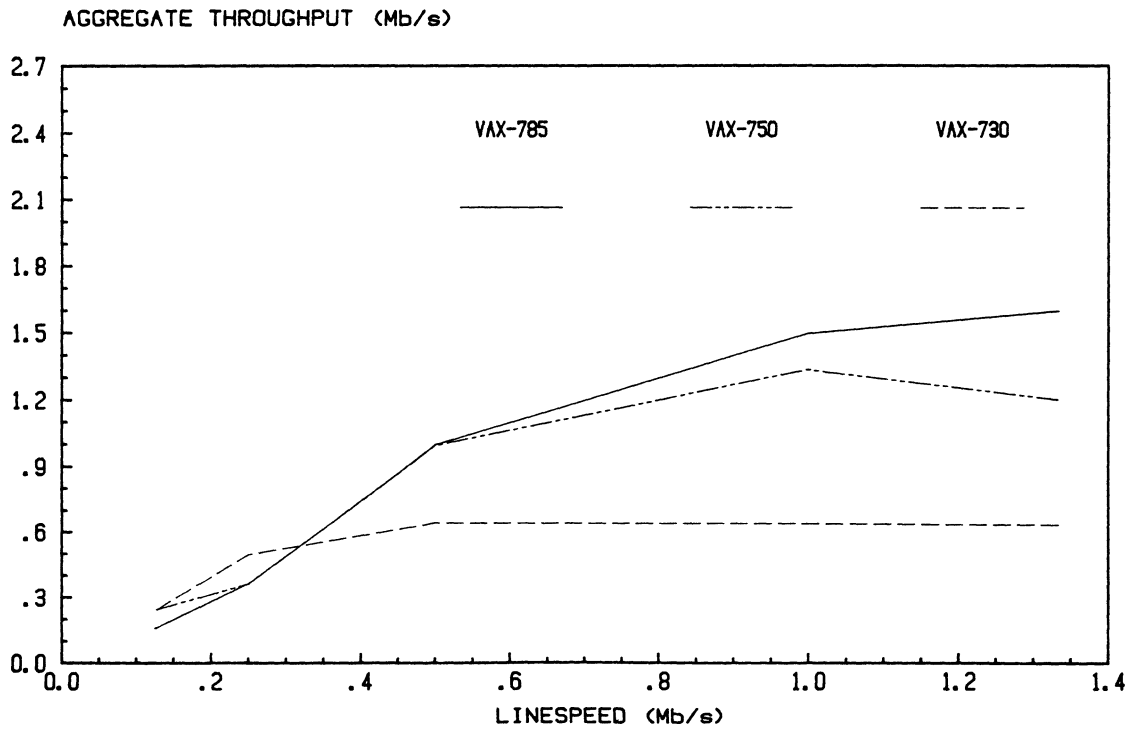
Graph 2. 256-Byte Buffers: Throughput vs Linespeed



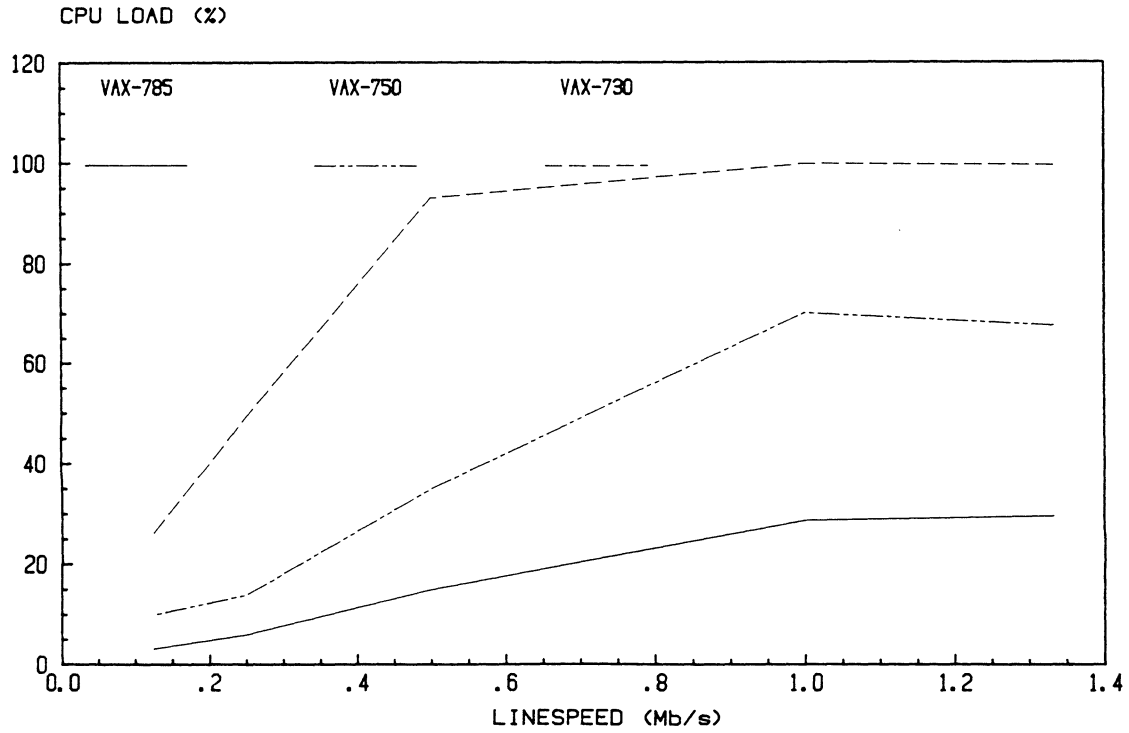
Graph 3. 1024-Byte Buffers: VAX Loading vs Linespeed



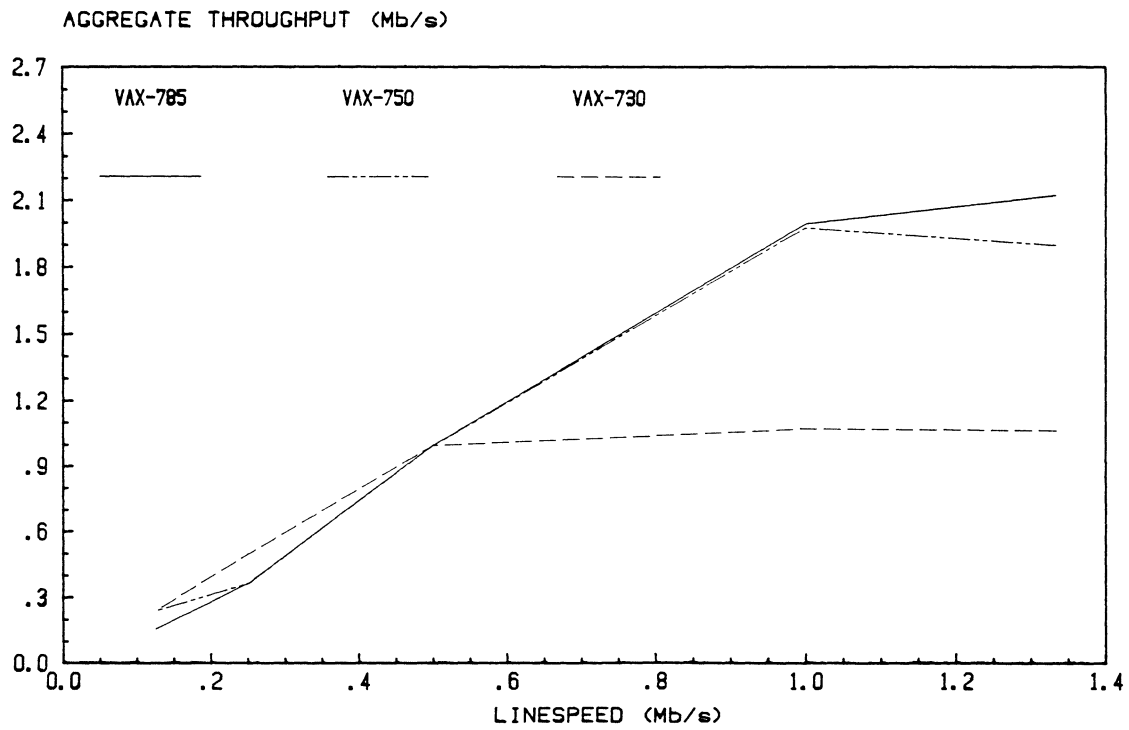
Graph 4. 1024-Byte Buffers: Throughput vs Linespeed



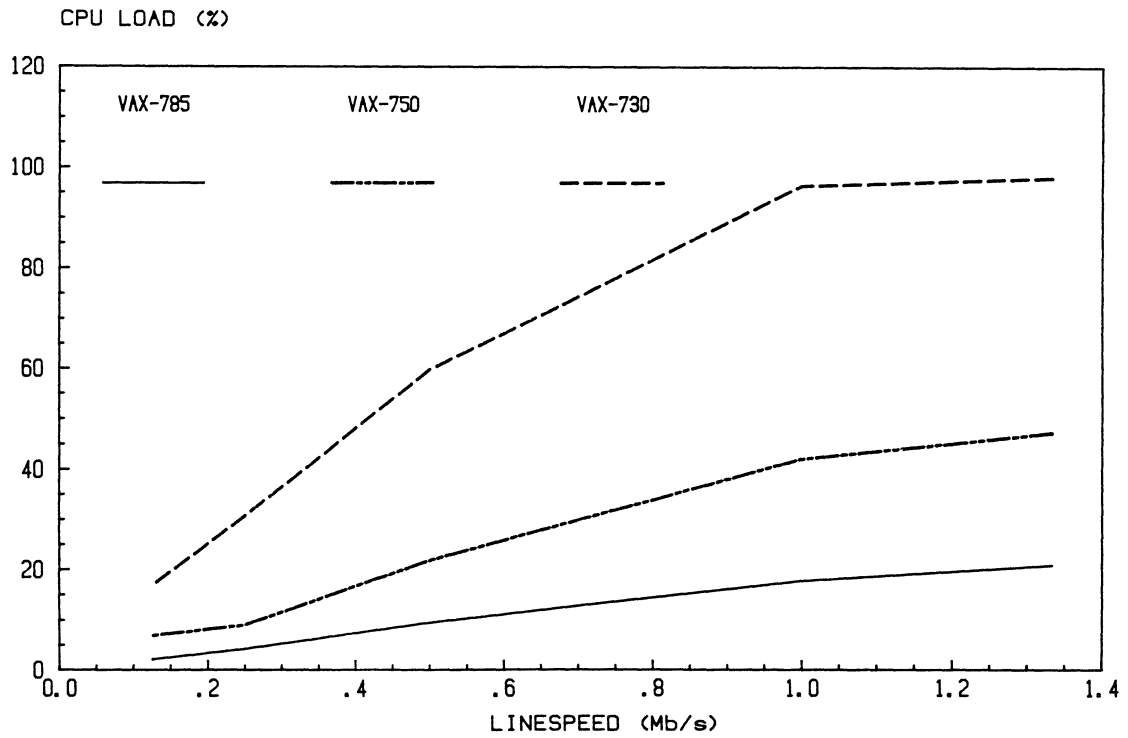
Graph 5. 2048-Byte Buffers: VAX Loading vs Linespeed



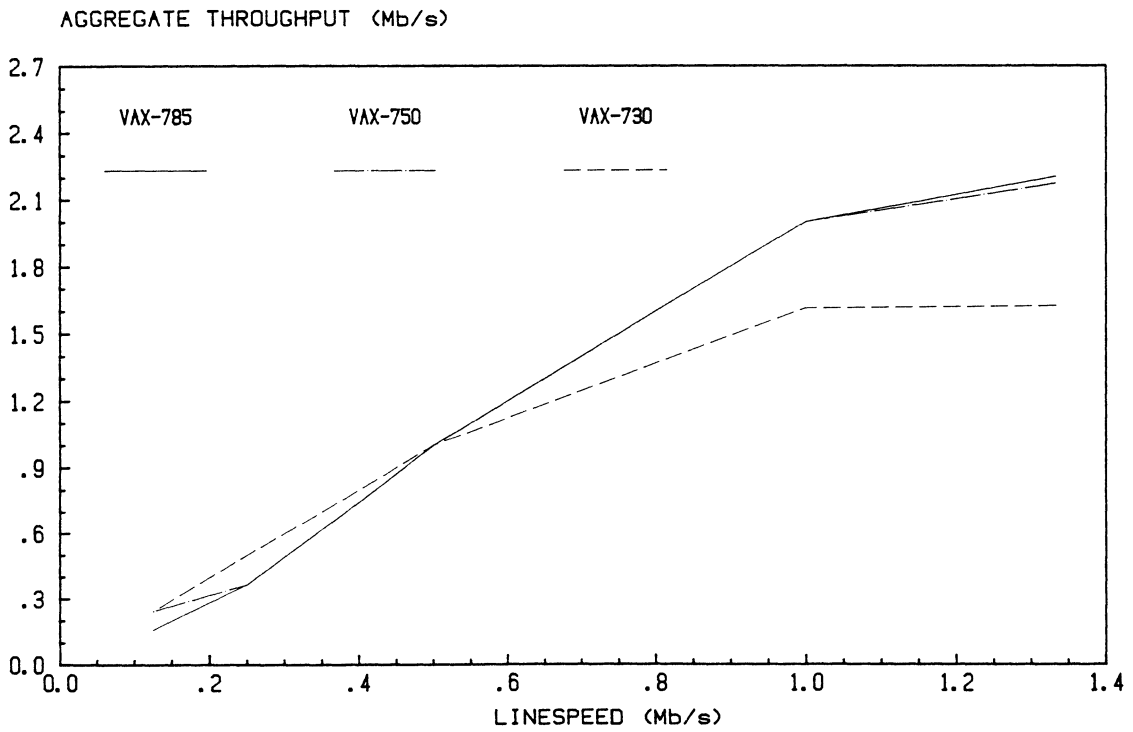
Graph 6. 2048-Byte Buffers: Throughput vs Linespeed



Graph 7. 4096-Byte Buffers: VAX Loading vs Linespeed



Graph 8. 4096-Byte Buffers: Throughput vs Linespeed



FUNCTION TO FUNCTION COMMUNICATION WITH HIGH LEVEL TRANSPARENCY

Thomas B. Macy
Kodak Colorado Division
Windsor, Colorado

High Level Transparency allows one program to connect to another program without needing to know where the other function resides or how that connection physically takes place. It also provides a standard method of inter-function communication. High Level Transparency can simplify the design, implementation, and maintenance of systems.

INTRODUCTION

In some contexts it's called High Level Transparency and refers to a characteristic of a high level communication protocol. In other applications it's a characteristic of Message Routing and refers to common methods of communicating messages. In some computer systems it might be part of an inter-process communication protocol which consists of a comprehensive set of rules governing communication and data access. In still other computer systems it is a characteristic of what might be called an application bus. What it is, is inter-function communication without regard to node residency. These concepts provide a framework for exchange of information among functions. And, in a network, the interface to this framework is independent of node residency. Two functions can exchange information and they can reside on the same computer or on different nodes in a multi-computer network. Where functions physically reside is transparent. In this paper I'll call it function to function communication with High Level Transparency, or just HLT for short. It simplifies system design, system tuning, system enhancements, and system maintainability.

Imagine a system in which functions are sources of information or transformation into which any function can tap. If you know some function has information you need for a new application, pass it an appropriate request and, with no change to the called function, wait for the response. Likewise, the functions you write should expect requests from the system and respond according to the rules governing HLT.

Because with HLT you know the inter-function relationships, the making of additions or modifications to functions becomes simplified. The ramifications of enhancements become more apparent than in systems without HLT. The total system is simplified through inherent modular

definition and standard communication methods. Development times should be shortened, first because of the enforced structure and second because of the ready availability of data already present elsewhere in a network. New network applications which need the functionality already defined elsewhere in the network do not require any external modifications to get it. Just tap into existing functions via HLT. Tuning of an installed multicomputer application can be as simple as redistributing functions among nodes until the right mix is found. Remember, node residency is transparent. Because of the ready availability of function operations throughout a network, duplication of software can be avoided as well as the duplication of related hardware.

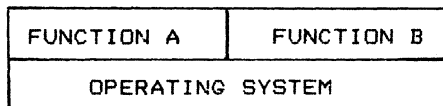
Networks are often ignored as a solution to a problem, and often an application will be designed without regard to the pre-existence of information or processes on other computers. Networking is seldom considered as a template when solutions to a problem are being sought. It seems the tendency of system's analysts in some cases is to view applications in a parochial manner. Rather than applying a problem to a template of a vibrant network of distributed systems, many analysts apply a single computer to solving a problem. This needs to change if the benefits of the newer technologies are to be realized. (A single computer may be the best solution, but it should be viewed as the simplest kind of network -- a network of one computer.) The advent of LAN's and the development of communication standards are signs of an exciting future and a rewarding one for those who are able to capitalize on the new technologies.

One way to capitalize on them is to be able to define the application in terms of the system functions, not in terms of a

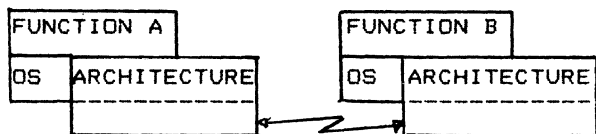
specific technology. Once these functions have been identified they can be assigned to technologies which are chosen based upon user needs which may not be directly related to the problem being solved by the proposed computer system. This simplification demands that HLT exist so that functional definition can be accomplished without regard to node residency.

DEFINITION

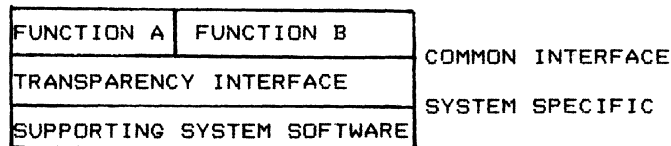
HLT is that quality in a computer system which allows for inter module or inter function communication without regard to supporting system software. Communication architectures are part of this supporting software. Providing function to function communication that causes the physical connection between functions to be transparent, implies the ability to both design and code application modules communicating with other modules without the need to consider where in a network the modules reside. Essentially, they become "plug-in" software units. If a program module needs to be added, code it and plug it in. If it needs to be moved to another node, unplug it from one node and plug it into the other. HLT then, while providing node transparency, transcends simply definition as another layer in a communications architecture. By providing an interface between functions, it becomes a foundation upon which applications can be built. The definition of the application environment need no longer be in terms of the technology used to implement the final solution.



Diversity of operating system functional links



Increased diversity from networked disparate operating systems.



Simplification through transparency

Figure 1

SCOPE OF THIS PAPER

To implement HLT, two essentials must be designed into the system. Standardized inter-function communication through routing must exist to perform the transfer of messages. Provision must be made for the specification of source and destination telling the router where to send a message.

In order to make the best use of HLT, we also used structured design concepts to promote clearly defined function definition. An aspect of this is something called I/O independence. While not required for HLT, these concepts do enhance its effectiveness and so are included in this paper's discussion of HLT.

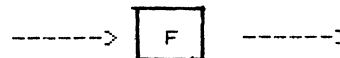
Based upon an effort to provide HLT, this paper is the result of work done in a network of four computers. Two HP 9920 computers with single tasking BASIC operating systems each control their own laboratory instrument. A PDP 11/24 running RSX11M acts as a communications controller and application coordinator. The IBM IMS data base system running on an IBM 3033 mainframe provides plantwide access to instrument readings as well as long term trending.

While we kept this first system simple, many enhancements have been identified. This paper describes the implementation of HLT on a transaction basis. Single requests or transactions are exchanged in a node transparent manner. Future enhancements could include the following:

1. HLT can be part of a more comprehensive inter-function communication supervisor providing for scheduling of module execution based upon some criterion such as priority and including data access tokens for mutual exclusion.
2. Communication between functions could be established and maintained as sessions consisting of many exchanges.
3. Notification of other nodes about the implementation of a newly added function could be a dynamic and automatic process.

STRUCTURED DESIGN FOR INDEPENDENT FUNCTIONS

Any program unit can be pictured as a discreet block of logic with an input and an output defined.



Inputs and outputs can be classified as either of two types. Functional I/O can connect to the outside world, as is the case with device drivers, or it can form a connection between programs within the computer system as with inter program communication. Device connections are defined externally to the application and

are more structured. However, most functional inputs and outputs within a modularly defined system exist to connect functions and are less structured.

What provides an interesting prospect in the design of systems is the provision of independent single functioned modules with inputs and outputs defined in a standard way, where possible, throughout the application. The more single functioned a program module is, the greater the likelihood of making its input and output universally available. Defining the input and output rules in a standard way provides flexibility in defining program links. While existence of HLT does not require such structured design, modular definition of an application tends to enhance transparency by making a larger base of simple functions available for connection.

Before HLT can provide the benefits of plug-in software modules to the designer of computer systems, the concept of structured design should be adhered to. According to rules of structured design, the goal of any computer system application should be its implementation as a group of highly independent single functioned modules. [3] Real benefits result when the application is composed of functions of high cohesion and low coupling communicating in a common way throughout the system. Cohesion is a measure of exactly how "strong," or single functioned, each module is. Coupling is a measure of exactly how interdependent the modules are. [6] Clearly defined single functioned modules enhance the benefits of HLT by providing concise module functionality at a low enough level to have appeal throughout an application. For example, while a complex function F may have only one unique use in an application, the functions which make it up, say X, Y, Z, may have use elsewhere in the system.

This paper refers to program modules as functions. If we define a software "system primitive" as a low-level component whose feasibility and correct functioning is absolutely essential in order to implement the system, then the definition of the system is essentially complete when all system primitives are defined. A "simple function" consists of the implementation of one system primitive and a "complex function" consists of calls to other functions. If all the system primitives can be identified, then the final application is nothing more than the combining of these into functions and the defining of the lines of communication.

Finally, there are many methods defined for decomposing an application into functions. [2,8] Some people tend to prefer one method of identifying functions over another. However, to effectively implement HLT while maximizing the benefits to be gained from structured design, any type of decomposition should be used. They all add to the identifying of system primitives and defining of functions.

STANDARDIZED COMMUNICATIONS

The concept of modular systems made up of plug-in units is not new. From the concept of structured software design, a hardware approach to computer systems has evolved called Function to Function Architecture (FFA). In FFA

"a set of functional modules called actors implements specific system functions. An actor is a functional computing element packaged as a discreet plug-in unit. Each actor has a defined method of interfacing with the FFA system, and an FFA-standardized means of communicating with other actors. The FFA itself is a set of conventions (rules) for gaining access to any actor. It provides the mechanism for actor-to-actor communication " [1] (p 142)

HLT is a software parallel to this hardware concept. The use of structured system design and structured programming provides for an application consisting of functions which are as independent as possible. However, many discussions on the subject of structured programming and design have not addressed the benefits to be gained through the "standardized" means of communication among these "software actors" or functions. As the plug-in nature of hardware modules is only possible because of a standardized means of communication among actors, so, also, is the ability to produce plug-in software modules dependent upon standardized communications. HLT provides this. Software modules can be plugged into the system anywhere in the network and be assured of proper inter-function linkages.

The question to be answered is how can a function be defined and coded so as to facilitate common communications in the connection between two functions? This is the key to node independent function to function communication in software. In order to provide this, the first thing to consider is a special characteristic to be designed into functions -- I/O independence. A second consideration is the technique chosen to invoke the logical and physical aspects of inter-function communication. This is the routing aspect of inter-function communication. We provided a router to implement HLT.

I/O INDEPENDENT DESIGN

Given the two functions F1 and F2 and a need for them to exchange data, there exists a logical connection L and a physical connection P between them. (See figure 2.) L defines the end to end relationship and P defines how it is to be invoked. Implementation of HLT requires that this connection between F1 and F2 have the following characteristics:

1. Connection L must be node independent. As part of the data format, it defines

which functions are connected, not how they are connected. This also enhances low coupling. See the common interface in figure 1.

2. Connection P is node dependent. It defines the physical steps necessary to establish a connection from its node's point of view. See the system specific interface of figure 1.
3. Connection L must be common throughout the network. This provides the hook for implementation and use of HLT.

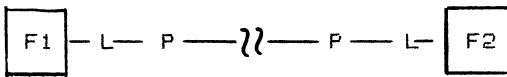


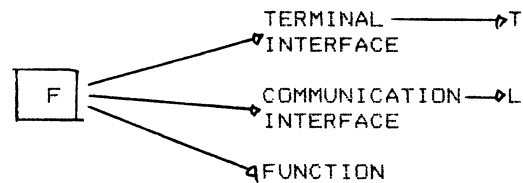
Figure 2

While L does not manifest itself as code in the final system, P does. All too often, even if F1 and F2 are identified as system primitives, P is included as part of their design. Structured design promotes the separation of this connection into a separate and distinct module. It can become a simple function in the application. Making L and P independent of F1 and F2 does more than promote simplicity. It promotes HLT by removing rules concerning details of communication from the context of functions.

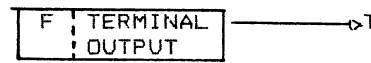
One consideration when defining application functions which are inherently node independent is that they must be designed free from dependence upon system I/O. (See figure 3.) Any function which connects to a system call with system unique methods will be dependent upon that node for proper execution. For example, outputting data to a terminal may require 1) some kind of an assign command for gaining access to the device and 2) execution of some kind of output statement like a WRITE. These instructions are interpreted by the operating system as local to the host node. If a program generates output specifically for a terminal, then that function's output is not available to other functions without special coding. It's then not available to other nodes. Therefore, node independence requires that functions be designed independent of I/O. I/O can be separated into singular functions.

Execution of I/O qualifies as a system primitive. To separate I/O from other functions, define system primitives which perform the necessary I/O and make these primitives simple functions. For example, output to a terminal may be defined as a singular function. Other functions needing to output to a terminal link to the singular terminal output function.

How then can all these functions connect to each other in a coherent manner where logical connections L are node transparent and physical connections are transparent to the functions involved?



Output from a function can go anywhere.



Output from a function tied to a specific destination.

Figure 3

ROUTING FOR STANDARDIZED COMMUNICATION

Once a structured design has yielded an application design with independent functions, the key to HLT is the reliability and effectiveness of the method used to connect these functions to each other. Communication between two functions might be defined as a session which is established when a function is called and ended when no further interaction is required. For simplicity, our application initially consisted of single transmission sessions.

In this simplest case, a Router is a program, defined by node, which merely passes a message from one program to another. The actual route a message takes is determined by a Routing File. Referring back to figure 2, L defines which functions are connected and is part of the data. This promotes simplicity. P defines how the functions are to be linked by the Router.

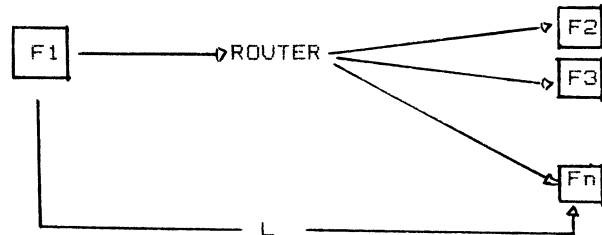


Figure 4

The Router and the Routing File implement the node transparent communication made possible by the I/O independence gained through structured design and structured programming. Those physical aspects of function to function communication which tend to be stumbling blocks in networks are isolated, being limited to the Routers implemented in the system and to utility functions performing the mechanics of system specific interfaces.

THE ROUTING FILE

The Routing File provides the means by which a Router can facilitate HLT in function to function communication. It defines the connections available on its node. A Routing File must contain the following information:

1. Legitimate function ID's
2. Where in the message the ID's reside (figure 6)
3. The length of the ID's
4. A map describing where a message is to be sent on the local node given a destination function ID
5. A specification of how to send the data to the function to receive the data on the local node and how to start that function

Routing ideally makes use of system procedures available on the node on which the Router runs. For example, on a DEC RSX 11M system, messages can be routed through a disk file or queue, passed through memory or common, and passed using EXEC calls.

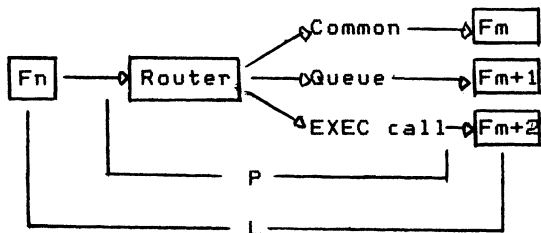


Figure 5

DESTINATION_AND_SOURCE_DEFINITION

Often linking from Fn to Fm will require a response return from Fm to Fn. In order to provide that capability in a generic manner, transaction formats should be defined with fields defining both source and destination function ID's. Within the application the length of these names may be determined by restriction on one of the nodes. For example, IMS requires 9 character transaction codes. If an application requires a link to IBM's IMS data base system, it may have its function ID fields defined as 9 characters in length. One could arbitrarily assign the first 18 data bytes of a block of data to be the location of these fields. This provides an adequate number of meaningful names.

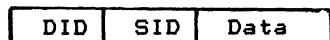


Figure 6

Here SID refers to the source function ID and DID refers to the destination function ID. When this format is adhered to among nodes, function access can be implemented in a node transparent manner via the router, thus providing HLT.

The two-way exchange of data described above is easily implemented within a single node. But how can it be implemented when Fm is on a different node than Fn? With an SID and DID used as defined, the implementation of a router on a node will provide for the exchange through what might be called indirect routing. Communication with other nodes is handled by specific functions -- usually singular. Routing to a function on another node is handled by the Routing File. If Fn wants to connect to Fm and the functions are on the same node, the Routing File tells the Router to pass the message directly from Fn to Fm. However, if Fn and Fm are on different nodes Nn and Nm, the Routing File on Nn tells the Router to pass the message to the singular utility function, Ft, handling transmission to the node containing Fm. On Nm, Fr is the utility function receiving messages from Nn. It passes the message to its local Router which, based on the DID, passes it to Fm. The Routing File defines these routes.

AN EXAMPLE

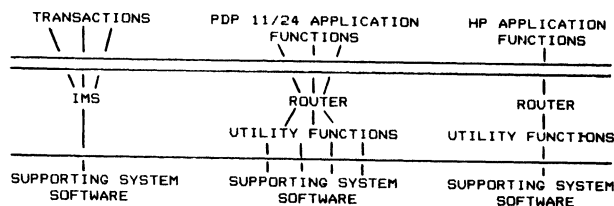


FIGURE 7 - APPLICATION DESIGN WITH HLT

In figure 7, the application layer definition is implemented in the top layer of functions. The middle layer, especially on the HP and DEC systems, provides the node transparency. From this picture note that if the application functions are coded in a high level language using standard language syntax, they are transportable to other systems providing support of that language. If vendor extensions to the language are used, the functions may not be transportable to other vendor's hardware, but they are still compatible with other systems of like vendor within the network. In either case the middle or transparency layer provides benefits in design, installation, maintenance, and tuning of a system.

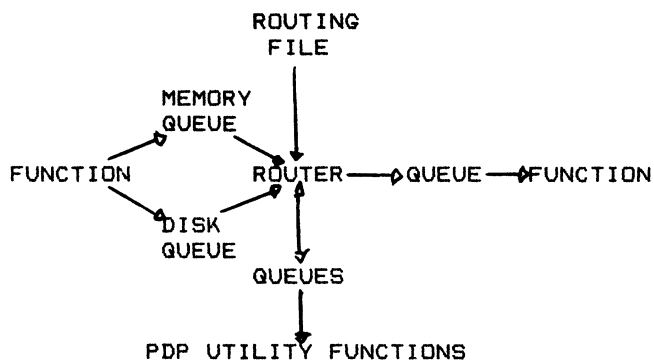


FIGURE 8 - PDP SYSTEM HLT OVERVIEW

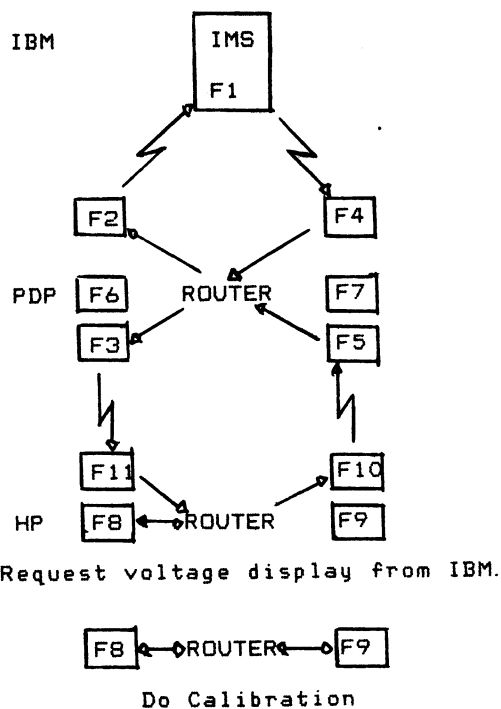
The one problem encountered in this design is the transmission delay incurred by funneling all communication through two queues and one router. However, in our application, response time of exchanges was not considered important because production operators were not involved in these exchanges. When response time does become an issue, new routers can be added to improve it.

Our application consists of three levels of computers. The lowest level, level 1, is the dedicated HP 9920's each controlling a piece of equipment. Level 2 is a PDP 11/24 RSX 11M system acting as the communications and shop floor resource controller. Level 3 is our IBM central processor. In this network, a subset of the identified functions are as follows:

FUNCTION	NODE	DESCRIPTION
F1	L3	Display a device's voltages
F2	L2	Transmit to IBM (utility)
F3	L2	Transmit to HP (utility)
F4	L2	Receive from IBM (utility)
F5	L2	Receive from HP (utility)
F6	L2	Request voltages
F7	L2	Output to a terminal
F8	L1	Read voltages
F9	L1	Do calibration
F10	L1	Transmit to PDP (utility)
F11	L1	Receive from PDP (utility)

Diagrammatically, this part of the application may be represented performing two operations in the following way. Note that output from F8 goes to different destinations and can go to more in the future if desired.

In this example, with the voltage reading made a separate function, any function on any node can request a reading of the voltages and, via routing, get the response returned. The calibration function can read voltages and the IMS transaction can read voltages. At some point in the future, a function on the PDP 11/24 could request voltages and require no change to any other function in the application. This example shows the same capability for



Request voltage display from IBM.

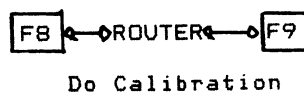


Figure 9

centrally coordinated time and date. Correctly defining and specifying application functions can make functions, that have traditionally been parochially defined by node, available to the entire network as an inherent aspect of correct system definition.

IN SUMMARY

With a few simple functions defined to route messages within a structured system, high level transparency can be achieved. This should provide 1) the ability to address application definition without regard to technology and 2) the ability to apply networked technologies to the structured design to meet other criteria. An application defined with HLT as a foundation is not limited by technology. Modules can be plugged in, unplugged, or moved about in a networked solution. The benefits gained from High Level Transparency are these:

1. easy node upgradeability
2. lower application development costs and lead times
3. simpler software maintenance
4. easier application tuning
5. potentially less capital involved in hardware.

High level transparency is one way to take advantage of what the future offers to computer system design.

Bibliography

1. Conrad, M., Hopkins, W., "Distributed Functions Build Modern Computer Systems", ELECTRONIC DESIGN, September 3, 1981, (pp 142-147)
2. DeMarco, T., "Structured Analysis and System Specification," Yourdon, Inc., 1978
3. Fish, Raymond, "Structured Design Ensures High Quality System", COMPUTER WORLD, September 26, 1977, Pg S/13
4. Giloi, W.K., Behr, P., "An IPC Protocol And Its Hardware Realization For A High-speed Distributed Multicomputer System," 8th Annual Symposium on Computer Applications, IEEE, 12-14 May, 1981, (pp 481-493)
5. McGlinchey, J., "Real-Time Computer Applications," DIGITAL Educational Services, 1984
6. Stevens, W., Myers, G., Constantine, L., "Structured Design," IBM JOURNAL, Vol 13, Number 2, 1974, (pp 115-139)
7. Yohe, J. M., "An Overview of Programming Practices", COMPUTING SURVEYS, Vol 6, No 4, December 1974, (pp 221-243)
8. Yourdon, E.N., "Techniques of Program Structure and Design," Prentice Hall, 1975

OFFICE AUTOMATION SIG

Lesley Tracy Kornis
American Management Systems, Inc.
Arlington, Virginia

Abstract

The purpose of this paper is to discuss the impacts of information sharing across office lines upon the the system design and productivity measurements of office automation. Because of information sharing, traditional productivity measurement and system design techniques may not yield the most efficient and expedient results. Rather, due to the very nature of information sharing, novel techniques can be developed for system design and productivity measurements which focus on information sharing, instead of the more traditional view of information processing.

INTRODUCTION

The intent of this paper is to discuss office automation and productivity measurement. However, a plan for productivity measurement must occur concurrently with a plan for designing the system. To devise a methodology for productivity measurement without understanding the system design is analogous to a farmer measuring crop production without knowing what he is growing, how many he will grow, or exactly how his crops will grow.

This discussion centers around information sharing in an office automation environment.

PRODUCTIVITY AND SYSTEM DESIGN

Productivity

Historically productivity has been measured quantifiably because profits are gauged by measuring a person's labors, be it manual labor, industrial labor, manufacturing labor, or farming labor. Recently, another category indigenous to the twentieth century has emerged. This is white collar labor, personified by the office worker whose primary purpose is the creation, the production and the use of information. Because office work is an integral part of doing business, it too has come to be viewed in terms of productivity, particularly in light of justifying expensive computer equipment.

Although technologies for the office have evolved quickly over the past 10 years, little has changed in the traditional view of productivity measurement - which is a method for assessing the success of a system in fulfilling its intended mission. As few as three years ago, articles, books and Federal Computer Conferences were discussing productivity measurement techniques for word processing. Just as these methods for

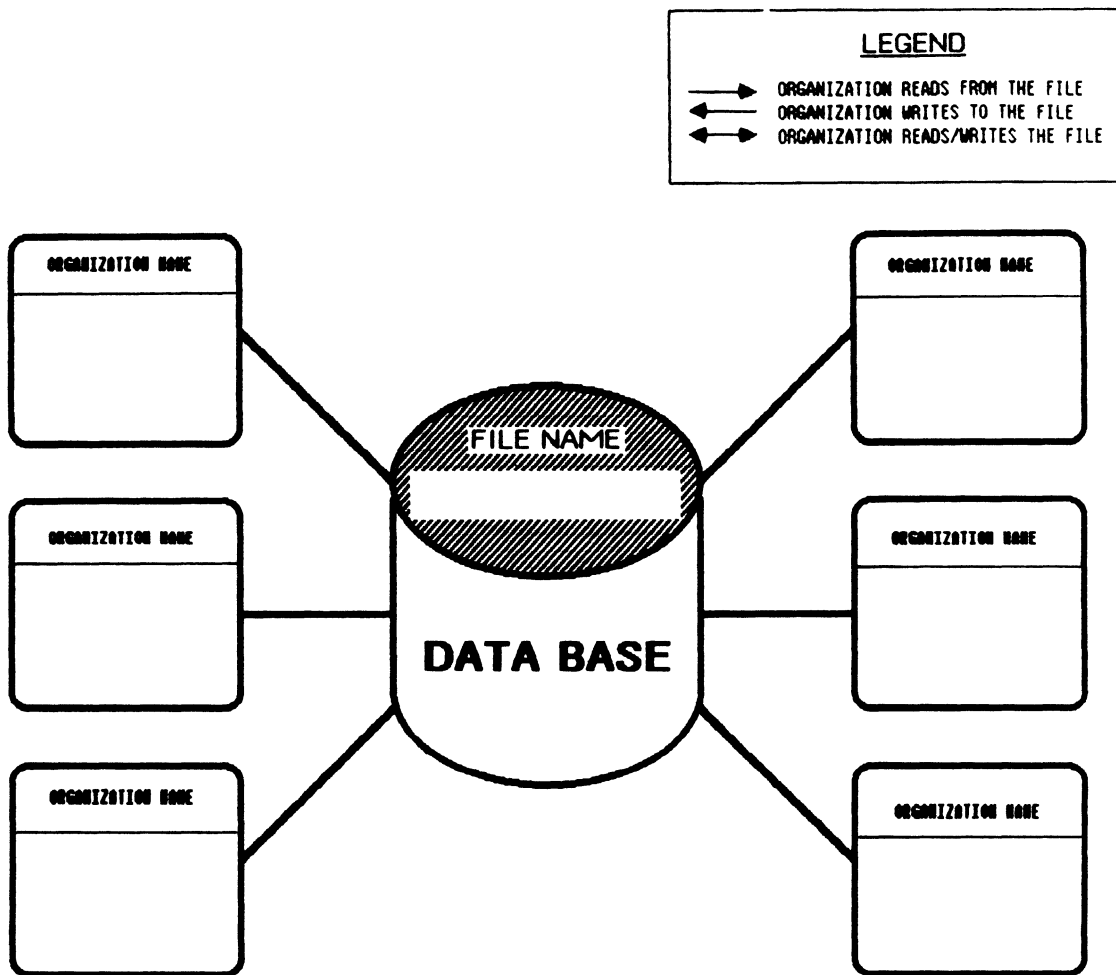
measuring office productivity were evolving and becoming standardized, new office technologies were being introduced. These technologies, now known as office automation (OA), are a consolidation of traditional word processing, data processing and other information tools and technologies into corporate, packaged systems. What has resulted is a gap between the existing standard productivity measurements for simple word processing and the newer more complex productivity measurements for office automation technology.

System Design

System design is the application of specific techniques for developing an automated system. The goal is to develop the system in the shortest length of time between user request and delivery. Traditionally tools for system design have lagged behind the rapid evolution of computer field technologies. Commonly used system design techniques include flow charts, data flow diagrams, structured English, decision tables, decision trees, and data dictionaries. Whatever tools are used, the sequence for designing a system is as follows: determination of the requirements; deciding what information is required for accomplishing the activities; and utilizing one of the tools described above to develop design specifications.

To illustrate more clearly, one such tool, data flow diagrams developed by Demarco (1), is a graphical representation of the system. Symbols are used to represent processes, data flows, and files. Activities of an organization are broken into processes and the data is depicted in terms of data stores. The data stores are sources of files which are identified in conjunction with the processes.

FIGURE 1



FILE INFORMATION

OWNER: _____

SIZE: _____

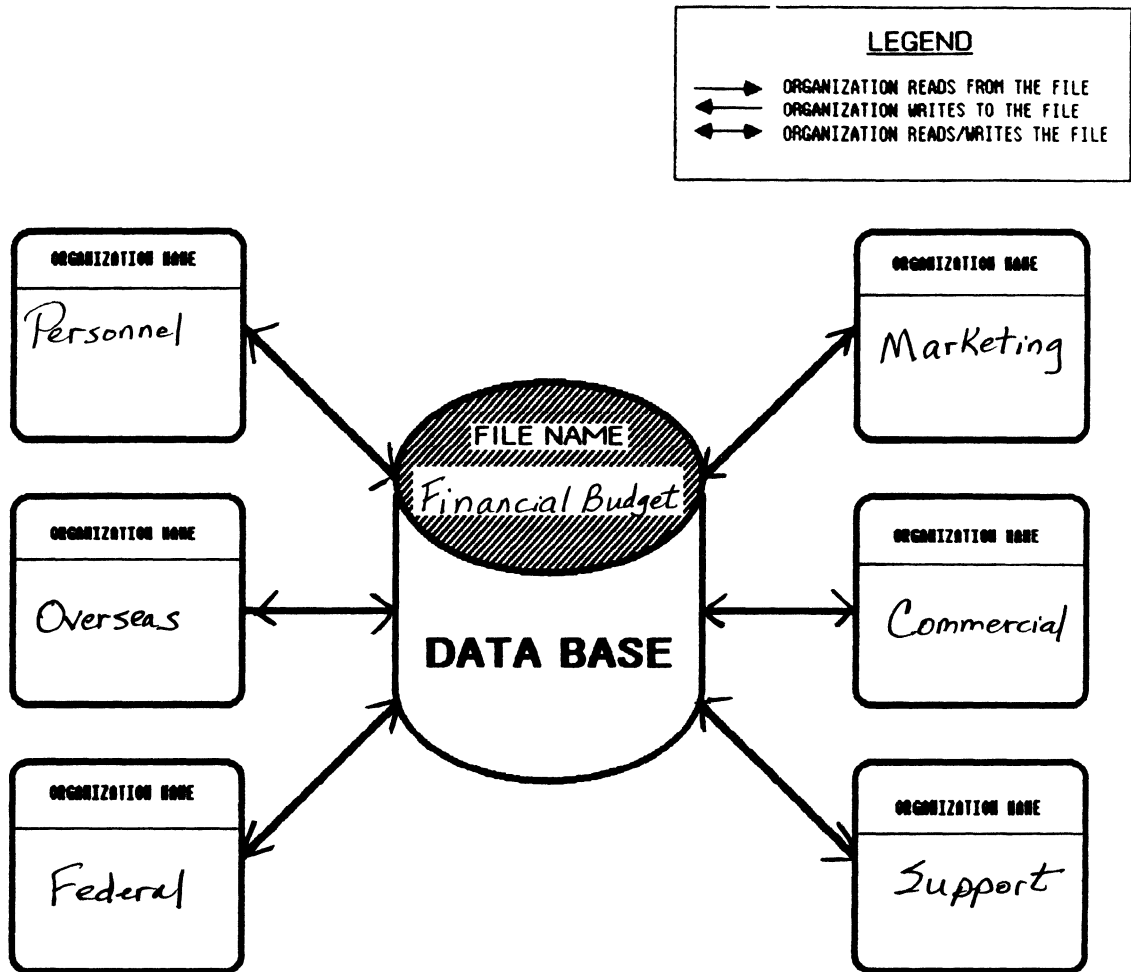
USAGE: _____

FREQUENCY OF USE: _____

CLASSIFICATION: _____

DESCRIPTION: _____

FIGURE 2



FILE INFORMATION

OWNER: Finance and Accounting
 SIZE: 15 megabytes
 USAGE: Enter data, calculate, and provide reports
 FREQUENCY OF USE: Quarterly

DESCRIPTION: Each organization creates and maintains their respective budgets. These individual budgets make up the financial budget.

Office Automation Definition

Office automation has come to mean many things - from word processing to a wide variety of data processing technologies used in the office environment, such as electronic mail, local area networking, and graphics, to name a few. For the purposes of this paper, office automation refers to an array of computers organized in a distributed processing system design. In this case distributed processing system is comprised of a typical mainframe, minis and microcomputers networked together to provide the vehicle for information sharing or independent processing at each level.

Information Leveraging

A distributed processing architecture and corresponding communication network facilitate the sharing of information. Sharing information produces information leveraging which is the electronic sharing of the same information by authorized offices with the effect of magnifying the value of information because of the multiple users. Although the value of information varies from office to office, all participating will gain from the leveraging of information which would otherwise be concentrated in one place. New processes performed on the information, such as word processing, data processing and communications are propelled by information leveraging among different personnel levels, different systems and for different applications. Duplicative collection procedures for similar information can be consolidated and synthesized into a comprehensive and coherent set of data which can be used by managers and professionals alike, and ultimately shared as a "corporate data base".

An office automation system, as used here, is a distributed processing system comprised of hardware (mainframe, mini and micro) serving different individuals, different organizations, and different applications. A unifying factor in office automation is not the equipment, not the people, not the application - but rather, it is the need to share information. And it is the resulting synergism among the individuals sharing that information which makes the design requirements and the productivity measurement techniques unique. Therefore, the sharing of information is the major reason for having an OA system (as defined here); otherwise clusters of independent micros would be fine, as would one mini with a group of terminals connected to it, or any independent localized design versus a distributed processing design.

Office Automation - Why it is Difficult to Measure Productivity

In light of the definition given above, there are many reasons why office automation productivity is difficult to measure. One such reason, previously mentioned, is the leveraging of information which is an abstract concept that does not lend itself readily to quantifiable measurement. Below are related reasons for the difficulties arising from measuring office automation productivity.

Recent and Dynamic Technologies. Office automation is comprised of recent and dynamic technologies. The term office automation and the meaning discussed here reflect this period of time in the mid- to late eighties. It is fair to predict that the meaning here and the meaning five years from now will differ, just as it differed five years ago when office automation primarily referred to word processing. Office automation is dynamic because it is still evolving and changing in tandem with computer technologies.

Office Automation - Greater than the Sum of its Parts. Another characteristic about OA that makes productivity measurement difficult, is that the benefits of office automation are greater than those of the sum of its parts (word processing, data processing, other technologies) and, therefore, to measure only one area of office automation, such as word processing, does not address the interaction or synergism among the parts which generate productivity benefits. Similarly, office automation affects nearly all aspects of office life, not just one discrete area such as typing. Therefore, benefits cannot solely be measured in terms of greater productivity, but rather benefits must also be measured in less quantifiable areas such as greater effectiveness of time usage, improved decision making, and better coordination and communication.

An example of this synergism is the use of a data base which may mean more readily available information. However, it can also effect the efficiency of word processing if it can automatically be reformatted into a report. Or it can provide time savings if the data can be automatically converted into a spread sheet for computations and forecasting.

Shortcomings to Present Approaches to Productivity Measurement. Productivity measurement has traditionally focused on quantifiable measurement, overlooking the crucial role of the manager and professional. For example, in the area of word processing, successful productivity measurement techniques are associated with secretarial/clerical word processing, rather than the more costly professional and managerial word processing which affects productivity quite differently from the secretarial/clerical ranks. Similarly, productivity associated with clerical tasks is typically quantifiable due to the structured nature of the tasks. For example, measurement techniques such as counting lines of output on a word processor or counting the number of information retrievals to determine the average time per retrieval in a manual and computer-aided environment, are all appropriate for the structured environment of the clerical staff. In contrast measurement of the executive and management levels is more difficult due to the unstructured nature of their work. Most agree that at these levels the majority of the time is spent communicating, be it in meetings, telephone conversations, and conferences, all of which are unstructured.

System design for office automation needs to capture the leveraging of information via file sharing. In the traditional view the major thrust of the system design is the determination of what process requires what specific capabilities. However, with office automation this area does not need to receive much initial attention because in many cases off-the-shelf software predetermines the capabilities (electronic mail, data base management, word processing, spreadsheet, graphics, and calendaring). These capabilities are based upon routine administrative requirements in an office environment. Therefore, system design for office automation does not have to be centered on how the data will be manipulated within a particular environment, because it is already known. And instead a systems analyst might say - here are the capabilities, what is your information and where does it go? In traditional system design the processes are the foundation, but in office automation the files/data should be the foundation.

NEW APPROACHES TO OA SYSTEM DESIGN AND PRODUCTIVITY

Presented here is a system design technique which focuses on the sharing of information. This approach is recommended in conjunction with the more traditional techniques. The important point here is that the information sharing functions are identified and incorporated into the design.

The initial step of this system technique, which identifies the information sharing functions, is to define the data that needs to be shared. A method for doing this is depicted in the following illustrations. Figure 1, the Information Diagram, presents a tool for obtaining the flow of information/data/files within an organization, as well as specific descriptive information regarding the files. This is a tool for creating a concise list of shared files. First, the file is identified. Second, the owner, or organization responsible for the file is requested to identify organizations that share the file. Third, information concerning the file, found in the section called File Information, (Figure 1) are provided.

Once a diagram is completed on every shared file identified within the organization, the data can be organized any number of ways for analysis. This data is the foundation of the design, insofar as it depicts the files that are shared throughout the organization. Figure 2 illustrates a completed Information Diagram and Figure 3 shows a representation of the the data.

The second step is to further identify the files in terms of what capabilities will be performed on them (i.e. word processing, data base management, spread sheet, etc.). This is followed by file size estimations for sizing purposes. The source of this information is in the file descriptions accompanying the Information Diagrams.

Once the shared files have been identified as stated above, a design called the information architecture is prepared. This design incorporates the concept of leveraging information. Essentially, the information architecture is comprised of two types of files: public files which are shared by others (and were identified in the Information Diagrams); and private files which are files that are not shared. The private files need to be identified by each organization, along with the same information requested on the Information Diagrams. Therefore, what information needs to be shared and by whom is determined before the design. The defining and grouping of information into files can be accomplished using the Information Diagrams (See Figure 1). Collectively these files will comprise the corporate data base. It is the grouping of files into public and private groups, which becomes the backbone of the information architecture.

The size, purpose, and importance of files, are all factors in determining where they will reside. It is conceivable that a public file could reside on a micro, mini or mainframe, depending upon its purpose.

Measurement of Productivity - Deriving a Productivity Factor for Leveraged Information

Presented in this section are some ideas for measuring productivity as it relates to the leveraging of information. The leveraging of information is characteristic of the managerial and professional workloads, which are typically unstructured and information oriented. One way to measure information leveraging is to look at the frequency of file usage. An audit trail will give this information, in addition to tracking the user of the file. The frequency of file usage can be an indicator of productivity as it pertains to information leveraging. It can also indicate the actual value or "corporate value" of information in relation to the office or the corporation as a whole, as determined by frequency of use.

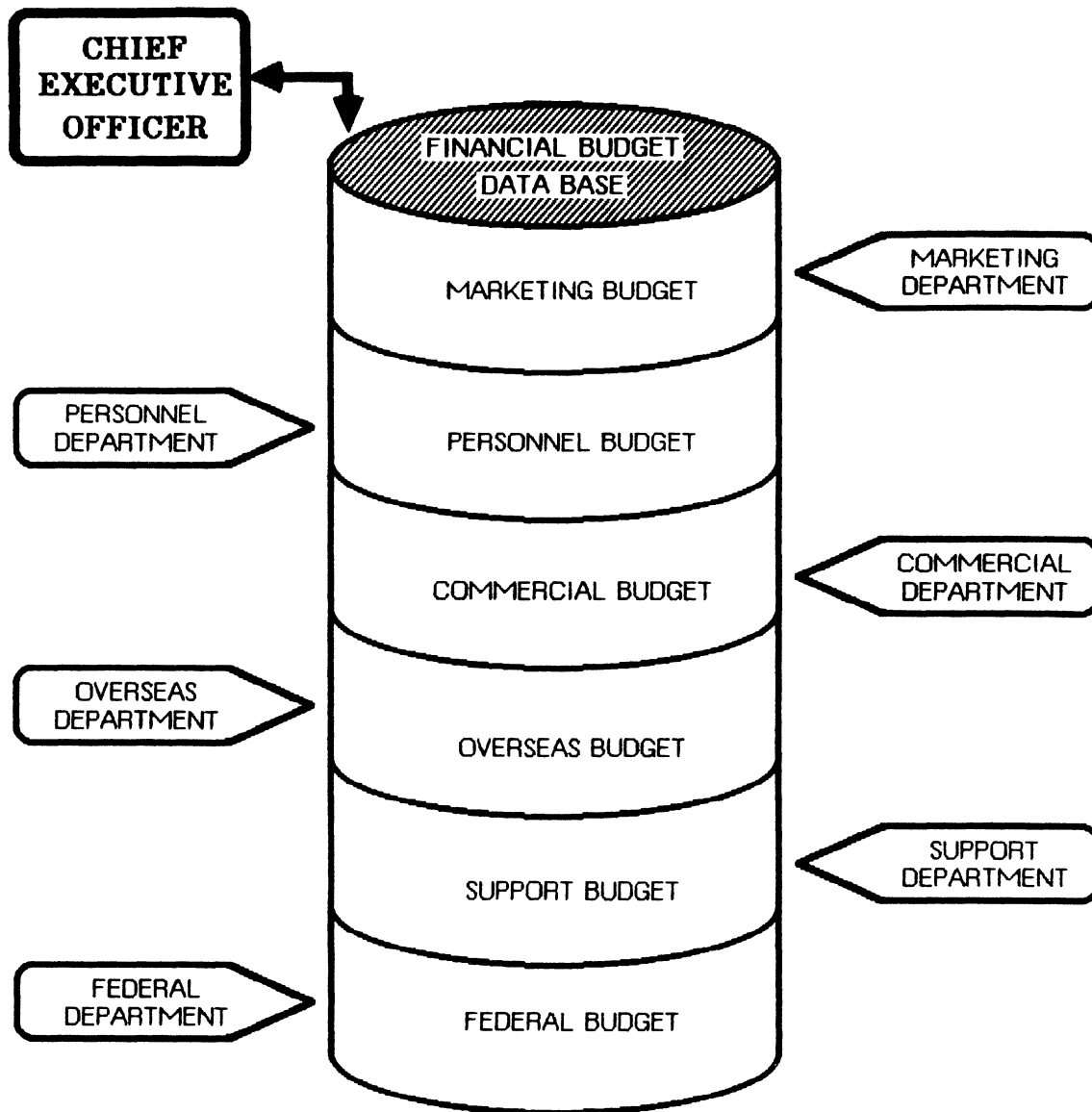
There are several different ways to use audit trails as productivity measurements. First of all an underlying premise is that the more frequently information within the system is used, the more productive those workers are that use the information. This is particularly applicable at the management and executive levels. In other words, the more frequently that they are accessing information, the more productive they are in terms of efficiency, improved decision making and effectiveness.

Different views of the audit trails will provide different indicators of productivity. Below are some ideas.

FIGURE 3

INFORMATION DIAGRAM

FINANCE AND ACCOUNTING



1. Weekly or monthly audit trail reports of frequency of file usage can yield a history of the most widely used files within the "corporate data base". Likewise, it can reveal those workers or organizations with the greatest participation in information leveraging, i.e., productivity improvement.

2. Files can be assigned a value based upon their administrative value. For example, an employee roster from personnel will be assigned a greater weight than a computer equipment list file. This is so because the frequency and widespread use of the personnel file gives it more "value" in an administrative or office automation sense, than does a more narrowly defined and used file such as a computer equipment list.

3. If in fact the files are assigned a value, as described above, then the frequency of use of those files of higher corporate value, will indicate how effectively and efficiently the system is being used.

SUMMARY

This paper presents some novel approaches for productivity measurement and system design that focus on the unique characteristics of office automation. None of these techniques is intended to be used exclusively, but rather, to be used in conjunction with more traditional methods, depending upon the purpose or goals in mind.

In order to experience the information synergism or leveraging discussed here, an organization must be committed to the sharing of information. In many organizations information is power, and therefore, it is not something that is readily shared. This is a principle that appears to be deeply engrained in many establishments.

Therefore, philosophical changes will have to take place concerning information and its value to a corporation as a whole, in terms of time and efficiency, in order for information to be leveraged effectively.

REFERENCES

1. Structured Analysis and System Specifications, DeMarco, Tom, Yourdan, New York, 1978.

ALL-IN-1 VERSION 2.0 - ONE YEAR OF OPERATION

E. C. Eimutis
Monsanto Research Corporation - Mound
Miamisburg, Ohio

ABSTRACT

An Automated Office Support System (AOSS) pilot project, consisting of 30 target users from a manufacturing group and 30 additional training and computer systems personnel, has been successfully completed.

The pilot users were connected to a Digital Equipment Corporation (DEC) VAX 11/780 computer running External Field Test Versions 2.1, 2.2, 2.3, and, subsequently, formal release of Version 2.0 of the ALL-IN-1 office automation software product. Electronic Messaging was the major office automation function that was used and evaluated.

OBJECTIVE

The objective of the Automated Office Support System (AOSS) is to increase overall organizational productivity.

The major premise is that ALL-IN-1, if properly implemented, will be a key factor supporting this increase in organizational productivity. The increased productivity will be accomplished in part by:

- improving the timeliness and accuracy of one-way communications
- minimizing individual work-flow deviations and unwanted interruptions
- improving the ease and speed of one-way communications
- minimizing telephone interruptions
- minimizing "telephone tag"
- reducing paperwork and paper handling
- reducing the frequency and volume of photocopying
- and by reducing the time spent in searching for and retrieving information.

The specific results to be worked toward in the pilot project were:

- to evaluate the office automation software, ALL-IN-1, especially the Electronic Messaging aspects of this software,
- to justify, procure, install, operate, and evaluate the VAX 11/780 computer system in an office automation environment,
- to establish a pilot target user group from an operational department, interface them to the computer and

enable them to use the office automation software,

- to evaluate specific end user application requirements and how they could be met,
- and to evaluate system programming requirements for software upgrades and for customization of the ALL-IN-1 software product.

RESULTS

The Automated Office Support System pilot project has been successfully completed.

A pilot Target User Group of 30 individuals from a manufacturing group were connected to the DEC VAX 11/780 computer system running the ALL-IN-1 office automation software product. The target users evaluated Electronic Messaging features of ALL-IN-1 and a majority of the users found these features very useful.

An Implementation Plan justifying the computer system was written and approved. The computer system was procured, installed, and made operational on time and on schedule. The performance of the computer hardware was initially erratic but, as of this writing, has stabilized to provide continuous operation and non-stop performance for the last four months. Peripheral breakdowns in the disk drives and in the communications interfaces, however, will require that clustering be implemented to enhance the performance of these devices and to provide a required availability of 99.9%

Specific end user observations and requirements were obtained by means of a "Needs Assessment". These observations have been analyzed and are described fully in this report.

A number of systems programming requirements arose concerning both software upgrades and customization of ALL-IN-1 menus. These requirements are documented below.

Several additional requirements and needs were identified in following areas: training, system software, applications software development, hardware, and general. These requirements are summarized below.

PRODUCTIVITY OBSERVATIONS

• improving the timeliness and accuracy of one-way communications

There were a number of examples where this was observed. Several individuals are using AOSS for weekly report generation and distribution. The time required for this routine manual process has been reduced by approximately 50%. One group in particular, has established weekly reporting from a remote office in Phoenix. The weekly summaries are available for Product Management on Monday morning via AOSS.

Other individuals have requested that agency personnel be given access to AOSS. This has facilitated the exchange of both routine specification data and occasional special information, such as an elaborate meeting agenda, several versions of which, were distributed electronically to over two dozen individuals. The time savings were significant both in the speed of dissemination and in the lower effort required to produce and revise the several versions until a final agenda was agreed upon.

• minimizing individual work-flow deviations and unwanted interruptions

While the evidence for this is not as clear as in the preceding case, a number of individuals reported that receiving electronic mail and being notified immediately of its presence did not act as an interruption. Actually, in several instances, where individuals were expecting a communication, this aided in the timely accomplishment of specific tasks. If, however, the electronic communication was used instead of an unannounced personal visit, then the savings in work-flow interruption are significantly improved.

The key to this aspect of AOSS is that it allows each individual to operate in the manner they find most convenient. That is, how quickly one responds to an electronic message may be a function of the current work in progress. The ALL-IN-1 software provides tools to help in this area. For example, if while editing a memo, one is notified of new mail, the "Interrupt" feature permits the reading of new mail without having to leave the editing process.

• minimizing telephone interruptions

Some pilot users have devised imaginative ways to use AOSS as a means to reduce telephone interruptions. They have assigned an individual the task of answering their phones and routing the phone messages to them via electronic mail.

• minimizing "telephone tag"

A majority of the pilot users felt that this was an important, though presently not readily quantified, time savings application of ALL-IN-1. That is, if unable to contact someone by phone, the user either left a message for the person to return their call or conveyed the information by way of ALL-IN-1.

• reducing paperwork and paper handling

The number of mail message files created by the 60 pilot users over an eight month period numbered approximately 11,340 (this corresponds to an average of 23.6 messages/person/month or a little over one message/person/day). The estimated average number of copyees/message is approximately two. Also, 95% of the messages were one page in length. the number of pages of paper saved are thus:

$$\begin{aligned} \# \text{ of pages of paper saved} &= 11,340 \times 3 \\ &= 34,020 \end{aligned}$$

• reducing the frequency and volume of photocopying

If one had to photocopy the two paper copies in the example above, assuming a 5 minute average photocopying time, the following manhour savings would result:

$$\begin{aligned} \text{Photocopying time savings} &= 11,340 \text{ messages} \\ &\times 5 \text{ minutes/message} \\ &= 56,700 \text{ minutes} \\ &= 945 \text{ manhours} \end{aligned}$$

Because the ALL-IN-1 system only saves the original text regardless of the number of copyees, this is a more cost effective way of disseminating information.

• reducing the time spent in searching for and retrieving information

A few of the users who made use of the electronic filing cabinet functions reported that the "Index" function was valuable in retrieving previously stored information.

OTHER RESULTS

Three specific needs were identified as vital to the successful implementation and acceptance of AOSS on a plant wide basis:

- **The need to devote resources and to develop plant wide training in office automation procedures and to demonstrate how ALL-IN-1 can be used most productively in the use of those procedures.**
- **The need to devote resources to the customization of the ALL-IN-1 software product, its forms, and "Help" menus for specific end user suggested applications.**
- **The need to allocate Application Support resources to identify, analyze, and implement ALL-IN-1 applications for a variety of specific end user needs.**

ADDITIONAL REQUIREMENTS AND NEEDS

TRAINING

A needs assessment was conducted to identify both training and non-training needs. The major training needs that were identified are as follows.

- The steps for connecting and disconnecting from the system, depending upon whether the Mound Local Area Network (MOLAN), direct or dial-up connections are used, need to be documented.
- Instructions for setting up the emulator software need to be provided.
- Training in the use of the full array of electronic messaging features needs to be provided.
- The ready availability of office automation tools does not mean that these tools will be used and if used that they will be used effectively. Education and training in electronic office procedures and protocols need to be developed.
- Training in the use of the editor, WPSPLUS, needs to be developed.

SYSTEM SOFTWARE

During the course of the pilot project, a number of requirements were identified relating to system software. These are summarized below.

- The main menu contains too many confusing and irrelevant choices. The main menu needs to be re-designed to make it more meaningful for the Mound

environment.

- The menu's that are modified have associated "help" forms providing varying levels of detail to explain various menu options. These "help" libraries need to be modified, as appropriate.
- A shared library of often used forms (for example, expense reports, ADP justification forms - the 7772, purchase requests, and so on) needs to be developed and made available.
- An easy to use, menu driven, file transfer procedure to transfer files or ALL-IN-1 messages to and from the PC's needs to be developed.
- Work station printers, including single sheet printers such as the Diablo printer, should be supported by ALL-IN-1.
- While on-line help and training provide extensive support in most of the ALL-IN-1 functions, many of the DEC keyboard references do not have a match on the IBM PC keyboards. The help libraries and training scripts need to be changed to reflect appropriate keyboard references.
- The mound phone book needs to be stored and made accessible via ALL-IN-1.
- During the short period of the pilot study (less than one year) more than 10,000 documents have been generated by the pilot users and stored on the system. While most of these documents are less than one page in length and therefore use little disk storage, in aggregate they are beginning to cause some access delay. A procedure needs to be developed that will archive (that is, remove from on-line storage) electronic mail messages and all references to messages older than some established and published time period (for example, 3 months).
- A number of system programming functions will be facilitated by the installation and use of DATATRIEVE, the DEC database management system. This needs to be done.
- Presently, users have unlimited disk storage privileges available to them. Disk quotas need to be established to prevent exhaustion of the on-line disk storage capacity.

- Until the subscriber base is substantially larger, paper mail will be commonly used. The software is designed to ask if paper mail is desired when it cannot identify an addressee. However, when the Answer (A) function is used, the paper mail addressees are not identified by name.
- Resources need to be allocated to the optimization of ALL-IN-1.

multiple disk controller units will minimize this problem. Multiple storage control units have been ordered and are scheduled for installation during December 1985.

APPLICATIONS SOFTWARE

Several applications software projects were suggested during the pilot study. The initial focus was on the development of various boiler-plate (template) documents and associated entry forms. Two of the application projects are summarized below:

- Reimbursable order forms that are issued by Sandia and that contain cost and schedule information on various components need to be placed on ALL-IN-1 for ease of communications among the contractors.
- Monthly status reports on various reimbursable projects are prepared by the project managers and communicated to a variety of users. Boiler-plate forms need to be developed and entry screens need to be designed to support this communication function.

HARDWARE

- Availability of physical memory is the single most important aspect of providing adequate response time. The existing 12 Megabyte processor should accommodate approximately 25 simultaneous users before response time degradation occurs.
- The connection of pilot users directly to the VAX processor by way of the VAX Communication boards proved to be unreliable and resulted, on several occasions, in individual users being unable to access the system. Terminal servers need to be used to provide a logical as opposed to a hard-wired physical connection to the system. The Mound Local Area Network (MOLAN) will provide this type of connection.
- Disk storage controller, disk storage power supply, and disk drive problems resulted in several instances of system shutdown during the prime operating period of 7:00 A.M. to 5:00 P.M. Clustering the processor with

GENERAL

General requirements that were identified are as follows:

- A manageable list describing all of the ALL-IN-1 functions needs to be prepared. How these functions can be accessed also needs to be concisely displayed.
- Documentation on "Getting Started" and identification of User Manuals needs to be provided.
- The IBM PC keyboard is difficult to use because of inadequate template designs to show the large number of functions that the keyboard can perform. This is further compounded by the fact that the IBM PC's must run a DEC emulator to communicate with the AOSS. A well designed multiple color template needs to be provided.
- A printer needs to be located in the mail room for printing paper mail addressed to individuals who are not subscribers to AOSS.
- Anyone connected to ALL-IN-1 via dial-up lines will experience delays when using a 1200 Baud Modem. Modems capable of operating at 2400 Baud or higher need to be evaluated for remote use.
- The availability goal of ALL-IN-1 during the hours of 7:00 AM to 5:00 PM on Monday through Friday, considered the prime period, will be 99.9%. Therefore, any maintenance on hardware or software, requiring the shutdown of the system(s) will not be performed during this period.
- Backups of end user mail messages and other files stored in individual electronic file cabinets will be performed twice a day, at noon and at 4:00 PM. Full system ("Image") backups will be performed nightly.

- The office automation software product uses WPSPLUS, a full feature document processing system, as its default editor. WPSPLUS is also available for the IBM PC. A feasibility study needs to be conducted to evaluate providing MultiMate as a VAX resident editor, providing a translator for WPSPLUS/MultiMate documents, and using WPSPLUS on PC's.

BACKGROUND

MRC - MOUND ENVIRONMENT

The MRC - Mound office automation environment is characterized by a mature and advanced use of existing and newly emerging office automation technologies. This sophistication is evident in the early selection and widespread acceptance of both hardware standards (viz. the IBM PC) and standardized professional work station software (LOTUS 123, MultiMate and dBASE III). The presence of the professional workstations created a favorable environment and also presented some unique challenges for the implementation of the Automated Office Support System.

AOSS

The AOSS is a computer-based system designed to improve the way information is handled in a contemporary organization. The use of AOSS can improve overall office productivity. The AOSS is defined by its hardware, software and communication components. The hardware and software consist of a Digital Equipment Corporation (DEC) VAX 11/780 computer running the ALL-IN-1 office automation software product. Communications to the VAX consisted initially of direct, twisted-pair (telephone cable) connections and connections by means of the MOLAN when that pilot became available.

ALL-IN-1

At the time the pilot project started, Version 1 of the ALL-IN-1 software product was being distributed for general use. Mound, however, had an opportunity to be an External Field Test site for Version 2 of the office automation software product. After an evaluation of the two versions, a decision was made to install External Field

Test Version 2 (EFT V2) of ALL-IN-1. During the course of the pilot project, three EFT versions were installed before formal release of Version 2 was made available. The pilot users thus had to live with a number of bugs until formal release of ALL-IN-1.

WHAT ALL-IN-1 DOES FOR THE USER

Most of the computer software used in an office handles fixed, regular, and structured transactions very well. For instance, payroll, general ledger, inventory, and other software are used for routine work. However, the more unstructured and variable aspects of a user's work are carried out largely without the help of a computer. Aside from work that requires the judgment and experience of a professional, much of this work involves tasks such as gathering or transmitting information, keeping track of assignments, filing, making copies, filling out forms, and maintaining a calendar.

In fact, a user may spend much of the workday on secondary or support activities, including many that might be considered administrative and clerical. These tasks may be essential, but performing them is the biggest drain on a user's productivity. ALL-IN-1 has been developed to increase end user productivity by reducing to a minimum the need for them to do this kind of secondary work, and by speeding the performance of that minimum.

TARGET USER GROUP

A target user group was identified from a vertical portion of the manufacturing organization. The group consisted of the two product managers responsible for production activities and their supporting functional contacts. This group was selected because their information handling requirements were perceived to be important to the organization and any facilitation of communications would provide visible productivity improvement.

ENHANCED COMMUNICATIONS

A majority of the target users felt that communications were greatly enhanced through the use of AOSS. One major observation was that the planned presence of a larger AOSS subscriber base would further increase the number of successful communications contacts. The target user group initially numbered 30 individuals and grew during the course of the pilot project to some 60 users. An additional 50 users were added during the course of the project for a variety of support and training purposes.

TARGET USER SURVEY

The initial users were surveyed to obtain information about each user and about the group as a whole in six different areas:

- **Access:** The difficulty in getting hold of others for exchange of information

- ⊗ **Intensity:** The level, importance, urgency, and quantity of information handled
- ⊗ **Communication Contacts:** Identification of the organizational level at which most communication is occurring
- ⊗ **Perceived Productivity:** The perceived efficiency and quality of the work of the professionals
- ⊗ **Activity Profile:** A quantitative picture of the typical workday of the professional
- ⊗ **Computer experience:** The level of experience each professional has with a computer and a keyboard

System Administration. Such things as the development of ALL-IN-1 forms, special menu selections and other custom tailoring of the software, and other computer systems and programming considerations needed to be worked out. Work stations, printers, storage devices, communications devices and other hardware is support of ALL-IN-1 had to be provided. Also, procedures for the smooth daily operations of the ALL-IN-1 system had to be developed.

Communications Consultant. A communications consultant on the Pilot AOSS project was selected from the MOLAN team.

User Representation. The key users supporting manufacturing were two product managers. The target user group consisted of supporting manufacturing personnel selected by the two product managers.

IMPLEMENTATION TEAM

Early in the pilot project, an implementation team was formed to act as:

- ⊗ Steering group
- ⊗ Sounding board
- ⊗ Goal-setting body
- ⊗ Project monitor

The team also made sure that upper management was kept informed during the pilot project of the implementation.

TEAM MEMBERSHIP

Six key areas were considered and represented on the implementation team:

Administrative Leadership. This was the focal point for all project activities within Mound. The Project Leader made procedural reviews and changes, promoted and presented the project to upper management, and was accountable for the implementation.

Project Coordination. The project coordinator developed the pilot project implementation plan, including the schedule and coordinated all implementation resources. He also developed tools for measuring how effectiveness and productivity will be judged, developed the specifications for the hardware and software, developed the acquisition plan, and managed the installation of all hardware and software products.

Education. Training materials and initial pilot user training sessions needed to be devised, scheduled, delivered, and followed up on. The in-house training and assistance team was an important part of the implementation.

Sue Ellen Franklin
Digital Equipment Corporation
Maynard, Massachusetts

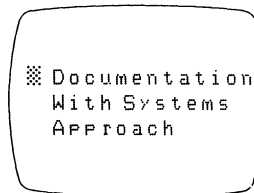
ABSTRACT

This paper discusses major issues facing user communications in the office marketplace. Traditional documentation sets are no longer sufficient to meet the needs of this market. User communications specialists must develop new strategies and new approaches. This paper identifies five such issues and discusses what Office Systems Documentation is doing to address them.

Office Systems Documentation (OSD) is a group of writers, editors, and publications personnel responsible for providing documentation for several of DIGITAL's office automation (OA) products. Our goal is to provide accurate, useful, and timely user communications packages for Business and Office Systems Engineering (BOSE) systems.

Working with OA software, both as users and as documenters, has made us aware of the importance of documentation and of our obligation to produce a unified, coherent user communications package rather than separate pieces of hard-copy documentation, on-line Help, and on-line training. To meet that goal, we have had to identify, define, and resolve some of the major issues facing communications specialists today. This paper addresses five such issues:

- Documentation for systems, not products
- Definition and role of the user interface
- Task orientation
- Integrated user communications
- The "user friendly" puzzle



Documentation for Systems, not Products

An important challenge facing communications specialists today is the need to develop a *systems approach* to documentation. The initial documentation efforts for ALL-IN-1 and WPS-PLUS were designed to teach us, as communicators, how to present a system as a system (providing solutions to business problems) and not as a collection of products. We needed to learn how to deliver to the customer a single, organized instructional package.

What Did ALL-IN-1 Teach Us?

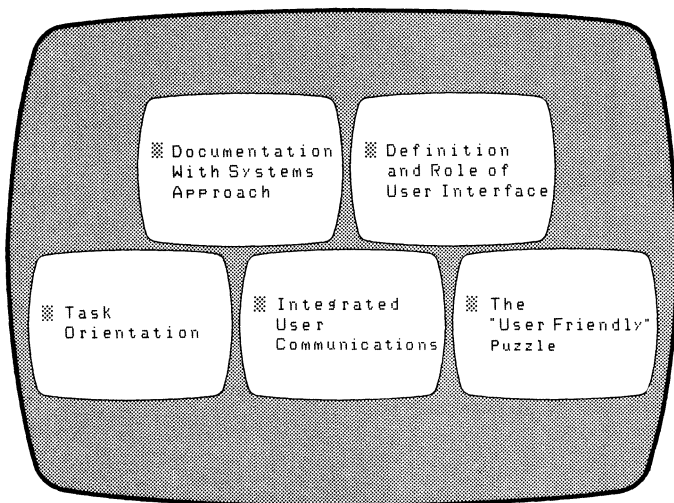
The most important, single fact about ALL-IN-1 is that it is a customizable, open system. This means the customer can:

- Change the appearance of menus and forms
- Add or remove menu options or commands
- Change the way a menu option or command operates
- Rearrange options on different or new menus
- Add or remove applications or entire subsystems
- Remove or redefine keys

In other words, the customer can alter ALL-IN-1's interface and functionality.

System customization impacts most the documents that describe the user interface – the *ALL-IN-1 Office Menu Getting Started Guide* and the *ALL-IN-1 Office Menu User's Reference* – and the on-line Help. Once the product has been customized, the default user documentation and Help no longer describe what the user sees. How can a documentation group deal with this issue?

Anticipating this issue, we developed the Customizable Documentation Kit. This optional kit contains the tools and the information the documenter needs to customize the user manuals and produce new manuals that physically match the default set.



The Kit consists of:

- The *ALL-IN-1 Style Guide* that discusses the ALL-IN-1 V2.0 documentation strategy, standards, conventions, templates, and specific documentation tools used in the manuals. This guide includes sample chapters to assist documenters in customizing the manuals. It provides technical specifications for materials and processes used, including specifications for type, size, and weight of paper, use of color, art, and other visual aids, and packaging information.
- The *ALL-IN-1 Writer's Guide* that discusses the software documentation process, including planning, reviewing, production, and printing. Technical documentation and production terms are kept to a minimum.
- A magnetic tape that contains the full text of the *Getting Started Guide* and the *User's Reference* in DSR format. Because these text files contain the DSR commands that format the documents, we call them *source* files. The meaning of the word *source* in this context is analogous to its use in programming.

The tape also contains master files and command files to ease the rebuilding of customized documents.

With the ALL-IN-1 project, the documentation group faced the challenge of providing a set of books for an interface designed to be changed (customized). That challenge produced the Customizable Documentation Kit. The Customizable Documentation Kit represents a first attempt at defining and delivering to the customer an on-line documentation system.

What Did WPS-PLUS Teach US?

WPS-PLUS presented a slightly different challenge. Here, the basic *core* of functionality stays the same while the interface changes slightly under different implementations and for different hardware configurations.

Currently, WPS-PLUS runs under several versions of VMS, ALL-IN-1, Rainbow/MS-DOS, and IBM's DOS. Because WPS-PLUS is available in so many configurations, there is interest in making the software and the documentation:

- Cost-effective
- Easier and faster to produce
- Fully international and easy to translate

However, none of these goals are as important as presenting to the user a single and consistent approach to learning WPS-PLUS regardless of the implementation.

The first step toward a solution involved two realizations:

- We realized that WPS-PLUS is basically the same wherever it operates. We call this *sameness* the *core* of WPS-PLUS.
- We realized that the differences between environments and versions of WPS-PLUS appear, largely, in isolated areas of functionality.

Therefore, we knew that we would have to devise a documentation strategy containing a core that was largely environment and version independent.

The solution that developed represents a highly modular approach to documentation. It attempts to identify those aspects of functionality that remain the same across all WPS-PLUS systems. As this functionality is the *core* of WPS-PLUS, it forms the basis of what we call the *core* concept in WPS-PLUS documentation.

This process creates general purpose, reusable modules of text and ensures consistent presentation of the software across multiple systems. It gives writers greater flexibility in building systems of communications products and makes it easier for customers who

are translating systems or putting together systems from DIGITAL and third-party components. This topic of "open" or "loosely-coupled" systems is very closely related to other important (*hot*) issues for both software and documentation: consistency, customization, and integrated user communications.

Core modules may vary in size from individual phrases and sentences to entire books. An example of a book that is more than 90% core is *WPS-PLUS List and Sort Processing*. At the other extreme, primers, tutorials, and installation guides (because they are system-dependent) are likely to have less core material. Users see the simplest illustration of this concept in the WPS-PLUS product names: *WPS-PLUS/VMS*, *WPS-PLUS/ALL-IN-1*, *WPS-PLUS/Rainbow*, and so on. The core name is the same in all three cases. Only the appendage that describes the /ALL-IN-1, and /Rainbow (or any of the other PC-based products) as non-core elements in this case.

The issue becomes more complex when dealing with functionality differences. For example, *WPS-PLUS Editor Functions* documents the WPS-PLUS editor and every version of the manual contains the same information – except for specific references to VMS, ALL-IN-1, or PC-based features.

More specifically, *WPS-PLUS/Rainbow Editor Functions* documents WPS-PLUS on a personal computer with floppy disk drives, and the user must be aware of the physical location of documents. To select a document, WPS-PLUS/Rainbow users must enter B: before the document title (if the document is stored on Drive B:). The B: preface is called the path name of the document.

As a result, *WPS-PLUS/Rainbow Editor Functions* reminds users to include the path name when they select a Library or Abbreviation document. This type of difference appears only in *WPS-PLUS/PC-based Editor Functions* and not in the VMS or ALL-IN-1 versions of this manual.

The following equations may also help make these concepts more concrete:

Core Software = Functionality that is always present, regardless of the version of the software or the operating environment

Core Documentation = Documentation of that functionality

Non-Core Software = Functionality that varies across operating environments

Non-Core Documentation = Documentation of those variations

The core approach to documentation is ideal for systems because it:

- Gives consistency in organization and format to users migrating to and from different operating systems or implementations
- Factors out generic material to create general purpose, reusable modules of information
- Presents to the user a single and consistent approach to learning the product regardless of the implementation
- Eliminates needless repetition in instructional material
- Captures and preserves well-designed, written, and tested modules of information

Since core documentation clearly benefits everyone, it is important to continue to produce and extend this movement toward user communications packages that are increasingly modular and generic (core). This approach gives users greater consistency in the same way that software achieves consistency from modular and generic code. We should be able to produce core documentation as long as engineering produces *core* products.

The Definition and Role of the User Interface

The user interface is critical to every product because it is through the interface that the user comes to know the product. Often described as that point where man and machine meet, it is the medium through which the user receives information about the software's functionality. Particularly in the Office Automation area, the user interface IS the product.

With this in mind, we define the user interface as:

- Product interface (menus, forms, screen design, prompts)
- Documentation
- On-Line Help
- Computer-Based Instruction (CBI)

During the development cycles for ALL-IN-1 and WPS-PLUS and other office automation products, the documentation group is able to provide considerable input in all these areas. We are directly involved in the writing of Help frames for both ALL-IN-1 and WPS-PLUS and serve as consultants in CBI and user interface development. We are beginning to write and be fully responsible for our CBIs. Increasingly, every effort is being put forth to make the on-line information consistent with and complementary to the hard-copy documentation.

The Product Interface

The software interface itself comes under continual scrutiny during the development of BOSE products. Important design considerations are that the products be predictable in behavior and consistent in the presentation of menus, forms, screens, error messages, and prompts.

This approach is basically top-down and menu-oriented. It minimizes memorization by presenting a list of choices available to the user at each step in an activity. The user is further assisted through the use of:

- Descriptive phrases and mnemonics
- A small number of form types
- A small number of universal options that may be invoked from any subsystem
- In ALL-IN-1, an open, customizable architecture that may be personalized to suit each user

ALL-IN-1 and WPS-PLUS also give users the ability to create User-Defined Procedures (UDPs), which are documents used to store frequently invoked commands or keystrokes. When the user invokes a UDP, the system executes this series of commands or keystrokes automatically. In this way, the system *watches and remembers* a sequence of steps.

The overall objective in the design of user interfaces is to apply a consistent set of simple rules and basic design principles.

Documentation

Everyone knows what hard-copy documentation is. The important questions today in user communications are: What is on-line documentation? Is it on-line files in a consistent format (MEM, for example) that the user can print on demand? Is it some type of on-line assistance? Is it some type of videotext information base? What is the relationship between hard-copy documentation, Help, CBIs, and other forms of on-line information?

For ALL-IN-1, OSD defined on-line documentation to be an optional tape containing RNO and MEM files organized and formatted exactly as they were for the writers on the project. With this tape, we supplied a book containing all the rules, guidelines, and standards that governed the printed, "official," default documentation.

We arrived at this definition of on-line documentation after careful study of the product requirements for documentation and the ALL-IN-1 user environment. We were asked to provide an on-line documentation package that was both flexible and customizable and both modular and integrated. We addressed these requirements in the Customizable Documentation Kit. The Kit:

- Is flexible and customizable. Each chapter is written according to a template or pattern and cross-references are kept to a minimum. This means that customizers can write new chapters according to the pattern provided by OSD or add or delete chapters or portions of chapters as they wish. It is hoped that writers using the templates will find that they are structured enough to provide a general pattern and open-ended enough to allow for individual creativity.
- Is modular and integrated. Each chapter is a single RNO file on the tape. These modules can be edited individually and reassembled (integrated).
- Uses documentation tools and editors available on VMS and ALL-IN-1.

Nevertheless, this is a very crude form of on-line documentation.

The following is a more sophisticated but still preliminary list of requirements pulled together to describe what on-line documentation might mean to a group of technical writers, editors, and publishers. Such a system would have the following characteristics:

- Reliability, stability (output is the same as input)
- Revisable content (read/write storage)
- Uses one storage format based on generically encoded data
- Handles any data type (compound documents, for example) and facilitates translation among data types (voice to text, text to voice)
- Allows separation of content elements, document structure, and output style
- Allows extraction of outlines based on document structure
- Allows content-based retrieval
- Allows annotation/review comments
- Allows distributed compound document data base
- Provides snapshots of documents in storage format for customer access
- Provides publications tools (spelling checker, syntax checker, rhetoric analyzer)

OSD is currently examining alternate distribution media (such as laser disk, CDROM) to find a system to meet these requirements and to move toward user communications packages rather than traditional "documentation sets." There is a growing desire to use on-line assistance and CBIs as an appropriate, and in some cases, a superior medium to hard-copy documentation. There is a growing realization that everything doesn't have to be on paper.



The On-Line Help

The on-line Help is continually expanded and improved in BOSE systems. In both WPS-PLUS and ALL-IN-1, you can get Help anywhere, anytime by pressing Gold H. Providing support to both new and experienced users, ALL-IN-1 and WPS-PLUS display context-sensitive messages in a window overlay at the top or the bottom of the screen. Help messages include a list of related Help topics as well.

The current on-line Help system:

- Provides Help on menus and forms
- Provides Help on menu options and form fields
- Provides Help on editing keys and general information topics
- Provides modularity – each Help topic is a single module in a VMS library
- Uses templates
- Builds cross-reference lists automatically

ALL-IN-1 also provides guidelines and templates for those who wish to write their own Help or supplement the existing ALL-IN-1 Help. It may not always be possible to follow these templates exactly, but the more consistent the format, the easier it is to update and customize Help.

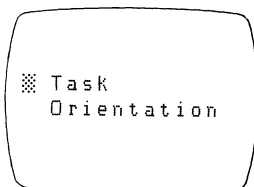
The CBIs

The CBIs for ALL-IN-1 V2.0 and WPS-PLUS embody important new principles. They are fully integrated with the software allowing users to get hands-on practice with product features in task units. In some task units, the CBI pulls in the user's actual files in the execution of a task. For example, while learning to read mail, users read their own real mail messages. If they have no mail, the CBI sends a mail message. This goes beyond simple interaction and *page-turning* and lets users learn how while actually doing. The user interacts with the product instead of interacting with the CBI simulating an interaction with the product. Feedback to responses helps guide users to successful completion of each lesson.

As with the hard-copy documentation, most of the CBIs are keyed to options on the menus. The CBIs are available from separate training menus.

The CBIs are designed to be easy to use. A user needs no prior knowledge of either of these products to activate and run the CBIs.

Finally, a new requirement is being heard in this area. This requirement is that users should be able to access CBIs from within Help, as well as from CBI training menus. This idea opens up new and exciting possibilities for CBIs and Help.



Task Orientation

Although difficult to define, it is important these days that documentation be seen to be *task-oriented*. Whatever the term means, it recognizes that users have jobs to do and the user information package should help and not hinder that process. The user documentation for ALL-IN-1 and WPS-PLUS is task-oriented.

That is, it is task-oriented in as far as the menus themselves are task-oriented and form lists of activities that the user may wish to perform. It might, however, be more precise to say that the user documentation for ALL-IN-1 is *menu and procedure-oriented* and the documentation for WPS-PLUS is *menu and function-oriented*.

More importantly, the documentation and the Help frames for both ALL-IN-1 and WPS-PLUS are written according to established templates (patterns). The templates make the structure of each chapter predictable. This consistency means that the user need become familiar with only one format or style of presentation. Because the user manuals are consistent in format, organization, and writing style, they are also easy to use, update, customize, or translate.

The ALL-IN-1 templates, in particular, are based on the menu structure as the organizing element in the product. They copy the software itself by presenting a top-down view, the user's view, of the system. The reader gets an overview first and then is led through a discussion of each option in increasing layers of detail and complexity.

The templates separate the user interface from the functionality. What the user does (that is, select options from menus to initiate actions, functions, applications) is covered by procedures (series of steps). What happens as a result is described in narrative paragraphs following the procedures.

This procedural approach means that users can quickly scan the *Getting Started Guide* and the *User's Reference* while getting a sense of the number of steps in any activity. The steps provide handy reference points for starting, stopping, continuing, or skipping tasks.

Finally, templates give greater stability to those documenting a customized system. User manuals document the user interface, the area most affected by customization and by differences in operating systems or implementation environments. This material is isolated in the procedures and can be easily dealt with there. Descriptions of functionality are isolated in paragraph modules and can be dealt with separately.



Integrated User Communications

As stated, OSD is chartered to provide documentation for several of BOSE's OA products. We work closely with development to produce technically accurate, useful material. Traditionally, an Educational Services group provides the training and the development group produces the user interface (menus, forms, screens, prompts) and the on-line Help. Therefore, the education of the user was in joint responsibility of at least three different groups with separate priorities, schedules, and constraints.

OSD believes that the user would benefit from a more consistent, complete, and unified approach to the systems that they buy. As we explored alternatives, we coined the term "integrated user communications" (UIC) and developed the following working definition.

Currently, "integrated user communications" means applying a single set of guidelines and conventions to every aspect of the user interface. As stated previously, the user interface is the critical component because it is through the interface that the user comes to know the product. Again, in the Office Automation area, the user interface IS the product.

To review, we define the user interface as:

- Product interface (menus, forms, screen design, prompts)
- Documentation
- On-line Help
- CBIs

Over time, systems will provide additional and improved information carriers (such as videotext, windows, on-line document libraries). These carriers will be added to this growing list of user interface carriers or components.

The goal of integrated user communications is to have all these components cooperate in a unified, coherent, consistent manner to teach the user the system.

It is no longer appropriate to separate information content and the medium used to deliver that information. Instead, we must match the strengths of each medium to the individual's level of expertise, preferred learning style, and type of information that must be transmitted.

What does this mean in more concrete terms? It means that instructional materials best serve the new or novice user by providing basic, step-by-step, task-oriented information. Examples of instructional material include: primers, tutorials, summaries, and executive overviews. In hard-copy documentation, this type of material duplicates the actual screens and menus that the user is seeing on-line and uses examples and illustrations as often as possible.

CBIs fall into this category also and are ideal for new users or for passing one-time-only information. They should be short enough to be quick refreshers, and they get high marks if the user can do real work while learning through the CBI. The ALL-IN-1 CBIs are interactive in this way. As mentioned previously, the user interacts with the product instead of interacting with the CBI simulating an interaction with the product.

IUC means providing better on-line Help material for the more experienced user who has gained some confidence with the system but who, occasionally, still needs help. User guides, quick lookup guides, master glossaries, master indexes, and the whole family of on-line material (prompts, system messages, on-line documentation) are examples. This material should be context-sensitive and designed for the person who uses the system frequently.

Finally, reference material best suits the expert user who wants only the facts and wants them presented in as concise and reference-oriented a format as possible. This user demands technical details, has a high tolerance for dense material, and a low need for illustrations and examples.

IUC also means that, for all users, we produce information packages that adhere to a single set of guidelines and conventions. At its highest level, this set of guidelines and conventions consists of:

- Technical accuracy
- Consistency and clarity in terminology and presentation
- The treatment of a system as a system, not as a collection of products

After technical accuracy, consistency is perhaps the most critical element. Perhaps surprisingly, it is a difficult goal to achieve – even within a single area like documentation. Now, however, we must carry this goal even further until we achieve consistency within and across documentation sets, on-line Help, CBIs and the user interface. The elements identified as critical to the issue of consistency are:

- Terminology (definition of terms)

Terms, particularly those that might be unfamiliar to the user or used in an unfamiliar way, must be defined clearly throughout OSD's user communications packages.

- Word usage

Words relating to the operation of the software must be used consistently throughout OSD's user communications packages.

- Format (presentation) templates

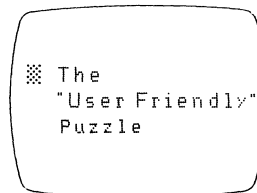
The way information is presented to the user should be consistent across products and across media.

We will increase the use of templates in documentation and in all forms of on-line material. Templates give the user a consistent instructional approach and help produce communication modules that are consistent in format, organization, and writing style. In that way, the user knows what to expect whether reading a WPS-PLUS book or a DECpage book, a WPS-PLUS Help frame or an ALL-IN-1 Help frame.

- Grammatical and stylistic elements

- Conventions

OSD is developing a single list of conventions to be used consistently throughout all office products communications. Specifically, we are moving toward a consistent usage for dot matrix, red print, uppercase, boldface, italics, and on-line prompts.



The "User Friendly" Puzzle

The stated strategic goal of DIGITAL's Business and Office Systems Engineering group (BOSE) is "end user information productivity." OSD is responsible for providing the user communications segment of that goal – greater and faster user productivity.

We have identified and discussed four communications issues in support of that goal: the systems approach to documentation, the definition and role of the user interface, task orientation, and integrated user communications. What has been the point of all this identification and discussion? What puzzle are we trying to solve? We are trying to solve the puzzle of communication. We are trying to discover how to transfer a body of knowledge about a product from our heads to the user's head in the most effective and efficient manner possible. We call this effort, perhaps rather loosely, user communications.

The best way of transmitting knowledge has been debated for centuries. Today, the best ways are called "user-friendly." This attribute is also known as "ease-of-use" and "easy-to-use."

I suspect that whether or not a system and its accompanying user communications package is user friendly is, like beauty, in the eye of the beholder.

I suspect that if users have questions and find the answers easily, they say that the system is user friendly. And, if they do not, they believe the system is not adequately meeting their needs, are slower to learn the system, and are slower to use the system to its fullest functionality. This is not by way of getting us off the hook. We have made progress. A sign of our progress may be that our foreign language translators find our documentation and on-line Help "too friendly" and are insulted by our air of informality.

User communications packages can never make a complex system look simple. ALL-IN-1, WPS-PLUS, and DECpage (to name just three BOSE products) are complex systems. With that complexity, the user gets greater power and functionality. That complexity also demands that the user expend more effort during the learning process.

This complexity also means that we must build ease of use into systems at the design stage. User friendliness is not a concept that can be layered on a product like a final coat of varnish. This means that we must work to design systems that are:

- Simple (not simplistic), intuitive, and logical
- Consistent and, therefore, predictable
- Customizable and open

The messages for us are clear. We have learned much about building user communications packages for office automation products and I believe that we continue to move in the right direction. Learning to document systems, not individual products, defining and clarifying the role of the user interface, orienting toward user tasks, learning to define and build integrated user communications, and struggling to solve the "user friendly" puzzle all support our common goal of increased user productivity.



PERSONAL COMPUTERS SIG



PUTTING THE READER BACK IN MANUALS:
COMPUTER MANUALS AND THE
PROBLEMS OF READABILITY

By
Thomas L. Warren
Department of English
Oklahoma State University
Stillwater, OK 74078

ABSTRACT

A common myth about aircraft is that the plane is ready to fly when the weight of the paperwork equals the weight of the aircraft. That paperwork includes manuals as well as other documents. As experience has unfortunately shown, manuals confirm another myth: When all else fails, read the manual. The novice and experienced users turn to manuals to answer questions and solve problems. Various 800 numbers help, but, in the dark of the night, only the manual is there. This paper examines manuals from the reader's perspective, beginning with a review of how humans process information. It then analyzes ten sample texts using nine readability formulas and a style analysis program. While demonstrating the inherent weaknesses of readability and style analysis results, the study does pinpoint some troublesome areas in current PC manuals. Ultimately, however, it is the user using the manual that determines its value. Writers need to become more aware of this measure of readability.

Psychologists tell us that we mortals have an inordinate fear of many things. High on the list are death, taxes, and public speaking. What the list makers overlook is something that strikes terror in the hearts of computer users--young and old, neophyte and pro, PC or VAX user. I'm speaking, of course, of the technical manual, that harbinger of things unenlightening.

Picture the poor user desperately trying to find out what happened to 35% of a file that disappeared when he searched forward (an event that recently happened to me). Calls to the 800 number don't work. Reviewing the manual led to more confusion, and I still have no idea how "GOTO" in "Select-86" can wipe out so much text.

My point is not to swap horror stories related to poor documentation. Rather, my purpose is to present some recent findings about the design of technical manuals--putting the reader back in the manual. I want to limit my remarks to the PC manuals first, because they are the manuals I know, and second, because of time. What I have to say, however, can apply to other manuals.

I want to divide my talk into three main sections:

1. Reader Considerations
2. Access to the Text
3. Analysis of Ten Texts

READER CONSIDERATIONS

When we learn to write, we often forget that someone must read the material we write. In school, Miss Grundy and Professor Flunkemoften were paid to read our essays, tests, and lab reports. Our families were glad to read letters (usually notes pleading for more money) from the young scholars. But once we leave school, we enter a much different world. Teachers read to verify that the student understands the material; employers and supervisors read because they need the information the document contains.

The same is true with PC manuals. I brought my Rainbow 100+ home in several boxes, as we all do, but one box was full of manuals and programs. As I write this, I count 24 manuals--an awesome sight for my wife who wants to learn to use this "new toy," as she calls it.

When we look at a page from a manual (Figure 1), what do we see? What happens when we read this page looking for information?

Insert Figure 1 Here

We see text surrounded by white space with some words highlighted. If I need the information on this page, I have to start at the upper left-hand corner and move to the right across the line to the end and return

MOVE

The MOVE command moves a block of text from one part of your document to another part. The MOVE command is helpful when you want to rearrange text in your document.

How to Use

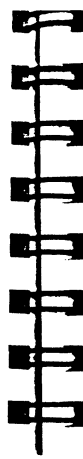
The following steps describe how to use the MOVE command:

1. Set the first pointer (see the POINTER command) at the top of the block of text you want to move.
2. Set the second pointer at the bottom of the block of text you want to move.
3. Move the cursor to the point where you want to put the block of text.
4. Press:

M

The following message displays:

>MOVE: Top Bottom (pointers) From 1 , To



5. After From, press the letter or number of the first pointer. For this example, press:

1

The screen displays the following message:

>MOVE: Top Bottom (pointers) From 1 , To

6. After To, press the letter or number of the second pointer, and the text is moved.

NOTE

Before using the MOVE command, you must know how to use pointers. If the block of text that you move does not start and end at the left margin, you may need to justify the text afterwards.

FIGURE 1. Text7

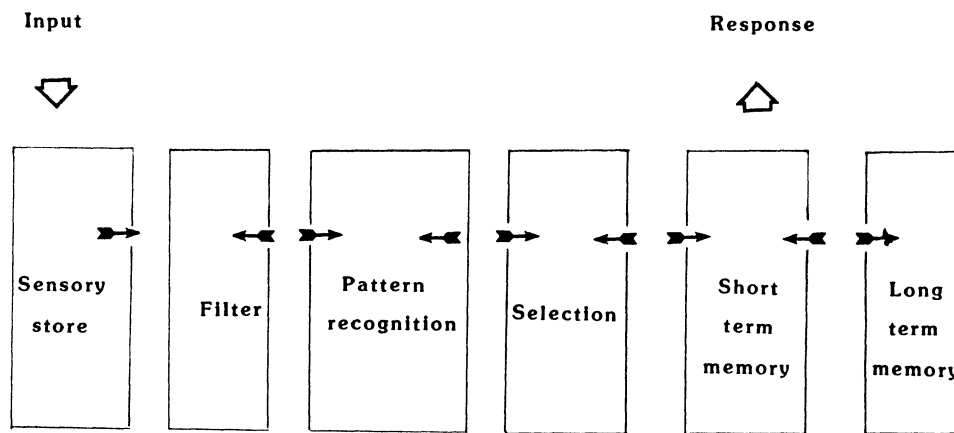


Figure 2. Information Processing Model
(Taken from Stephen K. Reed. Cognition: Theory and Application
[Monteray, CA: Brookes/Cole Publishing, 1982], p.4)

to start the next line. I could scan the page, noting that one set of highlighted words indicates how to use the command. If that is the information I need, I start reading, noting that the lead-in sentence is redundant of the heading, thereby wasting some of my time.

Figure 1 also demonstrates that technical manuals for computer users are not text in the same sense that a novel or newspaper story is text. Readers of manuals are interested in random access of information, not reading from beginning to end. They read these manuals differently than they might other forms of nonfiction prose. Many computer companies (Digital included) recognize this situation and prepare two documents or two sections in one document. One section is to be read through completely by the user (the "Getting Started" section); the other is for random access (the "Reference Manual" or "User's Guide"). For a better understanding of why writers must keep their readers in mind, let me turn to how we read--how we process the information from the printed page.

Information Processing

When we consider information processing, we must also consider a larger issue: Why communicate? One view is especially relevant here: "The primary purpose of human communication is to define and to understand reality so that other human purposes can be achieved" (Rogers and Kincaid, 63). When we find that communication does not allow us to get on with those other purposes, then we have poor communication that leaves us in as much uncertainty as we had before we turned to the communication. If we can assume that communication is to change uncertainty, writers can become better communicators if they know how that change occurs, how a reader processes the text that leads to a response. Figure 2 shows the stages through which the material passes during processing.

Insert Figure 2 Here

What follows is a discussion of the major parts of this model: Sensory store, filter, pattern recognition, and selection. My focus will be on why knowing the processing elements can benefit manual writers and, ultimately, readers.

Sensory Store

When we process a page of text, the signal moves through the visual sense (sight) and pauses slightly before moving along for further processing. At this temporary storage point, many of the signal's characteristics influence the encoding of that signal so that it can pass to the next stage. A rapid analysis of the signal evaluates the line, angle, and brightness of the image; its position on the page; the amount of material in the foreground and background; and color (see Appendix A for a list of design variables). In addition, the analysis includes the layout of the text on the page. For example, spacing becomes critical when looking

for specific pieces of information (the random access of information). Consider how easily the eye can pick up the stimulus from these pages (Figures 3, 4, and 5). Are these easy or hard to decipher?

Insert Figures 3, 4, and 5 Here

The left-hand page in Figure 3 presents the reader with a massive block of text with a blank line following line 5 (8 lines). If the reader wants to move some text, he or she must get through that block, reading left-to-right. The heading is treated differently by the designer (large letters, bold-face, extra space, etc.). The example ("Prepared by") is set off from what comes before and follows, but is not treated differently in terms of type face and style. The rest of the page is linear text plus a figure of what the reader is to assume is a screen.

The right-hand page contains 4 lines of text plus what one must assume is what the screen will look like. Steps are run together with explanation so that the reader must use many eye movements: Text to keyboard to screen to text to keyboard to text to keyboard to screen to text.

Figures 4 and 5 are also solid blocks of text with headings in bold-face, all capitals (Figure 4) or upper and lower case letters (Figure 5), and white space. Past these, the formidable block of text suggests that the reader do something else. That reader is forced to plunge into the text in traditional, left-to-right and top-to-bottom ways until finding the information all the time shifting from text to keyboard to screen to text.

What the three figures suggest is that spacing can play an important role in helping the reader get needed information. Typically, that spacing is of two types: vertical and horizontal. (Material in the following sections adapted from Hartley.)

Vertical Spacing: Space separates one line on the page from another. Titles, headings, subheadings, sentences, and paragraphs have space between them and the next element. This amount of space between lines in the text constitutes 1 unit of space. Adding extra units of space between heading elements isolates them and insures that the eye quickly picks up and separates headings from text. Coupled with the type style (roman, italic, bold), space makes the random access of information easier, reducing the amount of text that the reader must search to find specific information.

Another element in vertical spacing is the end of the body's text at the bottom of the page. Traditional layout fixes the number of lines per page. When pasteup artists reach that number of lines, they begin a new page. Dividing the text based on number of lines often interrupts the syntactic unit--the group of words in the sentence that carries meaning. A complicated sentence that continues on the next page may cause the reader to turn back and forth between the two pages trying to understand it. Having a flexible line count insures that the reader will carry a full

Moving a Block

The block move command (^KV) **moves** all the text in the marked block to the **cursor position**, deleting the original at its old position. If no block is marked when the command is given, or if either marker is hidden, an error message occurs (Appendix B).

The destination may be in the middle of a line, if desired — for example when rearranging sentences in a paragraph. Just put the cursor where you want the block moved to. The cursor is left at the beginning of the moved text.

The beginning and end markers **move with the block** and remain displayed. After inspecting the result, type ^KH to hide the block markers -- both to remove the distraction from the screen, and to protect against block commands typed by accident. If you wish to use the same block markers later, just type ^KH again.

The block move command moves **exactly** the characters you have marked, and does no automatic reformatting. Thus, text reformatting is often required after a move. After rearranging sentences, for example, use paragraph reform (^B, Section 4) to re-establish the margins. You may also notice that you included too many or too few spaces or carriage returns at the beginning or end of the block. These errors are easily corrected with a few regular editing commands.

After a block move, the command ^QV will move the cursor to the place the block came from. It's a good idea to inspect here after moving, as you may have left too many spaces or carriage returns behind, or you may need to reform the paragraph. Note that any place markers 0-9 in the marked block do **not** move with it—they remain at the place the block came from.

For an example of moving a column block, see Figure 6-1.

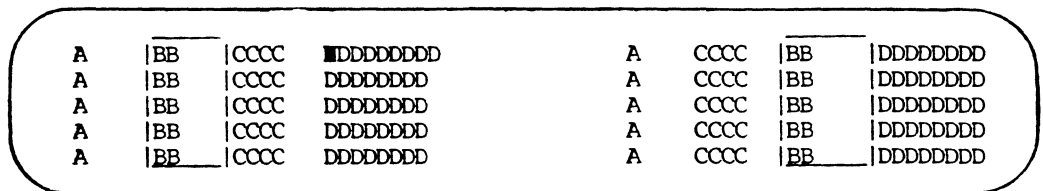


FIGURE 5. Text3

Move

Move

MOVE

Purpose

You use the MOVE command to move a specific block of text from one place to another. MOVE lets you delete the original block of text and insert it anywhere else you want.

Moving a Block of Text

1. Position the cursor at the beginning of the text you want to move.
2. Press the Do key.
3. Press the M key (for Move).
4. Shade the text you want to move.
5. Press the Return key.
6. Position the cursor where you want to insert the text.
7. Hold down the Ctrl key while you press the Insert Here key.
8. Press the Return key.
9. Press the N key (for No) if you want SAMNA to insert the text with its stored format. Press the Y key (for Yes) if you want SAMNA to insert the text with the current format.
10. Press the Return key.

Helpful Information

- The text you move is deleted from its original location. If you want to move a block of text and also retain it in the original file, use the COPY command.
- You can move any amount of text. You shade the text you want to move using the Word, Sentence, Line, Paragraph, Page, File, and arrow keys. However, the File key is not recommended.
- Any marks within the text are moved with the text.
- The text you move is temporarily saved, along with its format, in the TEMP file.
- The TEMP file holds one block of text at a time. Therefore, when you move (or copy) text, SAMNA replaces the contents of the TEMP file.
- If you want to save text stored in the TEMP file after you finish moving or copying, give the TEMP file another name. The text is now safely in the file with the new name.
- You can display and edit the TEMP file.

FIGURE 6. Text9

Copying or Moving Text

You use the copy or move procedure to save a specific block or column of text and insert it in a different location in the file. This is useful when you need to repeat the same information or to copy Format Lines. You can copy text to a temporary buffer or to a stored file.

Copying or Moving a Block of Text

To copy or move an entire block of text:

1. Position the cursor at the beginning of the text you want to copy.
2. Press the Do key.
3. Type the letter C (for copy) or M (for move).

When you move text, it is deleted from its original location. When you copy text, it is not deleted from the original location, and therefore exists twice in your file.

If you are storing the text in a separate file:

- Press the File key.
- If you do not want to use the default file name TEMP, type a file name.
- Press the Return key.

NOTE

If you do not specify a file to store the text in when you copy it, Samna stores it in a temporary buffer. This buffer can hold only about one full page of text. Therefore, if you have a large block of text to copy, you should store it in a separate file.

4. Shade the text you want to copy.
5. Press the Return key.
6. Move the cursor to where you want to insert the copy.

Text Procedures

7. Hold down the Ctrl key while you press the Insert Here key. If you stored the text in a file:

- Press the File key.
- Type the name of the file, unless you used the default file TEMP.
- Press the Return key.

Samna asks:

```
Which format should be used?
Type Y to use the current format. Type N to insert the
stored format.
Will the text be inserted into the current (displayed)
format? Yes/No (N)
Is the text you are inserting a column? Yes/No (N)
```

8. Respond to these questions and press the Return key.

NOTE

To move text with its original format, you must save the text in a separate file.

Copying a Format Line

To copy a format line:

1. Position the cursor immediately below the format line you want to copy.
2. Press the Do key.
3. Press the C key (for copy). If you are storing the line in a separate file:
 - Press the File key.
 - Type the name of the file if you do not want to use the default file name TEMP.
 - Press the Return key.

FIGURE 7. Text9

syntactic unit over to the next page.

Finally, breaking a paragraph at the bottom of the page after one or two lines may also interrupt the syntactic flow. Paragraphs should not begin on the last line of a page nor end in the first line of the next page.

With these thoughts in mind, let's look at the sample pages (Figures 6 and 7).

Insert Figures 6 and 7 Here

Note in Figure 6 how the headings and subheadings are set apart from the text. Figure 7 shows good use of vertical space, but runs the listing to the page bottom and on to the next page. Had the text ended before Step 4, Steps 4-8 could occur on a new page with "Copying a Format Line" starting another new page. Steps 4-8 also constitute a sub-block of commands, separated from Steps 1-3 by the "Note."

Horizontal Spacing: Editors often want a uniform line, justified at the right-hand margin, but the appearance is of a massive black block of text. Consider the effect the sample from the Text3 (Figure 8) has on someone looking for help in moving a block of text.

Insert Figure 8 Here

The text in Figure 8 has a justified right-hand margin, inhibiting to readers looking for specific information. A principle in document design (Felker, et al., 81-82) holds that the more white space you can have on a page, the easier the text is to access. Figure 9 shows an unjustified right-hand margin.

Insert Figure 9 Here

The psychological impact of such an arrangement encourages both random access and feelings of reading progress. The extra white space at the ends of the line reduces the impact of the solid block of text; shorter lines mean that you can read more of the page in a shorter time than with justified format. The unjustified page also means less text per page, so that you have less to scan in looking for the information you need.

In addition, unjustified right-hand margins eliminate the need to hyphenate a word at the end of the line. Hyphenation forces the reader to hold a part of the word in memory until picking up the rest of it on the next line (Figure 10).

Insert Figure 10 Here

(NOTE: I will indicate paragraph and printed lines by a number, a period, and another number. 1.5 means the first paragraph, fifth printed line.) The fifth line from the top of column 1 has a hyphenated word ("with-"). Should the reader not catch the hyphen, the negative ("not"--1.6) could get lost, giving a decidedly different meaning. In addition, if the reader accidentally skips a line (from 1.5 to 1.7), meaning is further confused. A third possibility is to skip from the fourth to the sixth line (1.4 to 1.6).

Another element in horizontal spacing is that with unjustified right margins, you are not obliged to start a sentence at the end of a line, breaking a syntactic unit when the space runs out (see Figure 11, 1.8).

Insert Figure 11 Here

When designers use unjustified right-hand margins, they can determine line length by the syntactic units, allowing the reader to process the units as meaning units and not as fragments.

Filter

Following the processing by the sensory store, the stimulus is now filtered by both psychological and physical means. Preconceptions can be called from long-term memory and affect the stimulus as surely as the physical factors (poor eyesight, for example). Some people have preconceived notions about computers and manuals so that their response to the signal will vary from that of another person who holds another view.

Pattern Recognition

Following filtering, the signal enters pattern recognition. Because communication transfers information and that transfer involves meaning, the mind extracts that meaning from three areas: the word, the sentence, and the organization.

Word (Semantic): When we communicate each word conveys a dual meaning: that meaning found in any dictionary and that meaning associated with the word. Hit, for example, has both dictionary and associative meanings. One may hit a ball, be a hit, take a hit, or any number of meanings you can find in a dictionary (I count 25 in my dictionary [Morris, p. 625]). Slang hit though, has meanings not found in a dictionary (hit, meaning to rob, as in "hit a bank," for example). I often wonder what happens when someone takes the "Hit the Return" literally.

The problem with manuals is that writers may use words their reader does not know (at either level); may know, but in a different context; or may associate different meaning from what the writers meant. I have often wondered why manuals tell us to "delete" characters rather than "erase" them (Figure 11, 1.2). My dictionary (Morris, 349) tells me that delete means "To strike out or cancel; omit . . ." and that it comes from the Latin delere, meaning "to wipe out, efface." Erase, on the other hand, means "1. To remove; rub, wipe, scrape, or blot out; efface. 2. To remove all traces of . . . 3. Slang. To get rid of (a person) by murder." Erase comes from Latin "eraddere, to scrape out, scrape off . . ." (Morris, 443). My point is that the metaphoric meaning writers want is much closer to erase than delete.

Paragraphs 2 and 3 of Figure 11 offer additional interesting case studies in why computer manuals are hard to read. Notice how sentence 1 of paragraph 2 (2.1-2) has 3 punctuation marks: a comma, a dash, and a period. These marks are familiar to everyone, yet put them together and you have a very hard sentence to process. The comma and "if desired" on the following line are standard structures. The problem is the

Moving a Block

The block move command (^KV) moves all the text in the marked block to the cursor position, deleting the original at its old position. If no block is marked when the command is given, or if either marker is hidden, an error message occurs (Appendix B).

The destination may be in the middle of a line, if desired — for example when rearranging sentences in a paragraph. Just put the cursor where you want the block moved to. The cursor is left at the beginning of the moved text.

The beginning and end markers move with the block and remain displayed. After inspecting the result, type ^KH to hide the block markers -- both to remove the distraction from the screen, and to protect against block commands typed by accident. If you wish to use the same block markers later, just type ^KH again.

The block move command moves exactly the characters you have marked, and does no automatic reformatting. Thus, text reformatting is often required after a move. After rearranging sentences, for example, use paragraph reform (^B, Section 4) to re-establish the margins. You may also notice that you included too many or too few spaces or carriage returns at the beginning or end of the block. These errors are easily corrected with a few regular editing commands.

After a block move, the command ^QV will move the cursor to the place the block came from. It's a good idea to inspect here after moving, as you may have left too many spaces or carriage returns behind, or you may need to reform the paragraph. Note that any place markers 0-9 in the marked block do not move with it—they remain at the place the block came from.

For an example of moving a column block, see Figure 6-1.

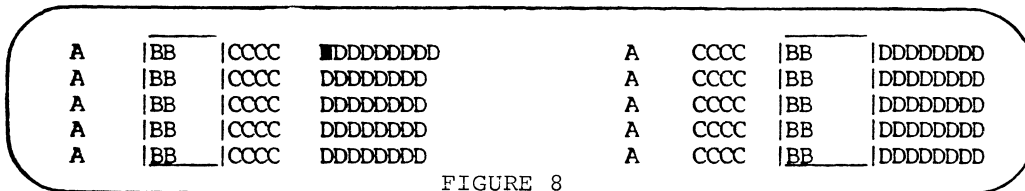


FIGURE 8
Text3

FIGURE 9

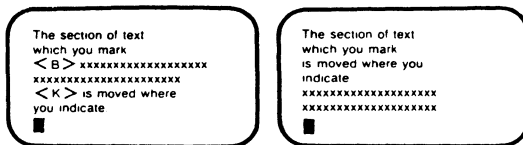


Moving Blocks

Use ^KV to move all text in a marked block to the current cursor position. The remaining text will move up to fill the space left by the moved block.

The destination of your block may be anywhere in the text area—between paragraphs or in the middle of a line. Place the cursor wherever you want to move the block, and press ^KV.

Here is an example:



Before

After

HOW MANY CHARACTERS MOVE?

The beginning and end markers move with the block and remain in the display. After the move, use ^KH to hide the markers, both to remove the distraction from the screen and to protect against block commands typed inadvertently. Place markers (0-9) in the marked block do not move with the block; they remain at the block's former place.

FIGURE 9.
Text4

Moving Blocks

In preparing letters, reports, lists, tables, articles, or books, you will frequently find it necessary to reorganize the draft as you are polishing it. One of the great advantages of a word processor is the freedom it gives you to rearrange text without having to cut and paste everything you want to move. WordStar can move rows of text, and later versions can also move columns.

STEP 1. TYPE `CTRL K B` TO MARK THE BEGINNING OF THE BLOCK

```

B:BLOCKS PAGE 1 LINE 1 COL 01          INSERT ON
-----|-----|-----|-----|-----|-----|-----|-----|
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
    
```

Move the cursor under the first letter in the block you want to move. Type `^KCB` and a control character (B) will appear on the screen just to the left of the letter under which you positioned the cursor. (If you are marking a column see page 90.)

STEP 2. TYPE `CTRL K K` TO MARK THE END OF THE BLOCK

```

B:BLOCKS PAGE 1 LINE 1 COL 01          INSERT ON
-----|-----|-----|-----|-----|-----|-----|-----|
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
    
```

Move the cursor one space to the right of the last character in the block you want to move. If you ended the block with a carriage return, which will be indicated by a < flag in the far right column, place the cursor at the beginning of the next line so the carriage return will be moved along with the block. (If you are marking a column see page 90.)

STEP 3. MOVE THE CURSOR ONE LINE BELOW THE LINE ON WHICH YOU WANT TO MOVE THE BLOCK

```

B:BLOCKS PAGE 1 LINE 13 COL 01          INSERT ON
-----|-----|-----|-----|-----|-----|-----|-----|
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
    
```

When you move the block, the first line of text will appear one line ABOVE the line on which you place the cursor.

STEP 4. TYPE `CTRL K V` TO MOVE THE BLOCK

```

B:BLOCKS PAGE 1 LINE 9 COL 01          INSERT ON
-----|-----|-----|-----|-----|-----|-----|-----|
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
    
```

The block will move from its former position to one line above the line on which you placed the cursor. The space vacated by the block you moved will be filled up by the rows of text below the block, which will move up into this space.

STEP 5. TYPE `CTRL Q V` TO RETURN TO THE POSITION FROM WHICH THE BLOCK WAS MOVED

```

B:BLOCKS PAGE 1 LINE 1 COL 01          INSERT ON
-----|-----|-----|-----|-----|-----|-----|-----|
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2 BLOCK 2
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1 BLOCK 1
    
```

The text in the space vacated by the moved block, as well as the moved block itself, might require paragraph reforming with `^B`. To return the cursor to the position from which the block was moved type `^QV`.

FIGURE 10. Text5

Moving a Block

When you want to move a block of text from one location to another in a document, label the block and use Del to remove it from its original location. WRITE puts it in the block buffer. Then move the cursor to the new location and press F6. WRITE brings in a copy of the block in the block buffer and inserts it at the cursor location (again, the block remains in the buffer).

For example, suppose you want to move the "prepared by" section of the annual report so that it appears on the financial statement page of the annual report. Press Home and then the down arrow to move to the line

Prepared by

Label the section by pressing F5 and then press the Enter key five times. The screen looks like this:

7-6

```
ANNUAL REPORT
1982
HQUI CHARTER TRIPS

Prepared by
South of Reno, 8 Branch
218 Main Street
San Rito, California

-----
annual          Liability          95 Full   Line 18 of Page 1
F1=help
```

Then press Del to remove the lines from that location, as shown below:

```
ANNUAL REPORT
1982
HQUI CHARTER TRIPS

-----
annual          95 Full   Line 14 of Page 1
F1=help
```

FIGURE 11. Text2

Next, move the cursor to the financial statement on page 3 using the down arrow or PgDn key. Insert 6 blank lines so that the words Financial Statement appear at the top of page 4. Page 4 now looks like this:

Moving a Block

The block move command (^KV) **moves** all the text in the marked block to the **cursor position**, deleting the original at its old position. If no block is marked when the command is given, or if either marker is hidden, an error message occurs (Appendix B).

The destination may be in the middle of a line, if desired — for example when rearranging sentences in a paragraph. Just put the cursor where you want the block moved to. The cursor is left at the beginning of the moved text.

The beginning and end markers **move with the block** and remain displayed. After inspecting the result, type ^KH to hide the block markers -- both to remove the distraction from the screen, and to protect against block commands typed by accident. If you wish to use the same block markers later, just type ^KH again.

The block move command moves **exactly** the characters you have marked, and does no automatic reformatting. Thus, text reformatting is often required after a move. After rearranging sentences, for example, use paragraph reform (^B, Section 4) to re-establish the margins. You may also notice that you included too many or too few spaces or carriage returns at the beginning or end of the block. These errors are easily corrected with a few regular editing commands.

After a block move, the command ^QV will move the cursor to the place the block came from. It's a good idea to inspect here after moving, as you may have left too many spaces or carriage returns behind, or you may need to reform the paragraph. Note that any place markers 0-9 in the marked block do not move with it—they remain at the place the block came from.

```
Financial Statement
September 30, 1982

Assets                                Liability
Land (1) 3,462,000    Accounts Payable 862,000
Buildings (2) 635,000    Loans 5,275,000
Aircrafts (3) 7,219,000    Taxes Paid 217,000
Equipment 792,000
Cash (4) 1,182,000

Total Liabilities 6,354,000

-----
192 Full   Line 26 of Page 4
```

FIGURE 12. Text2

dash following "desired." Normally, a dash introduces elements that further amplify an idea in the group of words coming before it (the subject or verb or object/complement or a combination). The "if desired" is a commonly used tag to indicate that an option exists. Combine both parts and you get, in effect, an amplification option. The problem is what did the writer mean to amplify: the "desire?" The "destination"? Which? The reader must pause to infer the reference.

Paragraph 2 offers another example of the typical problems found in a manual. Sentence 1 offers information/explanation; sentence 2 offers instruction (speaking directly to the reader through "you"); and sentence 3 returns to information/explanation rather abruptly. What is missing is a transition that tells the reader why sentence 3 has an important piece of information/explanation. The reader is likely to respond "So what?" to that third statement.

In paragraph 3, the second sentence also offers 3 kinds of information:

1. "After inspecting the results" tells the reader that she or he has something that should have already happened. The writer is telling the reader to do something after the reader was assumed to have already done it.
2. From ". . . type ^KH . . ." to ". . . markers . . ." tells the reader to do something.
3. From ". . . --both to remove . . ." to the end tells the reader why the action described in the second part is necessary.

Individually, each purpose is valid (although you must admit that telling someone to do something after assuming that they have already done it is a bit odd). Combining these 3 sentences leads to confusion. The reader reads part 1 and thinks, "Oh, I should have done something." Part 2 opens with the command [You] "type" leading the reader to infer that this may be the action to be performed (something like "To inspect, type . . ."). The reader is liable to skip over inspecting when typing "^KH" thereby losing for the moment the right to inspect.

Language (semantics) can also be a problem with these 3 paragraphs:

1. Why did the writer use "beginning" and "end"? Using "ending" would make the words parallel--a way of showing how ideas relate to one another. Using "end" could confuse the reader because "end" could be interpreted as a command form of the verb (see 3.1).
2. Why did the writer use "destination" instead of "placement" or "location"--both of which save a syllable and processing (see 2.1).
3. What is the meaning of "same" (3.4)? Is the reader to infer that the previous block markers are erased (deleted) to be used at the new location? ("Select-86"

has a finite number of markers you can use to move text.)

4. Why modify the command "type" with "just" (see 3.4)? Such a word suggests talking down to the reader. It, along with such modifiers as "obviously," "merely," and "simply" suggest that the writer is lecturing to an uninformed reader.
5. The writer uses "move" as a modifier ("block move command") and a verb ("moves") in 2.1. Using one word to function two ways in a sentence is often confusing. (What does this question mean: "Who input the input input?") What is wrong with words such as "shifts," "changes," "transfers," "transposes," "conveys," and the like? Certainly "transfers" and "transposes" are more elegant than "moves," and our writer seems to prefer elegance over simplicity (how else account for "destination"?).

Multiply these problems by the hundreds of pages this manual has, and you get some idea of why readers look in the manual as a last resort

Sentence (Syntactic): The predominate sentence pattern in English is the group of words with a subject, a verb, and, often, an object or complement. Any of the elements may be multiple, but each subject slot contains subjects, verb slot verbs, and object/complement slot objects/complements. Our minds are attuned to that pattern and can almost understand a sentence regardless of the words used if it follows that pattern. For example, Lewis Carroll's "Jabberwocky" opens

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe (Carroll)

And it almost sounds right. Consider next this example in prose:

The concept of the unit includes a new structure. The nature of this function supports the use of basic inputs. This concept also supports other factors via a system approach. The system status, not the structure status, requires that aspect. In terms of size, the effort is small. But in terms of function, it provides the support required. To address these goals involves using all the system's support function. Among the factors, of course, are unit input functions. The support, which uses a system approach, affects these goals. These too require new concepts. We shall reach these goals. (Cohen)

It likewise almost sounds right. The mind recognizes the pattern and tries to gain meaning from it. We know that "John hit Bill" differs in meaning from "Bill hit

John" because of the position of the words in the sentence (its syntax). The point is that the reader expects sentences to fall into commonly recognized patterns. One study supports this expectation (Christensen). Professional writers of all sorts of prose use a very high percentage (98.5%) of basic sentences or that basic sentence with a short opener. Basic, however, does not mean primer sentences ("See Dick run).

Let's look at a sample from Figure 11:

The block move command moves exactly the characters you have marked, and does no automatic reformatting. Thus, text reformatting is often required after a move. After rearranging sentences, for example, use paragraph reform (B, Section 4) to re-establish the margins. You may also notice that you included too many or too few spaces or carriage returns at the beginning or end of the block. These errors are easily corrected with a few regular editing commands.

The first sentence is really 3 sentences. The first runs from "The block . . ." to ". . . exactly the characters . . ."; the second is ". . . you have marked . . ." with the connector that omitted; and the third from ". . . does no . . ." to ". . . re-formatting" with the subject understood (It). Table 1 shows the structure of the sentences in that paragraph, their purpose, and a comment.

Table 1. Analysis of Sentences in Paragraph 4

SENT. NO.	PATTERN*	PURPOSE	COMMENT
1	S-V-0+[Connector]-S-V+"and"+[S]-V-0	Inform	Leads to action
2	[Connector]+S-V-C	Inform	Prepares reader to act.
3	Introductory+[S]-V-0	Inform+ Action	
4	S-V+"that"+S-V-0	Inform	Prepares reader to act.
5	S-V-0	Inform	Informs of action (but doesn't tell what to do).

The table shows that this paragraph contains sentences that are rather complex. Readers in a hurry to get information will not want to puzzle over the syntax.

If we look closely at the paragraph, we can find a number of problems, especially if we look at both the semantic and syntactic meanings. I mentioned above the problems

with one word used in two ways ("move" as noun and verb in sentence one of the sample). The main subjects of each sentence can lead to confusion: Sentence 1, "block move command", with two verbs ("moves" and "does"); sentence 2, "text reformatting"; sentence 3 "[you]"; sentence 4, "You" and "you"; and sentence 5, "errors." The constant shifting could easily confuse the reader.

Organization: The writer can convey meaning by sequencing ideas within a sentence (syntax), within a paragraph, and among paragraphs. The reader needing information should receive that information with minimal processing. When the writer places roadblocks between the reader and the information (for example, using conditional statements such as "If . . . , then . . ."), the reader has to process and discard auxiliary information before finding the required information. Again, Figure 11 offers some interesting examples. Notice how various sentences (5 in number) open with a conditional (1.2-3; 3.1-2 and 3-4; 4.3; and 5.1). The reader must respond to a condition before performing an act. Two sentences (2.2 and 5.4) open with the command form of the verb. Taken as a whole, the reader faces a lot of information with relatively few action statements.

Meaning also comes through patterns of organization. Suppose you wanted to list ten keystrokes that a user needs in order to move text around in a document. You could prepare a numbered list with the lower numbered steps occurring prior to the higher numbered steps (Figure 6), or you could, instead, use letters such as A, B, C, and D. Another technique is to use bullets (Figure 6). But what if you used none of these? That you listed one step before another tells the reader that that step was chronologically prior to the next one. But what if, in putting the keystrokes together, they get somewhat jumbled? The reader is left to impose an order that may or may not be correct.

Selection

The third stage in information processing is pattern selection. When we read a computer manual, we read it to find specific information. In my example from a manual, I need to know how to move text around the document. Because moving text is not a frequent occurrence for me, I don't remember all the commands to use. So, I consult the manual. My success or failure depends on how easily I can access the specific information. The semantic meanings may be there, but other factors may prevent my getting the information (such as layout and design and readability).

With these points as background, let me now turn to my second major point, access of text.

ACCESS TO TEXT

Random access of information is made possible by such devices as titles,

headings, subheadings, running heads, and numbering systems. Of these, the two most important are headings and subheadings, and numbering systems.

Headings-Subheadings

Because readers of computer manuals do not read sequentially, retaining and sifting the information as needed, they must turn to the relevant sections. Indexes and tables of contents all help, but they only get the reader to the page, rarely indicating where on the page to find the information.

Once having located the page in the manual, the reader must find a specific section. If the writer uses headings, they should allow rapid access to the information, and when combined with typographical access structures (italic, bold, roman, size, caps, and lower case--see Appendix A) should prove distinctive enough for quick access. Positioning the heading is also important. Is it in the margin? Centered? Embedded? Finally, the heading's content should be helpful providing information rather than a generic label (see Figures 12, 13, and 14).

Insert Figures 12, 13 and 14 Here

Figure 12 shows a page with no headings to break-out the steps. Figures 13 and 14 show good uses of headings, but, in the case of Figure 14, a poor use of numbering (see below). Figure 14 also uses a generic heading ("Basic Concepts").

Numbering

The second access device is a numbering system that indicates the level of heading in combination with its position. Numbering systems can be helpful when they do not get in the way (what if you had to number headings/subheadings to five levels [I.A.1.a.(1) or 1.1.1.1.1 or 11111]). Such systems can be confusing, especially if you have cross references in the manual. A reader looking for heading numbered 1023 on page "10-17" might have problems. If page and section numbers were commonly written 12-34 as were the figure and table numbers, imagine the confusion in trying to cross-reference "See Figure 10-17 on page 10-12." Figure 15 shows decimal headings to 4 levels. (An anomaly with this manual is a cross-reference in Figure 14, manual page 6-13. 3.2. For a crossreference, the reader must scan all of chapter 9.)

Insert Figure 15 Here

Understanding

If the elements discussed above (layout and design, spacing, headings, numbers) contribute to helping readers get into the text, all is lost if they do not understand the material. Understanding is but one of three measures of a reader's access to the information. At one level, we have readability (applying various formulae to passages to determine grade level), then understanding, and finally comprehension (having the information become part of the

person's database and permitting that person to draw inferences). Mechanically, at least, readability can be influenced by the number of syllables and words per sentence because the formulas assume that long words and long sentences are hard to understand. On a different mechanical level, readability can be influenced by the number of clauses (groups of words that have subjects and verbs) because sentences with several clauses look and are hard to understand.

Understanding occurs when responses occur--the reader understands when he or she can perform a task, solve a problem, or agree with someone on a situation, among others (Lee). The material of the material to the final processing stage of processing in forms that allow easy access is crucial to that understanding. The various readability formulas are important because they give some insight into the form the signal takes when it enters final processing.

Now we can turn to the sample texts (listed in Appendix B) and evaluate their availability to the reader.

ANALYSIS OF TEN TEXTS

The various samples I have conform, to some degree, to the principles mentioned above. So, why all the fuss about unreadable manuals? Certainly, they need better layout and design to improve the random access of information; certainly the quality of the writing can be improved. But, do they actually help the reader understand what to do?

For simplicity, I analyzed the ten texts for readability and style. What follows are the methods and discussion of that analysis.

Methods

I selected one wordprocessing command to analyze: "Move a Block of Text." I typed it into two text analysis programs: (1) "Grammatik" and "Comment" on my 100+, and (2) "Readability Calculations" on an IBM PC. "Grammatik" and "Comment" analyze such factors as Content Index (a measure of the information in a document--see Appendix C), number of "to be" verbs, number of prepositions per sentence, percentage of transitions, use of "Th" openers for sentences, vagueness, number of short and long sentences, and number of problems identified based on a phrase dictionary (Barker).

"Readability Calculations" analyzes the text for nine readability formulas (see Appendix C): Dale Chall, Holmquist, ARI, Flesch, Kincaid, Powers, Fry, Coleman, and the Gunning Fog Index (Micro Power & Light Company).

Appendix D presents the results of analyzing nine computer manual instructions for moving a block of text (four examples from WordStar, four examples from other word processing programs, and one new version of a word processing program), plus a control text (not from a word processing program manual).

Copying or Moving Text

You use the copy or move procedure to save a specific block or column of text and insert it in a different location in the file. This is useful when you need to repeat the same information or to copy Format Lines. You can copy text to a temporary buffer or to a stored file.

Copying or Moving a Block of Text

To copy or move an entire block of text:

1. Position the cursor at the beginning of the text you want to copy.
2. Press the Do key.
3. Type the letter C (for copy) or M (for move).

When you move text, it is deleted from its original location. When you copy text, it is not deleted from the original location, and therefore exists twice in your file.

If you are storing the text in a separate file:

- Press the File key.
- If you do not want to use the default file name TEMP, type a file name.
- Press the Return key.

NOTE

If you do not specify a file to store the text in when you copy it, Samna stores it in a temporary buffer. This buffer can hold only about one full page of text. Therefore, if you have a large block of text to copy, you should store it in a separate file.

4. Shade the text you want to copy.
5. Press the Return key.
6. Move the cursor to where you want to insert the copy.

7. Hold down the Ctrl key while you press the Insert Here key. If you stored the text in a file:

- Press the File key.
- Type the name of the file, unless you used the default file TEMP.
- Press the Return key.

Samna asks:

```
Which format should be used?  
Type Y to use the current format. Type N to insert the  
stored format.  
Will the text be inserted into the current (displayed)  
format? Yes/No (N)  
Is the text you are inserting a column? Yes/No (N)
```

8. Respond to these questions and press the Return key.

NOTE

To move text with its original format, you must save the text in a separate file.

Copying a Format Line

To copy a format line:

1. Position the cursor immediately below the format line you want to copy.
2. Press the Do key.
3. Press the C key (for copy). If you are storing the line in a separate file:
 - Press the File key.
 - Type the name of the file if you do not want to use the default file name TEMP.
 - Press the Return key.

FIGURE 13. Text9

6.4.2 THE CUT COMMAND

This feature allows you to remove any amount of text from a document. Using the Paste command described later in this chapter, you may then move the cut text to another position within the same document or to another document within the same Document Directory. You may also elect to do nothing with the text you have cut, thus deleting it from your document. The Paste command in this case allows you to recover the last block of text you deleted in this manner.

6.4.2.1 Basic Concepts

Think of the Cut function as performing the same operation you would perform with a knife on a paper document. At the point where you would begin your cut in the paper document, you place the Select Marker in MASS-11. Moving the cursor in MASS-11 is similar to running your knife around the text you want to remove, working towards the end of the section. Finally, at the opposite end of the text from where you started, you executed the MASS-11 Cut function, which is similar to lifting the section of text from the document. Unlike the knife and paper operation, however, you are not left with a gaping hole in your document. MASS-11 automatically moves the text below the cut up to meet the text above the cut, so that there is never a hole left by the Cut operation.

The cut text is stored in a temporary holding area, or "paste buffer". The paste buffer contains the cut text until another piece of text selected with [SEL] is cut or copied, until you change Document Directories, or until you exit MASS-11. The amount of text that can be cut at one time is limited only by the disk quota allocated to your account by the System Manager.

6.4.2.2 Rulers in Cut Text

If the text you select has any ruler in the paste buffer with the text these rulers will be inserted into cut text with rulers from a document will be placed in the document format of the text which remains

6.4.2.3 To Cut Text Out of a Document

1. Position the cursor on the first character of the text to be cut.
2. Press the [SEL] key. Observe the diamond Select Marker.
3. Using any of the cursor movement commands, position the cursor one position past the last character to be cut.
4. Press [CUT] ([KP-]) to remove the text.

6.4.3 THE COPY COMMAND

This feature allows you to make a copy of any amount of text in a document. Using the Paste command described later in this chapter, you may then insert this text in another position within the same document or in another document within the same Document Directory. Use this function to avoid having to retype identical text in several places in a document, or in several different documents. Use it also when you want to be absolutely sure that text which appears in one place is identical in every respect to text which appears in another place.

6.4.3.1 Basic Concepts

The operation of the Copy function is fairly straightforward. An identical copy of the text you have selected is stored in the same paste buffer which is used for Cut operations. The paste buffer contains the copied text until another piece of text selected with [SEL] is cut or copied, until you change Document Directories, or until you exit MASS-11. The amount of text that can be copied at one time is limited only by the disk quota allocated to your account by the System Manager.

6.4.3.2 Rulers in Copied Text

If the text you select has any rulers embedded in it, these rulers will also be stored in the paste buffer with the text. If you paste the copied text into another location, these rulers will be inserted into the document along with the text.

FIGURES 14 AND 15.
Text8

Discussion

We can note a number of unusual findings when examining the data (Appendix D). First, few of the readability formulas can agree on the grade level of the text. The grade levels of Text9 range from 2-3 (Fry and ARI) to 9.5 (Dale Chall), suggesting that readability formulas may be statistically valid when looking at multiple samples using one formula, but that they range widely when looking at one text with several formulas. This view is supported by looking at the actual readability scores (before converting to grade level). While having a Fog Index of 7, Text9 has a Flesch Reading Ease score of 88 that is described as "easy" and for 5th grade readers of pulp magazines (see Appendix E).

The "Grammatik" and "Comment" analysis shows that the samples range widely in content (53-99 on a scale of 100; see Appendix C for the basis of the ratings). They also show that no text consistently falls in the acceptable range on the various elements for analysis (Table 2).

TABLE 2. Acceptable Levels "Comment"

ITEM	ACCEPTABLE LEVELS*	NUMBER SAMPLES ACCEPTABLE
"To Be"	Under 30%	1 (Text8)
Prepositions	2 per Sentence	4 (Text1,4, 5,8)
Transitions	Above 20%	5 (Text1,3, 5,7,8)
"Th" Openers	Under 9%	2 (Text6, 10)
Vagueness	Under 1%	6 (Text1,4, 6,7,8, 10)
Short Sentences	Under 30%	3 (Text4,5, 8)
Long Sentences	Under 22%	All but 1 (Text8)

* See Appendix C for discussion of Acceptance Levels

Table 3 shows a summary of an analysis of Text8 based on items discussed in the first part of this paper.

Table 3. Summary of Text8 Analysis Using Factors other than "Grammatik," "Comment," and "Readability Calculations."

FACTOR	COMMENT
Vertical Spacing	Text is justified top and bottom; extra space between headings/subheadings and text; breaks bottom of page in middle of instructions (sample p. 6-10 to 6-11).
Horizontal Spacing	Right-hand justification; breaks lines based on character count rather than syntax.
Sentence Structure	(Sections 6.4.2, 6.4.2.1, and 6.4.2.2 only). 11 simple; 4 complex; and 2 compound.
Headings/subheadings	Arabic numbers plus decimals; 4 levels.

The table shows that the writers followed some of the principles. Most notable are the headings/subheadings system (but not the numbering). They used 65% simple sentences; 23% complex; and 12% compound. The high percentage of simple sentences, however, is somewhat misleading. The writers embedded sentences within sentences, a practice that requires a more sophisticated level of processing. Embedded sentences are like subroutines in that they serve as information sources for the meaning of the sentence, yet do not stand alone to convey that information. Sentence 2 in paragraph 6.4.2 (1.2-4) opens with a sentence embedded ("... [which is] described later in this chapter . . .") followed by the main clause ("you may then move the cut text . . ."). The next sentence (1.4-5) contains the main clause ("You may also elect to do nothing . . .") plus an embedded clause ("you have cut") followed by a participial phrase ("deleting it from your document"). All in all, the reader must process a large amount of sophisticated text to learn about the "move" command. Of the 11 simple sentences, 9 are embedded.

Add structure and layout problems to the meaning problems (what does "working towards the end of the section" [2.5] mean?) and you have a complex, hard-to-read manual. The sophistication of the sentence structure in Text8 rivals that of a Time essay. But, we all can agree that the purposes of both the writers and readers are quite different.

CONCLUSION

The conclusion based on the data, it seems to me, is obvious. Text8 (Figure 14) had more categories acceptable (5) than any other sample (Table 2), yet it certainly is

far from easy for the reader to follow. It has the highest Fog Reading and Dale (Hall Grade Level (12) because of an average sentence length of 24 words (with the longest being 62 words), supporting the notion that it requires a higher level of formal education to comfortably read and understand. It may have a larger percentage of simple sentences, but a large percentage of those sentences are embedded with other sentences. The 3 analyses using the other factors also contribute to the ambiguity.

The ultimate test, however, is just how easily the reader can access the information. The analyses do not suggest an answer to that question. Rather, we are left with what actually happens when the user has a problem to solve: When all else fails, look in the manual! Writers and designers who combine layout and design, typography, and statistical analyses can present manuals for verification and validation that will place the reader back into them. Ultimately, however, all the measures available are useless if the writer fails to remember that there is a reader in the act of reading who needs to randomly access specific information to solve a problem.

REFERENCES CITED

- Aspen Software Company. "Grammatik" (Includes "Random House Proofreader," Version 1.15 [1982]), Version 1.84. Aspen, CO: Aspen Software Company, 1981.
- Barker, Thomas T. "Comment." Lubbock, TX: Texas Tech Microcomputer Laboratory, 1984.
- Carroll, Lewis (pseudo). "Jabberwocky." in Frank Kermode and John Hollander, general editors, The Oxford Anthology of English Literature, vol. 11. New York: Oxford University Press, 1973, p. 1488.
- Christensen, Francis. "Notes Toward a New Rhetoric," College English October, 1963, pp. 7-18.
- Cohen, Gerald. Readability Sample. Personal Correspondence, 1984.
- Felker, Daniel B., et al. Guidelines for Document Designers. Washington, DC: American Institute for Research, n.d.
- Hartley, James. "Current Research on Text Design," Scholarly Publication, 16, No. 4 (1985), 355-368.
- Lee, Irving J. "Why Discussions Go Astray," in S.I. Hayakawa, ed. The Use and Misuse of Language. Greenwich, Conn.: Fawcett, 1962, pp. 29-40.
- Micro Power & Light Company. "Readability Calculations: According to Nine Formulas." Dallas, TX: Micro Power & Light Company, 1984.

Morris, William, ed. The American Heritage Dictionary of the English Language. Boston: Houghton Mifflin Company, 1979.

Rogers, Everett and D. Lawrence Kincaid. Communication Networks: Toward a New Paradigm for Research. New York: The Free Press, a Division of Macmillan Publishing Co., Inc., 1981.

APPENDIX A DESIGN VARIABLES

- TYPOGRAPHY
- Typefaces
 - Typesizes
 - Emphasis (underlining, Marginal Notations)
 - Numerals
- PAPER
- Color
 - Weight and Reflectance
 - Surface
- COLOR (Contrast)
- SPATIAL ARRANGEMENT OF TEXT
- Page size
 - Number of columns
 - Separation of columns
 - Length of lines
 - Leading
 - Paragrapg Indentation
 - Vertical spacing
 - Margin size (Inside, outside, top, bottom)
 - Margin justification
 - Hyphenation
- SPECIAL FEATURES (Typography, spatial arrangement of text)
- Titles
 - Subheadings
 - Tables and Figures
 - Abstracts and summary
 - Table of contents
 - Bibliography and References Cited
 - Cover and spine titles
- AESTHETIC CONSIDERATIONS
- Typeface
 - Page size
 - Horizontal spacing
- PRACTICAL CONSIDERATIONS
- INTERACTION OF VARIABLES
- Type size, line length, leading
 - Margin justification, line length, hyphenation
 - Margin justification and leading
 - Typeface and type size
 - Line length and typeface

APPENDIX B:
MATERIALS ANALYZED

- TEXT1 pfs:Write. User's Manual. Software Publishing Company, 1983, pp. 7-5 to 7-7.
- TEXT2 Puotinen, C.J. The Last Word on WordStar. NY: Holt, Rinehart, and Winston, 1983, p. 156.
- TEXT3 WordStar manual supplied with Epson QX-10 (Release 3.3), p. 6-4.
- TEXT4 WordStar manual supplied with Kaypro (Release 3.3), 1983, pp. 4-8 to 4-9; and Rainbow 100, 1983, pp. 4-8 to 4-9.
- TEXT5 Curtin, Dennis P. The WordStar Handbook (release 3.3). Somerville, MA: Curtin and London, Inc., 1983, pp. 88-89.
- TEXT6 Samna Word II manual supplied with Rainbow 100, 1984, pp. 68-69.
- TEXT7 Select-86 manual supplied with Rainbow 100, 1983, pp. 56-57.
- TEXT8 MASS-11 Reference Manual, WS-200 Editor (Version 4-C), Hoffman Estates, IL: Microsystems Engineering Corporation, 1984, pp. 6-9 to 6-13.
- TEXT9 Samna Word III manual supplied with Rainbow 100, 1984, pp. 3-17 to 3-19.
- TEXT10 "Symposium Invoice Form Instructions," Fall DECUS U.S. Symposium, 1985, p. 10.

APPENDIX C
DETAILS OF
PROGRAMS

COMMENT

A Revision Aid Program for Writing Classes

DESCRIPTION

COMMENT is a computer-assisted revision aid designed to adapt Grammatik (tm Digital Marketing), a style and grammar analysis program, to writing classes. After students type in their papers on a word processor they correct the spelling and then run Grammatik. COMMENT then prompts students to enter data from Grammatik's statistical summary.

COMMENT automatically calculates percentages of surface-level elements, compares the percentages to standards pre-set by the instructor, and prints out advice in several modules (see below).

SET-UP, a companion program to COMMENT, allows the instructor to customize the standards against which COMMENT evaluates student writing.

CONTENT INDEX

This module calculates the number of verbs, nouns, adjectives, and pronouns in a paper or report. The percentage of these "content" words over articles, conjunctions, and prepositions, "structure," words, gives the writer a gauge of the paper's informative value. The CONTENT INDEX is also adjusted for the number of weak "to be" verbs and the count of possible vague terms like "great" and "many."

This module counts the number of weak "to be" verbs in a student's paper and compares the total to standards set by the instructor. The program uses GRAMMATIK's count of forms of "be" verbs and adjusts the figure down by 30 percentage points to account for repeated verbs in sentences and to achieve a more accurate reflection of the percent of "to be" verbs per total sentences.

PREPOSITIONS

The PREPOSITIONS module divides the number of sentences by the number of prepositions to calculate the number of prepositions per sentence. Acceptable numbers of prepositions per sentence are pre-set by the instructor. If students, say, use more than 2 prepositions per sentence, they are warned of possible dullness and wordiness and given examples as models for revision.

CONTINUITY

The CONTINUITY module uses the count of transitional phrases and the total sentence count to calculate the percent of transitional phrases per sentence. Optimum percents of transitional phrases are set by the instructor. The program tests the calculations for "too few" and "too many" transitional phrases and advises the writer either to add phrases or to be alert to possible wordiness. The CONTINUITY module is only useful to writers with an understanding of the limits and uses of mechanical transitions.

TH OPENERS

The TH OPENERS module counts the number of occurrences of ". Th... ." phrases and calculates the percent of these openers to the total sentence count. The acceptable percentage is pre-set by the instructor. The default value of 8% is based on a study of 25 first-year papers written by Texas Tech English students. In the case of descriptive reports (i. e. descriptions of mechanisms in technical writing courses) the allowable percentage of TH OPENERS may be as high as 50%.

VAGUENESS

The VAGUENESS module uses the count of possible vague terms in GRAMMATIK'S User Category 7 that is provided by the Microlab with the program. The program calculates the percent of possible vague terms per total words and advises the student to revise if the count is above the instructor's pre-set standard. Percents for possible vague terms often fall around 2-5%.

SENTENCE VARIETY

The SENTENCE VARIETY module uses the count of short sentences (<14 words) sentences and the count of long sentences (>30 words). It attempts to give the writer some idea of the balance of long and short sentences. It advises the writer either to try sentence combining or sentence splitting techniques in the event of a disproportion. The proportions of long and short sentences are pre-set by the instructor.

"Readability Calculations"

```
+++++
]
]
]          *** GENERAL NOTES ***
]
]      This program uses nine different formulas to compute read-
]      ability. All formulas are not appropriate to all grade levels,
]      although they may all return scores. For example, it is not
]      possible to compute grade levels less than FOURTH GRADE with
]      the Dale Chall formula. It is up to the user to determine
]      which formula and which scores are most appropriate for
]      the text being analyzed.
]
]      There are more than fifty readability formulas in existence.
]      A good overview is found in George Klare's article 'ASSESSING
]      READABILITY' which appeared in READING RESEARCH QUARTERLY,
]      Volume 10:1 (1974-1975), pp62-102.
]
]          PRESS [Q] to QUIT or [ENTER] to Continue
]
```



```

+-----+
]
]
]           FLESCH READING EASE
]
]   Rudolf Flesch published his first readability formula in 1943.
]   He was primarily interested in adult reading matter both in
]   terms of reading ease and human interest. He has a formula
]   for each. It is the 'Reading Ease' formula which is calcu-
]   lated here. The formula uses data from the Dale List of 3000
]   words. The score obtained is an index score which is then
]   translated to grade level by this program.
]
]   Source for this formula is: 'A New Readability Yardstick'
]   by Rudolf Flesch, 'JOURNAL OF APPLIED PSYCHOLOGY', No. 34
]   December, 1950, pp.384-390.
]
]           PRESS [Q] to QUIT or [ENTER] to Continue
]
+-----+

```

```

+-----+
]
]
]           *** FLESCH-KINCAID ***
]
]   Kincaid has modified the original Flesch formula for use
]   with Navy enlisted personnel undergoing technical training.
]   Unlike the original Flesch formula, the Flesch-Kincaid will
]   calculate grade levels less than fourth. The Flesch-Kincaid
]   Formula has also become a Military Standard, a dangerous
]   precedent because 'READABILITY' may not be so rigidly defined.
]
]   The primary source for the Flesch-Kincaid Formula is:
]   Kincaid, Peter, et al. 'Development and Test of a Computer
]   Readability Editing System (CRES). Final Report,' Naval
]   Training and Evaluation Group, TAEG-R-83, March, 1980.
]   ED 190-064 (ERIC Document)
]
]           PRESS [Q] to QUIT or [ENTER] to Continue
]
+-----+

```

```

+-----+
]
]
]           *** The FOG Index ***
]
]   The FOG index is a very popular readability formula largely
]   because of its ease of manual application. It does tend to
]   give scores which are higher than scores given by other
]   formulas. One explanation offered for this is that the FOG
]   formula is designed to measure the level of comprehension
]   as opposed to the level of speaking. Whether or not this is
]   considered a valid explanation is up to the reader.
]
]   The FOG Index was developed by Robert Gunning in 1952. It
]   was published originally in 'The Technique of Clear Writing'
]   McGraw Hill, c1952, Revised edition, 1968.
]
]           PRESS [Q] to QUIT or [ENTER] to Continue
]
+-----+

```


APPENDIX D
TABLE OF DATA

PROGRAM	SECTION	TEXT SAMPLE NUMBER									
		1	2	3	4	5	6	7	8	9	10
READABILITY	#WORDS	280	148	288	234	492	300	193	1193	689	363
	3-SYLLABLE	24	20	29	19	33	13	9	90	33	48
	SENTENCES	16	10	16	13	27	26	15	55	59	30
	SYLLABLES	885	217	413	341	671	385	258	1625	872	569
	SYLL./100 WDS	137	147	143	146	136	128	134	136	127	157
	SENT./100 WDS	5.7	6.8	5.6	5.6	5.5	8.7	7.8	4.6	8.6	8.3
	FOG READING	10	11	11	10	10	6	7	12	7	10
	FLESCH EASE	73	68	67	65	73	86	81	70	88	61
	FLESCH GRADE	6	7-8	7-8	7-8	6	5	5	7.8	5	7.8
	POWERS EASE	5.4	5.6	5.7	5.8	5.4	4.5	4.9	5.7	4.5	5.8
	HOLMQUIST	6.8	6.6	6.7	7	6	6.6	6	7.1	6.3	6.9
	ARI	7.5	7.5	8.6	9	7.7	3.2	4.6	9.4	3	7.7
	FLESCH/KINCAID	7.4	7.5	8.3	8.6	7.6	4.1	5.2	9	4	7.6
	COLEMAN	7.7	9.1	8.8	9.2	7.5	5.1	6.4	7.8	4.8	10
DALE CHALL	9.5	9.5	9.5	12	7.5	9.5	7.5	12	9.5	12	
GRADE LEVEL	DALE CHALL	9.5	9.5	9.5	12	7.5	9.5	7.5	12	9.5	12
	HOLMQUIST	6	6	6	6-7	5-6	6	5	6-7	5-6	6
	ARI	7	7	8	8-9	7	2-3	4	9	2-3	7
	FLESCH	6	7-8	7-8	7-8	6	5	5	7-8	5	7-8
	KINCAID	7	7	7-8	8	7	3-4	4-5	8-9	3-4	7
	POWERS	5	5	5	5	5	4	4-5	5	4	5
	FRY	6-7	6-7	7-8	7-8	6-7	3-4	4-5	7-8	2-3	8-9
	COLEMAN	7	8-9	8	8-9	7	4-5	5-6	7	4	10
	GUNNING FOG	10	12	12	10	9.5	5-6	6-7	11	6	9.5
	GRAMMATIC	AV. SENT. LENGTH	16	13	16	17	19	11	13	24	12
AV. WORD LENGTH		4.3	4.5	4.5	4.4	4.1	4.0	4.1	4.2	3.9	4.8
LONGEST SENT.		26	23	27	25	42	23	25	62	41	25
SHORTEST SENT.		5	6	9	11	9	4	2	3	2	2
"TO BE" VERBS		1	2	7	5	12	5	2	40	16	10
PREPOSITIONS		53	21	37	37	76	38	33	194	79	39
COMMENT*	CONTENT	53	68	58	72	52	99	57	56	97	85
	"TO BE" (30%)	24	12	11	6	16	11	16	44	1	4
	PREPOSITIONS (2)	3	2	2	3	3	1	2	4	1	1
	TRANSITIONS (20%)	72	9	29	7	27	12	29	37	7	7
	"TH" OPENERS (9%)	11	9	24	21	15	4	21	16	7	0
	VAGUENESS (1%)	0.4	1.3	1.5	0	2.2	0.3	0.5	0.6	1.3	0.3
	SHORT SENT. (30%)	44	45	35	21	27	58	43	22	69	66
	LONG SENT. (15%)	11	0	0	0	8	0	0	22	2	0
	PROBLEMS NOTED	3	4	7	1	15	1	3	22	13	1

*For explanation of values, see Appendix C.

APPENDIX E

Interpretation of Reading Ease Scores

(Taken from Rudolf Flesch, The Art of Plain Talk (NY: Harper & Brothers, 1946), p. 205. Note the date of publication. In a 1979 book [How to Write Plain English] Flesch raises the "School Grade Completed" one year so that 90-100 now is 5th grade.)

Score	Description of style	Average sentence length in words	Syllables per 100 words	Typical magazine	Potential audience (typical audience one step above)	
					School grade completed	Per cent of United States adults
0 to 30	Very difficult	29 or more	192 or more	Scientific	College	4.5
30 to 50	Difficult	25	167	Academic	High school or some college	24
50 to 60	Fairly difficult	21	155	Quality	Some high school	40
60 to 70	Standard	17	147	Digests	7th or 8th grade	75
70 to 80	Fairly easy	14	139	Slick-fiction	6th grade	80
80 to 90	Easy	11	131	Pulp-fiction	5th grade	86
90 to 100	Very easy	8 or less	123 or less	Comics	4th grade	90

SYMPOSIUM INVOICE FORM INSTRUCTIONS

Do not use the Symposium Invoice Form if you do not plan on attending the Symposium or Pre-symposium Seminar.

GENERAL INFORMATION:

- Digital employees note: no cross charges will be performed for products offered on Symposium Invoice Form.
- Cancellations: apply only to symposium, session notes, and pre-symposium seminar sections of the invoice form.
- Transfers:
 - **Will only be accepted when no changes are made to original attendee's record.**
 - If any changes to original attendee's record are made, a cancellation will take place and a new registration form and payment must be submitted.

INVOICE FORM HEADER INFORMATION:

- Make sure DECUS number, if known, is provided.
- If not a DECUS member you are required to fill out a membership form (found on page 89) and submit with the Symposium Invoice Form.

SYMPOSIUM SECTION:

- Indicate number of days you plan to attend.
- Check which days you are attending if less than five.
- Enter corresponding dollar amount on appropriate line.
- Enter symposium amount due on line (A).
- If you are not ordering additional products carry subtotal (A) to line (F) at bottom of form.

SESSION NOTES:

- Enter quantity.
- Enter corresponding dollar amount on appropriate line.
- Enter session note amount due and place on line (B).

SUBSCRIPTION SERVICE:

- For U.S. Chapter members **only**.
- No cancellations for subscriptions will be accepted.
- Enter quantity.
- Enter corresponding dollar amount on appropriate line.
- Enter subscription service amount due and place on line (C).

LIBRARY:

- For U.S. Chapter members **only**.
- No purchase orders accepted for library programs offered on this form.
- No cancellations of library orders will be accepted.
- If registration is cancelled you will receive library programs by mail.
- Fill out "Ship To" address on back of invoice form.
- **LIB1 = 11-SP-18**
Language System for RSTS/E V7.2,-8, RSX-11 M V4.0, RSX-11 M-PLUS; RT-11 V4.0, VMS V3.2 in Compatibility Mode, TSX-PLUS V2.2/3.0 on 9 Track Magtape, 800 BPI, DOS-11 Format.
- **LIB2 = 11-SP-47**
PORTACALC: 3D Spreadsheet, for IAS, RSX-11 D, RSX-11 M, RSX-11 M-PLUS, VAX/VMS on 9 Track Magtape, 1600 BPI, RMSBCK Format.
- **LIB3 = VAX-LIB-3**
1985/1986 DECUS VAX/VMS Library Tape #4 on 9 track Magtape 1600 BPI, VMS/BACKUP Format.
- **LIB4 = PRO-123**
PRO Package of BASIC, PASCAL, PORTACALC, KERMIT, and a Desk Top Calendar for PO/S on 5¼" Floppy Disks, FILES-11 Format.
- Enter quantity (no more than 9).
- Enter corresponding dollar amount on appropriate line.
- Enter library program amount due and place on line (D).

PRE-SYMPOSIUM SEMINAR:

- Enter code number for first, second, and third choices (see pages 15—52 for code description).
- Enter pre-symposium seminar amount due (\$195) and place on line (E).

INVOICE FORM TOTAL:

- Add lines A and other product lines (B,C,D, and E) and place total amount due on line (F).
- Signature: By signing this form you agree to abide by the Canons of Conduct listed on the reverse side of the invoice form.

Credit Card Customers: ● MC = Mastercard ● V = Visa ● D = Diners Club/Carte Blanche

- Check appropriate credit card box and enter credit card number and expiration date.

- Mail To: DECUS Symposium Administration, 219 Boston Post Road, (BP02), Marlboro, MA 01752

TEXT10. Control Text

RT-11 SIG

Ned W. Rhodes
Software Systems Group
1684 East Gude Drive
Rockville, MD 20850

ABSTRACT

RT-11 Extended Memory (XM) monitor. It consists of a foreground data acquisition routine that acquires data from a DRV-11W interface unit via a device driver and stores data in a 16KW histogram array. A background routine controls the entire data acquisition process and communicates to the foreground job via MQ, the communications handler. In order to get the best throughput, a technique known as "jam queueing" is used to ensure that the handler always has a queue element to process. The XM monitor provides an excellent environment for the development and execution of large, virtually overlaid data acquisition programs through its management and support of extended memory.

1 INTRODUCTION

This paper will describe a real-time data acquisition system that operates under the RT-11 operating system, Version 5.1. The data will be acquired from a two dimensional, position sensitive, proportional counter that is used to examine the structure of polymer blends. Polymer blends are combinations of materials that have differing characteristics. When these materials are bonded together, the new structure can have yet another characteristic. For example, ABS is a common polymer blend that is used in most computer terminal enclosures. ABS is a combination of two different materials -- one is very rigid to provide structural strength, while the other is flexible. The combination results in a material that is both rigid and that can absorb minor impacts without breaking.

The paper will be divided into five major sections. In the first section, the hardware used in the data acquisition system will be described. Section 2 will discuss the system design considerations, while section 3 will consist of a design walkthrough of the chosen design. Section 4 will contain listings of the programs that make up the system. The final section will summarize the findings of the paper.

1.1 Purpose

The purpose of the paper is to take a quick look at the hardware involved in the system and to then take a more detailed look at the system design and the system software. This will be a software oriented paper rather than a paper that is only hardware oriented.

1.2 Hardware Description

This data acquisition system is in use at the National Bureau of Standards in Washington DC. An LSI-11 is connected to the two dimensional counter via a DEC DRV-11W. The device is primarily an XRAY detection device.

An XRAY source bombards a polymer sample. The sample gives off electrons that hit an X-Y grid. When the electrons hit the grid, a current is detected at both ends of the X and Y grids. Detection circuitry measures the relative rise time of the pulse at each end of the grids, which delivers two seven bit numbers to the computer. These seven bit numbers correspond to the X and the Y coordinates where the electron hit the grid.

A histogram is a convenient data structure to store this particular type of position information. If the X and the Y addresses are combined to form a 14-bit

address, then a program could increment a counter at that address whenever an electron was detected at that position. The histogram bin counter would then give an reading as to the number of particles that were detected at each X and Y positions.

A DRV-11W interface card is used to transmit data from the counter to the computer. Once the word count register is loaded on the DRV and the card is enabled, the counter will transmit the requested number of samples at its own rate using direct memory access (DMA). In other words, the counter will transmit data asynchronously, depending upon how many electrons strike the grids in a given unit of time and without CPU intervention.

2 DESIGN CONSIDERATIONS

Now that the hardware has been described, the design of the data acquisition software can now be discussed. The data acquisition system had to be capable of :

1. Acquiring data via DMA using the DRV-11W.
2. Reaching a 10,000 sample per second data acquisition rate.
3. Storing the data in a 16 KW histogram array.
4. Taking periodic snapshots of the histogram array under program control.
5. Supporting a background controlling routine that would do other tasks while data was being acquired.

2.1 Possible Solutions

Given the above specifications, it was important to choose the proper RT-11 monitor for the data acquisition system. The Single Job (SJ) monitor could not meet all the requirements due to the fact that it does not support multiple jobs -- it would not be possible to collect and analyze data at the same time. The Foreground/Background (FB) monitor was considered, but it had problems supporting a foreground job that used a 16 KW histogram and a background job of any significant size. It seemed that the Extended Memory (XM) monitor was the logical choice because of its support of extended memory.

Once the proper monitor had been chosen, then it was time to divide the tasks between jobs. It seemed reasonable to acquire the data in a foreground job, and to reduce and analyze the data in a background job. It now had to be decided on which type of mapping to use for the various jobs in the system under the XM monitor.

2.2 FORTRAN Virtual Arrays

Virtual arrays were considered and immediately dropped due to their slow implementation. Under the XM monitor, APR 7 is used as a "sliding window" map into the virtual array. This means that only 4 KW virtual array locations can be mapped at one time, and that whenever the system needs to map to an element outside of the currently mapped window, it must first slide the window. Due to the random nature of the experiment, it was thought that this window would have to be moved for each element that was updated and the code to move this window is very costly in time. Note that in order to map the entire histogram array, 4 complete APRs would be required. And, the other problem with FORTRAN virtual arrays, is that they use APR 7 to map to the virtual arrays. This precluded the program from talking directly to the I/O page and controlling an external device.

2.3 Privileged Foreground Job

A privileged foreground job initially maps the low 32 KW of memory under RT-11. When the program is run, it normally is loaded so that it uses APR 5. If the program is large enough or there are a number of handlers loaded in the system, the program could load so as to use both APRs 4 and 5. If the histogram requires 4 APR registers, only APRs 0-3 would be available, which in turn means that the program could not connect directly to the DRV-11W interrupt. The net result is that this type of mapping is very wasteful of the low memory resource (1-2 APRs) and is very dependent upon a system's configuration -- the program would have to load in nearly the same place each time so that the correct APR registers would be available. This did not seem to be an optimum solution.

2.4 Virtually Overlaid Foreground Job

In order to better utilize the memory attached to the processor, a virtually overlaid foreground job was considered. Because of the mapping, it is not possible to address either the I/O page or the interrupt vectors with a virtually overlaid foreground job. That would mean that a handler would have to be used to acquire the data. But, by virtually overlaying the data acquisition job, very little low memory would be consumed so that a fairly large background job could be supported.

The only other problem with a single foreground job is the fact that the control of a foreground job is hard to automate due to the fact that the operator has to type a control-F to address the foreground job and then a control-B to address the background job. Because of this, it is hard to automate the control of a foreground job through the use of a command file or IND.

The conclusion here is that a virtually overlaid job will solve the low memory problems, but that a handler would be required to talk to the device and handle the interrupts. It seems that a background controlling job would also be required to command the foreground data acquisition job due to the fact that a command file cannot interact directly with a foreground job.

2.5 Handler

The use of a device handler allows for the most general solution to the data acquisition problem. A handler knows how to map user buffers in extended memory and it allows the system to hide the specifics of the interrupt service routine from the user. And, a handler will work very well with a virtually overlaid program. The only potential problem with a handler is the fact that a handler is queue driven. This means that a handler processes requests from its queue in a serial fashion. That means that after it processes one queue element, there is a finite period of time where it is shuffling queue elements and cannot be acquiring data. In order to meet the sampling rate needs of this system, the handler needs to spend a minimum amount of time shuffling these queue elements so that it can acquire data. If the program were to directly connect to the interrupt vector, it would receive the end-of-block interrupt, start the next scan of data and then process the acquired data. With a handler, it receives an end-of-block interrupt, returns the completed queue element and then initiates the next data request. Note that there is a longer period of time, in the handler, where data is not being acquired, than in the case of the direct connection to the interrupt vector.

3 SELECTED DESIGN

Overall, the data acquisition system would consist of :

1. A background controlling routine.
2. A foreground data collection routine.
3. A handler to acquire the data.
4. Virtually overlaying both jobs to conserve the use of low memory.

3.1 Background Controlling Routine

This routine is needed to control the acquisition of data and to interface with the operator. Because it is a background routine, it may be automated by the use of command files or IND control files. In order to control the foreground routine, some synchronization commands had to be defined. They were :

1. B -- Begin data acquisition
2. Z -- Zero the histogram array
3. S -- Stop data acquisition and write the histogram to disk
4. I -- Initialize for data collection
5. X -- Exit

In all cases, the commands are echoed back to the background routine. Through the use of these simple commands, a background routine can effectively control the acquisition of data.

3.2 MQ Communications

The MQ handler can be used to send messages between any two jobs in an RT-11 system. The way it is used in this case is to have the foreground data acquisition routine "hang" a read from any job to MQ and then to wait for a message. Now, when the background controlling routine is activated, it will send an initialize (I) command to the foreground job. The foreground job will receive the initialize message and then will open a dedicated communications path to the background controlling job. Now, the controlling job and the data collection job can communicate over MQ.

Note that all reads of messages from MQ use completion routines so that no CPU time is required to poll MQ to see if there are messages. A library of subroutines were written for the background controlling routine that "hides" the actual implementation from the user; his interface is only a set of subroutine calls. And, because of the use of MQ, the controlling job can be another system job -- it does not have to be a background job. This can provide extra flexibility when needed for special data acquisition tasks.

3.3 Foreground Data Collection Routine

The design of the data collection routine is quite simple as it is a slave process to the controlling routine. The routine is all event driven, either off of messages from MQ or as result of data collection interrupts. As a result of being event driven, the foreground routine consumes very little CPU resources (except when it is processing an event) so that there is plenty of CPU time available for the background routine.

Once the foreground routine receives the command to collect data, it immediately queues two data request to the handler, requesting completion routines once the data has been acquired. In the completion routine the data will be added to the histogram array and then another data acquisition request will be made to the handler unless a stop request has been received. This cycle of collection and addition will continue until a stop request is received.

Once a stop request is received, the foreground job monitors a counter that contains the number of outstanding I/O requests that are queued to the handler. Once that count has gone to zero, the foreground routine can write the data out to an intermediate disk file and then wait for the next command from the controlling background routine.

3.4 Handler Routine

A device handler to acquire data from a DRV-11W is a relatively simple program. All the handler has to do is: 1) validate the request; 2) convert the user's 16-bit virtual address into the proper 18-bit physical address; 3) load the device registers; and 4) start the conversion. Once the interface card interrupts after the data is acquired, the handler will be entered again to field the interrupt. If there are no errors, then the handler simply returns. If there are errors, the handler will report them back to the user.

3.5 Jam Queueing

In order to use this technique, the data collection routine "jams" two or more data requests to the handler. This has the affect of queueing two or more queue elements to the handler. Then, when the handler completes the current request and returns the queue element to the monitor, the monitor will check to see if they are any pending queue elements waiting on the queue for that handler. If there are queue elements waiting, the monitor will initiate the first waiting request BEFORE returning the completed queue element to the calling program. This has the effect of allowing for faster system throughput because waiting requests will be scheduled as fast as possible by the monitor.

Normally, a program will queue one request to the hander, wait for it to complete and then queue the next request. Using the technique of "jam queueing", two requests are queued initially so that the handler always has something to do. A program can save the time required to context switch and schedule the data collection routine, and allow for higher system throughput.

3.6 Virtual Overlaying

In order to minimize the amount of low memory that is consumed, both the controlling routine and the data collection routine should be virtually overlaid. The easiest way to accomplish this is to create a small main routine that calls a subroutine that performs the real work. For example, the main routine for the foreground data collection routine is :

```

.mcall      .qset
.asect
.=44
.word      2000

.psect     code
.globl     acquir
AROOT:
.qset      #queue,#10.
jmp        acquir

.psect     data
queue:.blkw 100.
.end       AROOT

```

Note that the root only consists of some extra queue elements and a jump instruction to the main code that is in a virtual overlay region. The following instructions link the data collection routine as a virtually overlaid program.

```
LINK AROOT/XM/PROMPT/MAP/EXE:ACQUIRE
ACQUIR,SAVRES/V:1
//
```

4.2 Background Controlling Routine

This routine is the background controlling routine. Note the use of a series of library subroutines to talk to the foreground job.

```

subroutine data
c
c
c   Control the collection of data
c
c
call setup           ! init
call readit         ! read the return char
call mqwrit('Z')   ! zero the array
call readit         ! read the return
Y=SECNDS(0.)
call mqwrit('B')   ! begin data collection
call readit         ! read the return
10  Y2=SECNDS(Y)
    IY2=Y2
    IF(MOD(IY2,10).EQ.0) TYPE *,IY2
    IF(Y2.LT.300.) GO TO 10
C   write(7,9000)
C9000 format(' Hit return to stop data collection ', $)
C   read(5,9001) i
C9001 format(i1)
call mqwrit('S')   ! stop the data collection
call readit         ! read the S
call readit         ! read the W
C   call mqwrit('X') ! stop the job
end
subroutine readit
integer getmq
100  ichar = getmq() ! see if a char is avail
    if (ichar .eq. 0) goto 100 ! if not loop
    write(7,9000) ichar
9000 format(lx,a1)
return
end

```

4 CODE EXAMPLES

This section contains listings of the code required to acquire the data and to control it.

4.1 DROOT

This is the main routine for the background controlling routine

```

program droot
call data
end

```


4.3 Communications Subroutines

These subroutines communicate with the foreground collection routine.

```

        .title   Foreground communications subroutines
;
;
;   This is a collection of subroutines to support data acquisition
;   that is taking place in the foreground.  The routines are:
;
;   SETUP -- Called to setup up things for data acquisition.  It
;             allocates more queue elements, opens channels to the
;             MQ handler, clears out the command ring buffer and hangs
;             a read to the MQ handler.
;
;   MQWRIT ( char) -- Sends this character command to the foreground
;                       job
;
;   GETMQ -- This is a character function that returns either a 0 if
;             no character is available, or the character that the
;             foreground jobs sends back.
;
;   Ned W. Rhodes
;   Software Systems Group
;   1684 East Gude Drive
;   Rockville, MD 20850
;
;
;   .mcall  .lookup,.close,.readw,.writw
;   .mcall  .dstatus,.exit,.print,.twait
;   .mcall  .readc,.gtjb,.ttyout
;   .mcall  .read,.wait,.qset
;   .globl  igetc
;   .ident  /V1.0/
;   .enabl  lc
;   .nlist  bex
;   .sbttl  SETUP
;   .page
;   .psect  code
;
;
;   setup::
;   .qset   queue, 10.           ; let's have more queue ele.
;   mov     parm,r5             ; get parameter block
;   call    igetc               ; get a free channel
;   mov     r0,mqread           ; and save it
;   mov     parm,r5             ; get parameter block
;   call    igetc               ; get a free channel
;   mov     r0,mqwrt            ; and save it
;   .lookup area,mqread, jobdes ; open MQ for reading
;   bcc     1$                  ; no error
;   mov     mqopen,r0           ; MQ open error
;   jmp     error                ; and exit
;
;   1$:    .lookup area,mqwrt, jobdes ; open MQ for write
;   bcc     2$                  ; no error
;   mov     mqopen,r0           ; MQ open error
;   jmp     error                ; and exit
;
;
;   zero out the ring buffer
;
;
;   2$:    mov     5.,r0          ; 5 words
;   mov     ring,r1             ; the address of the buffer
;   3$:    clr     (r1)+          ; clear a word
;   sob     r0,3$               ; and loop
;   mov     ring,cur            ; initialize pointer
;   mov     ring,next          ; initialize pointer

```

```

;
;
;   Send an init to the foreground job
;
;
;   .gtjb   area, job, -1           ; get my job name
mov     6.,r0                       ; 6 characters
mov     job,r1                      ; address of one buffer
add     22,r1                       ; advance to name
mov     i,r2                        ; where it goes
inc     r2                          ; really here
44$:   movb  (r1)+,(r2)+           ; move it
sob     r0,44$                     ; and loop
;
;
;   Hang a read to MQ
;
;
;   .readc  area,mqread, commnd, 5., mqcomp, 0
bcc     4$                          ; hang a read
; no error
mov     reader,r0                   ; get the message
jmp     error                       ; and say the error
4$:   .writw area,mqwrt, i, 5, 0    ; init it
return ; and return
;
;   .sbttl  MQ read completion routine
;   .page
mqcomp: movb  comtxt,@next           ; save in ring buffer
inc     next                       ; bump pointer
;   .print  hit
cmp     rend,next                   ; at the end?
bne     5$                          ; nope
mov     ring,next                   ; reset address
5$:   .readc  area,mqread, commnd, 5., mqcomp, 0
; hang a read
return ; and return
;   .sbttl  GETMQ
;   .page
getmq:: clr  r0                     ; start with a zero
cmp     cur,next                   ; are pointers the same?
beq     6$                          ; yes, exit
movb    @cur,r0                    ; get the character
inc     cur                        ; bump pointer
cmp     rend,cur                   ; at the end
bne     6$                          ; nope
mov     ring,cur                   ; reset address
6$:   return                        ; and return
;   .sbttl  MQWRIT
;   .page
mqwrit:: movb @2(r5),cbuf           ; move over character
;   .writw  area,mqwrt, cbuf, 1, 0  ; write it up
return ; and return
;   .sbttl  Error exit
;   .page
error:  .print                      ; say the error
;   .exit
;   .sbttl  Data and Storage
;   .page
;   .psect  data
jobdes: .rad50 /MQ/

```

```

        .ascii  /ACQUIR/
queue:  .blkw  100.          ; additional queue elements
sblk:   .blkw   4           ; status return area
area:   .blkw  10.          ; emt area
commnd: .word   0           ; read buffer
comtxt: .blkw  10.          ; command coming up
mqread: .word   0           ; mq read channel
mqwrt:  .word   0           ; mq write channel
job:    .blkw  12.          ; job info
ring:   .blkb  10.          ; 10 commands
rend    =   .              ; end of buffer address
cur:    .word   ring        ; current pointer
next:   .word   ring        ; next empty position
parm:   .word   0           ; zero parameters
I:      .ascii  /ITEST  /   ; initialize buffer
cbuf:   .ascii  / /        ; command buffer
        .psect  msgs
mqopen: .asciz  /?SETUP-F-Error opening MQ/
reader: .asciz  /?SETUP-F-Error posting a read to MQ/
hit:    .asciz  /?MQCOMP-I-Got a read from MQ/
        .end

```

4.4 Foreground Data Collection Routine

This routine is the foreground data collection routine. It communicates with the background routine via MQ and it acquires data using a handler AB.

```

        .title  Foreground Data Acquisition Routine
        .sbttl  Program Description
        .mcall  .lookup,.close,.readw,.wrtw
        .mcall  .dstatus,.exit,.print,.ttyout
        .mcall  .wait,.write,.readc,.enter,.twait
        .mcall  .mrkt
        .ident  /V1.0/
        .enabl  lc
        .nlist  bex
;
;
;   Foreground routine that acquires data from a DR11-W under the
;   control of the background routine. Program should be linked
;   virtually due to the fact that it has a large buffer.
;
;   It works like this:
;
;   1. This routine is started and it zeros the histogram buffer
;   2. Next it waits for a command from the background job via
;      the MQ handler.
;   3. This job executes the command and echos it back to the background
;      once it completes the command.
;
;   The commands are :
;
;   B : Begin data acquisition
;
;   Z : Zero the histogram array
;
;   S : Stop data acquisition and write histogram to disk.
;      This routine will stop data acquisition only when there are
;      no outstanding requests to the handler and then echo back to
;      the background program. After it writes the data to disk, it will
;      echo back that fact to the background job.
;
;

```

```

;      W : This is returned to the background once the data is written to
;      disk.
;
;      I : Initialize for data collection and open an MQ communications
;      channel to the background.
;
;      E : This is not really a command, but an error condition that signals
;      the fact that the handler detected an error.
;
;      X : Exit.
;
;
;      In the completion routine, a co-routine is used to save the
;      registers.
;
;      Ned W. Rhodes
;      Software Systems Group
;      1684 East Gude Drive
;      Rockville, MD 20850
;
      .sbttl  Program constants
      .page
abchan = 1          ; AB handler channel
mqread = 2         ; MQ read channel
mqwrit = 3         ; MQ write channel
dchan = 4         ; the data channel
herr = 1          ; hard error bit
begin = 'B        ; begin command
zerob = 'Z        ; zero command
init = 'I         ; init command
stop = 'S         ; stop command
hderr = 'E        ; hard error from handler
exit = 'X         ; exit command
      .sbttl  Program initiation
      .page

      .psect  code
acquir::
      .print  start          ; print start message
      call   zero           ; zero the histogram buffer
      .dstat  sbk, AB       ; get the handler status
      bcc    1$             ; no error
      mov    sterr,r0       ; get the message
      jmp    error         ; and exit
1$:      .print  tst          ; print handler status
      tst    sbk+4          ; check the status
      bne    2$            ; handler is there
      mov    noload,r0     ; get the message
      jmp    error         ; and exit
2$:      .lookup area, abchan, ab ; open the handler
      bcc    3$            ; branch if no error
      mov    look,r0       ; get the message
      jmp    error         ; and exit
3$:      .lookup area, mqread, jobdes ; open MQ for any job
      bcc    cmdlop        ; no error
      mov    mqopen,r0     ; MQ open error
      jmp    error         ; and exit
      .sbttl  Main command loop
      .page
cmdlop:
      .print  wait          ; waiting for data
      call   ztext         ; zero the text buffer
      .readw  area, mqread, commnd, 10. ; hang a read to MQ
      mov    commnd,r2     ; number of words transferred
      mov    comtxt,r1     ; address of buffer
1$:      .ttyout (r1)+      ; echo command
      .ttyout (r1)+      ; echo command
      sob   r2,1$         ; for number of words

```

```

;
;
;   Check for command
;
;   cmpb    begin,comtxt           ; begin command?
;   bne     2$                    ; nope
;
;   Begin command
;
;   21$:   tst     readc             ; anything going on?
;         bne     21$              ; yes, wait til done
;         clr     stopf            ; clear the stop flag
;         inc     readc            ; increment outstanding count
;         .readc  area, abchan, buffa, 128., adone, 0
;         ; Start buffer a reading
;         bcc     22$              ; no error
;   23$:   mov     reade,r0         ; Get the message
;         jmp     error            ; And exit
;
;   22$:   inc     readc            ; increment outstanding count
;         .readc  area, abchan, buffb, 128., bdone, 0
;         ; Start buffer b reading
;         bcs     23$              ; Error from readc
;         .wait   mqwrit           ; wait for previous i/o on MQ
;         .write  area, mqwrit, comtxt, 1, 0
;         ; echo command back
;         jmp     cmdlop           ; and repeat
;   2$:   cmpb    zerob,comtxt     ; zero command
;         bne     3$              ; nope
;
;   Zero command
;
;   call    zero                  ; zero the buffer
;         .wait   mqwrit           ; wait for previous i/o on MQ
;         .write  area, mqwrit, comtxt, 1, 0
;         ; echo command back
;         jmp     cmdlop           ; and repeat
;   3$:   cmpb    init,comtxt     ; init command?
;         bne     4$              ; nope
;
;   Init command
;
;   mov     6,r0                  ; number of characters
;   mov     comtxt,r1             ; get buffer address
;   add     1,r1                  ; past command
;   mov     userj,r2              ; address of destination
;   7$:   clrb    (r2)             ; insure a zero
;         cmpb    40,(r1)         ; is it a space
;         beq     8$              ; yes
;         movb    (r1),(r2)       ; move it
;   8$:   inc     r1               ; bump the address
;         inc     r2               ; bump the address
;         sob     r0,7$           ; and loop
;         .close  mqwrit
;         .lookup area, mqwrit, bg ; open MQ to user's job
;         bcc     9$              ; no error
;         mov     ulook,r0        ; get the message
;         jmp     error            ; and exit
;   9$:   .wait   mqwrit           ; wait for previous i/o on MQ
;         .write  area, mqwrit, comtxt, 1, 0
;         ; echo command back
;         jmp     cmdlop           ; and repeat
;   4$:   cmpb    stop,comtxt     ; stop command?
;         bne     5$              ; nope

```

```

;
;
;   Stop command
;
;
55$: inc      stopf                ; set the stop flag
     tst      readc                ; see how many reads hung
     bne      55$                  ; still some out
     .wait    mqwrit                ; wait for previous i/o on MQ
     .write   area, mqwrit, comtxt, 1, 0 ; echo command back
;
;
;   Now write the data to disk
;
;   Watch the timing here, as this job will continue to run
;   even though the background now has the signal that collection
;   has stopped.
;
;
     .enter   area, dchan, dfile, -1 ; Open the file
     bcc      66$                  ; no error
     mov      enterr,r0            ; get the error message
     jmp      error                ; and error out
66$: .writw   area, dchan, hist, 16384., 0 ; write the data
     .close   dchan                ; close the file
     .wait    mqwrit                ; wait for previous i/o on MQ
     .write   area, mqwrit, writen, 1, 0 ; let bg know we wrote it
     jmp      cmdlop                ; and repeat
55$: cmpb    exit,comtxt           ; exit command?
     bne      10$                  ; nope
     .exit                                ; exit
;
;
;   No command match, just ignore
;
;
10$: .print   nocom                ; message
     .wait    mqwrit                ; wait for previous i/o on MQ
     .write   area, mqwrit, comtxt, 1, 0 ; echo command back
     jmp      cmdlop                ; and repeat
     .sbttl   Buffer a completion routine
     .page
adone: call    savres                ; save all registers
     bit      herr,r0              ; check for error
     beq      1$                    ; no error
;
;
;   read error
;
;
     inc      stopf                ; tell program to stop
     .print   harder                ; error on read
     .wait    mqwrit                ; wait for previous i/o on MQ
     .write   area, mqwrit, errpt, 1, 0 ; alert bg to error
     return                                ; and return
;
;
;   no read error
;
;
1$:  mov      buffa,r1                ; get the buffer address
     mov      128.,r2                ; buffer length
     call     dohist                ; do the histogram update
     tst      stopf                ; check stop flag
     bne      2$                    ; no more data
     .readc   area, abchan, buffa, 128., adone, 0 ; Start buffer a reading
;
;
2$:  return                                ; and return
     dec      readc                ; decrement count
     return                                ; and leave

```

```

        .sbttl  Buffer b completion routine
        .page
bdone:  call    savres                ; save all registers
        bit     herr,r0              ; check for error
        beq     l$                    ; no error
;
;
;      read error
;
;
        inc     stopf                ; tell program to stop
        .print  harder                ; error on read
        .wait   mqwrit                ; wait for previous i/o on MQ
        .write  area, mqwrit, errpt, 1, 0 ; alert bg to error
        return                                ; and return
;
;
;      no read error
;
;
l$:     mov     buffb,r1                ; get the buffer address
        mov     128.,r2                ; buffer length
        call   dohist                 ; do the histogram update
        tst    stopf                  ; check stop flag
        bne    2$                      ; no more data
        .readc area, abchan, buffb, 128., adone, 0
; Start buffer a reading
; and return
2$:     return
        dec     readc                  ; decrement count
        return                                ; and leave
        .sbttl  Do the histogram update
        .page
;
;
;      r1 contains the buffer address
;      r2 contains the number of words to do
;
;
dohist: mov     (r1)+,r0                ; get the value
        asl    r0                      ; word offset
        inc    hist(r0)                 ; bump the bin value
        sob   r2,dohist                 ; and repeat
        return
;
;
        .sbttl  Zero the histogram
        .page
zero::  mov     16384.,r0                ; get buffer size
        mov     hist,r1                 ; get address of buffer
l$:     clr     (r1)+                    ; clear a word
        sob   r0,l$                     ; and loop
        return
ztext:  mov     10.,r0                  ; ten words
        mov     commnd,r1                ; address of buffer
l$:     clr     (r1)+                    ; clear the buffer
        sob   r0,l$                     ; and loop
        return
        .sbttl  Error exit
        .page
error:  .print                                ; say the error
        .exit

```

```

        .sbttl  Data and Storage
        .page
        .psect  data
ab:      .rad50  /AB           /           ; data acquisition handler
jobdes:  .rad50  /MQ/
        .word   0,0,0
bg:      .rad50  /MQ/
userj:   .word   0,0,0           ; users job name
dfile:   .rad50  /DATHISGRMDAT/       ; DAT:HISGRM.DAT
sblk:    .blkw   4               ; status return area
area:    .blkw   10.            ; emt area
commnd:  .word   0               ; read buffer
comtxt:  .blkw   10.            ; command coming up
writen:  .ascii  /W /           ; data written
errpt:   .ascii  /E /           ; error on read
stopf:   .word   0               ; stop flag
readc:   .word   0               ; outstanding read count
atime:   .word   0,5            ; 5 ticks
btime:   .word   0,4            ; 4 ticks
buffa:   .blkw   128.           ; a buffer
buffb:   .blkw   128.           ; b buffer
hist::   .blkw   16384.         ; 16-bit buffer
        .psect  msgs
sterr:   .asciz  /?ACQUIR-F-Handler is not in system/
noload:  .asciz  /?ACQUIR-F-Handler not loaded/
look:    .asciz  /?ACQUIR-F-Lookup error/
mqopen:  .asciz  /?ACQUIR-F-MQ lookup error/
nocom:   .asciz  /?ACQUIR-I-Unrecognized command from MQ/
ulook:   .asciz  /?ACQUIR-I-MQ lookup error to user job/
reade:   .asciz  /?ACQUIR-F-Read error to handler/
harder:  .asciz  /?ACQUIR-F-Hard error detected by handler/
enterr:  .asciz  /?ACQUIR-F-Enter error for data file/
tst:     .asciz  /?ACQUIR-I-.DSTAT call complete/
start:   .asciz  /?ACQUIR-I-Program started/
wait:    .asciz  /?ACQUIR-I-Waiting for data/
        .end

```


4.5 Data Collection Handler

The handler to acquire data from the DRV-11W is shown below.

```

.title AB.SYS
.sbttl ADC/842/DRV-11B DMA Device Driver for RT-11 Version 5.2
.ident /V1.0/
.enabl lc
;
;
; AB is a DMA device driver the ADC chain number one for use under
; RT-11 Version 5.2. This driver reads data from an external nuclear
; type ADC interfaced to a DEC DRV-11B via a TEC model 842. This
; is a re-write of the original handler authored by Paul L.
; Robertson 29-Jun-82.
;
;
.sbttl Preamble section
.page

.mcall .drdef
;
;
This device is assumed to have a CSR of 172410
and a VECTOR of 124
;
;
ab$wct = 172410 ; Word count register
ab$adr = ab$wct + 2 ; Bus address register
ab$csr = ab$wct + 4 ; Control and status register
ab$io = ab$wct + 6 ; Input/output buffer
;
;
Function definitions
;
;
go = 1 ; Go bit
fun1 = 2 ; Function 1
fun2 = 4 ; Function 2 and Init for user device
fun3 = 10 ; Function 3
ie = 100 ; Interrupt enable
.drdef ab,377,<ronly$>,0,ab$csr,124
.sbttl I/O Initiation Section
.page
.drbeg ab
mov abcqe,r4 ; Get pointer to queue element
mov q$wcnt(r4),r0 ; Get word count
bmi hderr ; write, which is an error
beq abdome ; seek, which completes
;
Request validated, we are reading
;
;
mov fun2,@ ab$csr ; Reset the device
neg r0 ; Make word count minus
mov r0,@ ab$wct ; Load the word count
.if eq,mmg$t ; Not XM
mov q$buff(r4),@ ab$adr ; Load the address
mov ie!go,@ ab$csr ; And start the device
.iff ; XM
cmp (r4)+,(r4) ; Advance to buffer pointer in QE
call @$mpttr ; Convert virtual to physical
mov (sp)+,@ ab$adr ; Load the lower 16 bits
mov (sp)+,r0 ; Get the high order bits
bit 1700,r0 ; Check for a 22-bit address
bne hderr ; Hard error - 18-bit device
bis ie!go,r0 ; Add in interrupt enable and go
mov r0,@ ab$csr ; Load and start it
.endc
return ; And return

```

```

        .sbttl Interrupt Section
        .page
        .drast ab,5,abdone
        tst    @ ab$csr          ; Any error?
        bmi    hderr            ; Yes
        br     abdone          ; And exit
        .sbttl Hard error
        .page
hderr:  mov    abcqe,r4          ; Reload pointer
        bis    hderr$,@-(r4)    ; Set a hard error
        br     bye              ; And exit
        .sbttl Done routine
abdone: mov    fun2,@ ab$csr    ; Reset the device
        clr    @ ab$csr        ; Stop everything
bye:    .drfin ab              ; And return to monitor
        .drend ab              ; End of handler
        .end

```

5 CONCLUSIONS

This paper has shown a typical data collection routine for the RT-11 environment. RT-11 provides a flexible data acquisition environment when coupled with the Extended Memory monitor. System throughput can be maximized through the use of "jam queueing" queue elements to a device handler. And, virtually overlaid programs make good use of the extended memory that is attached to an LSI-11.

RANDOM SYSTEM CRASHES
or
"BUT IT WAS PROPERLY GROUNDED!"

Thomas J. Shinal
Vice President
General Scientific Corporation
Rockville, Md.

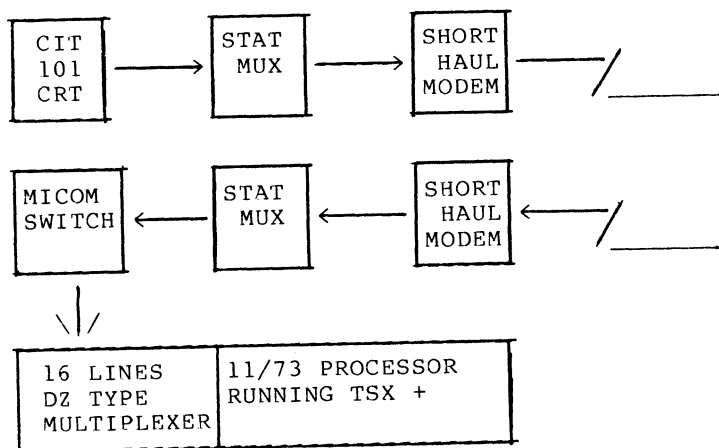
1 OVERVIEW

The following scenario is a composite of a few of our PDP-11 sites and therefore, I've taken some liberties in the actual situation. However, the data as presented actually happened despite all of my precautions.

The terminals are as far away as two miles from the CPU and connect by statistical multiplexors and short-haul modems via a Micom or a Gandalf Digital Switch to the DZ multiplexor.

The equipment involved is based around the 11/73 processor with 1 Mbyte of memory, 80 Mbytes of Winchester Disk, 800/1600 BPI tape, 16 channels of DZ multiplexing and 8 printers on DLV 11Js. The operating system is RT-11 with TSX+. All software and all hardware is up to current revision level.

The terminals are C-ITOH 101s and are connected to the DZ's in the following configuration.



2 SYMPTOMS

We would be getting random system crashes or halts. The problems showed up as being seasonal. Late Fall, we noticed an increase of halts and crashes to random locations. As they occurred over a number of systems, it was difficult to pinpoint. Hardware was eliminated by systematic component swapping, even to the power supplies and backplanes with no relief. That left only the operating system as the culprit. As S & H was far enough away, they seemed easy to blame. Many, many calls to S & H resulted in extensive

cooperation from them as to fault isolation. But, alas, there was no common thread. We modified handlers and inserted breakpoints to try to isolate. Symptoms looked like the Disk Handler was possibly at fault. We would get TSX messages indicating that the disk had become unready and other cryptic messages like that. Investigation proved that this wasn't the case. Other times we would lose data going out to the printers via the DLV-11J ports (They also shared the Stat Mux's).

This continued for months. A crash every other day might not sound like much until you multiply that by the number of systems installed and the workload affected. These are not software development machines, but production systems in a Government installation. Tempers were getting short and future business was seriously questioned. We were living at the site with logic analyzers, scopes and the such. At one point, after many nights of frustration, silliness set in and we tied balloons to the four corners of the cabinet to help to "KEEP THE SYSTEM UP".

3 GREMLINS UNMASKED!

Two problems surfaced..

The first problem showed up under scrupulous examination after noticing a light flicker from a test lamp. We checked the AC power and we found "GLITCHES"... What we saw was a power brownout that lasted about a cycle and a half. This was enough to tell the power monitoring circuitry that a "hit" was on its way and to start an orderly shutdown. The system did exactly what it was supposed to do.. It shut down.

NOTE

BPOK H -- The assertion of this line indicates that there is at least an 8 ms reserve of DC power and that BDCOK H has been asserted for at least 70 ms. Once BDCOK has been asserted, it must remain asserted for at least 3 ms. The negation of this line indicates that power is failing and that only 4 ms of DC power reserve remains.

Many if not all of non-DEC system chassis' have larger power supplies and may or may not have the same timing requirements. Our particular system at that time had a linear supply which was under load specifications, however the supply did not meet specifications (later replaced with a switching supply of larger capacity. Backplane problem also).

The system power recovered prior to the Power-Fail message showing up. By the time the operator saw a problem, power had restored with no indication as to what had happened. The system was sharing a power main with a large motor somewhere in the building. Perhaps it was even the freight elevator. As it takes the Government months and months to respond with a procurement for a Power Line Conditioner (Procurement Red Tape), we cured the

"Symptom" by disconnecting the AC monitoring circuits from the backplane (BPOK H). We lost our protection in the event of a real power hit, but that was the tradeoff until conditioning could be done. It hurts on possibly negating disk shut-down protection, but it didn't affect ours as they had their own protection circuitry. Nothing else was to be gained as without battery back up, there was nothing to save in event of AC failure. The BPOK H was a liability.

The second showed up as building heat was turned on in the Fall. We all know the problem of static. We suspected that there was also a problem in the computer room however, a new room with proper humidity control and grounding was just around the corner. With the new installation, we felt at ease... for a day or so.

We were told that the room was properly grounded. The floor grids (raised computer floor) were properly tied to the building power grounds. We checked. We discovered that the ground that was referenced was not grounded at all but simply the power neutral wire. Any unbalance on the power lines could and probably did introduce noise into our system.

The next step was to install power line conditioners that would totally isolate the systems from the power grid. We insisted on our own ground rod. Each of the system sub chassis' were individually tied to a common point on each cabinet. Each cabinet was individually wired to the ground rod. Heavy gauge tinned braid (3/4 inch) was used throughout. The braid should be insulated. RF grounding techniques were employed to eliminate any possible RF Ground Loops. The ground rod was buried in a moist area in a salt impregnated earth for good conductivity. We did all of the right things. We thought that we were home free. Systems weren't crashing as much and things were looking up. We still had a few random crashes, but we were free to attack them now.

4 IF I HADN'T SEEN IT MYSELF!

I was getting complaints from the farthest user from the system. My field service personnel had visited the site many times and hadn't found anything. I stopped in on a whim. As I entered the office, a CRT operator greeted me with an Ouch! What Happened? She got a static shock from the keyboard. Further investigation showed that the Stat Mux was down as well as the CRT. I called the Computer Center and sheepishly asked if System II had just crashed. You can guess the answer. If one

of my technicians had given me that story about static crashing the system from two miles away via stat mux's and short haul modems not to mention the digital switch, I'd be inclined to pink slip him. But... being of higher rank, I found it in my heart to forgive me.

The entire system was floating. The AC lines in the buildings had high impedance grounds. The buildings are old and many of the receptacles although three prong, were not grounded. We, being the good guys with the excellent grounding (<2 ohms) became the current sink for the Base. This further manifested itself whenever the Micom Switch was powered up from a maintenance shutdown. Every one of our systems crashed. Micom confirmed that they generate "some noise" on power up and will crash some systems.

We were the good guys that became the bad guys. The customer hinted that because we were crashing, we must be at fault. What to do? Un-ground us and float with the rest of the world? Ground them (Not in the budget)?

The only thing that helped was that other computers (PRIME, BBN, CDC) were crashing also. We didn't know this for awhile. We wore blinders and saw only "our" problems. Stepping back and looking at a larger picture might have saved us all some grief.

5 SOLUTIONS

1. Customer recognition of the problem.
2. Power Line Conditioners
3. Proper Grounding of System Components
4. Proper Grounding within the buildings
5. Anti static sprays at the terminal areas
6. Optical isolators on every communication line totally isolating data as well as isolating ground lines
7. More System Support Balloons

Harry Haenen
 Dept. Clinical Neurology and Information Processing
 University Hospital Groningen
 P.O. Box 30.001
 9700 RB Groningen, The Netherlands

ABSTRACT

A multiprocessor network concept is described and its implementation under RT-11. The multiprocessor concept may be seen as alternative to using a multi-user single processor system. However, the multiprocessor option has multiple CPU power and memory available over a single processor system. With decreasing hardware prices, the multiprocessor is the better solution especially in highly demanding environments such as high speed data acquisition and processing. The datacommunication software provides transparent use of remote devices. Memory-only systems may be run using a remote system disk.

INTRODUCTION

Data processing often starts with a single CPU system. A multi-user operating system then seemingly makes CPU and other peripherals available to multiple users. However, with the advent of newer user-friendly software like screen editors, graphics etc. CPU load increased and responsiveness often decreased considerable. A concept with multiple systems, connected by high speed datacommunication links can face the higher demand. Shared disks, printers etc. assure that data are available to multiple users and that expensive peripherals do not run idle for longer periods.

In the laboratory a multi-user system is often inappropriate. High speed data acquisition may block the whole system and frustrate other users. Undisturbed processing may now be realized by giving each application its dedicated processor. Again, high speed datacommunication links assure that expensive peripherals are not needlessly duplicated, that data may be shared and realise parallel processing (multiple CPU power for a single job).

The multiprocessor goal is believed to have been closely approximated with the package here presented. An earlier version was already described elsewhere [1] (reprints available on request).

In the remainder of this article DC will be used as an abbreviation for datacommunication.

CONCEPT

The multiprocessor concept should fulfill the following requirements:

- High speed communication: remote systems should seemingly be close. The data amount to be stored or processed elsewhere may be quite large,

therefore the transfer speed should be high. Low cost as well as more sophisticated (DMA) hardware should be supported.

- Low overhead, simple communication protocol. This contributes to high speed and may keep CPU load low during transmissions.
- Any network topology may be realized: from the simple point to point connection to complex structures.
- No modification of standard system components: all software should be realized within programs and handlers (device drivers).
- Hardware dependent code should only appear within handlers.
- No arbitration in who issues a transfer request. One site should always be "listening" to the other.

A basic point to point connection is symbolically represented in Fig. 1. System A always issues the transfer requests. It has a DC handler which controls the physical data link. System B has continuously running a so called DC service job (task), which is ready to serve requests from the DC handler at the other side. Note that although the

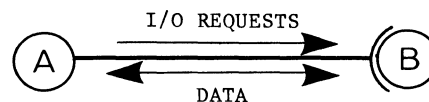


Figure 1. Data link concept

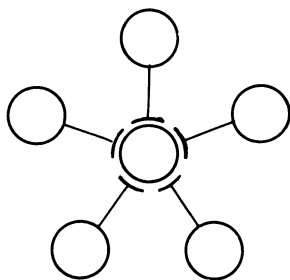
I/O requests go in only one direction, the data go in both directions. With a data read A receives data from B and with a data write A transmits data to B. Besides I/O requests for data transfers, also special function requests may be issued by A. For example by issuing a special function request, A could ask B to return the size of disk unit. Logically there will be several channels within one data link. Each channel is then used to allocate a device unit or file or used to perform a special operation. For example one channel, the message channel is reserved for the exchange of messages ("mail") between A and B. For that purpose B has reserved a "mailbox" file, which stores news for A as well as B and messages received from A for B and visa versa. Such a message from A for B may be e.g. a request to give read and/or write access to a certain device unit.

In order to safely transfer data over the link, a datacommunication protocol is needed. The protocol assures that both sides of the link "understand" what the other is "doing". Also the data integrity can be guarded by applying an error detection algorithm over all data received.

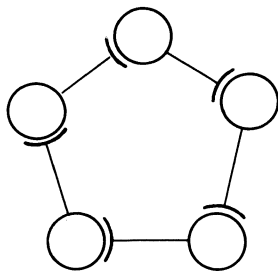
With the basic link now defined even more complex networks can be set up as shown in Fig. 2.



Symmetric link



Star



Ring

Figure 2. Basic network structures

THE RT-11 LINK

The DC service job runs under RT-11 as a Foreground or a System job. In a monitor with system job support up to 7 DC jobs may run simultaneously. The DC service job links to the DC hardware by using a handler, called the "job-handler". A DC job is now started with the following commands:

```
.LOAD QJ                ! Load the job-handler
.ASS QJ JOB             ! Name it logically JOB:
.FRUN/BUFFER:nnnn DCJOB ! Run the DC job
```

next job:

```
.LOAD DJ
.ASS DJ JOB
.SRUN/BUFFER:nnnn/NAME:DCJOB1 DCJOB
```

etc.

When the /BUFFER:nnnn is specified an extra data buffer of size nnnn words is allocated to the job. This buffer is added to the default internal buffer of 256 words (1 disk block).

The complement of the DC service job is the handler driving the DC hardware at the other side of the link. The protocol used by this handler and the job-handler is a modified Radial Serial Protocol (RSP, [2]). This protocol basically transfers words of data and is described in detail in [1]. The RSP protocol can maintain up to 256 data channels over one link. The current implementation uses only 15 of these channels as these channels are one-by-one coupled to an I/O channel in the service job. Although RT-11 allows defining up to 256 I/O channels for each job, 16 is the default number. As one channel (#15) is used by the job handler, 15 remain to be used for allocating devices. Each data channel has a number 0-14 which also will be called the RSP unit number. An example of DC data channel allocation is given in Fig. 3. As a handler has maximum 8 device units (0-7), only the first 8 data channels can be controlled by the DC handler. Normally the DC handler is defined to the system as a random access device (disk) and therefore cannot be used to simulate e.g. a remote lineprinter.

Both problems, accessing the higher data channels 8-14 and simulating several different type devices are solved with the introduction of pseudo-handlers. These are handlers that do not drive hardware themselves, but use the DC handler for that purpose. The DC-handler has provisions for receiving requests from pseudo-handlers: an internal queue. I/O requests from the DC handler itself and from the pseudo-handlers are stored in this queue and removed from the queue when they are served. These pseudo-handlers make it also possible to use also the data channels higher than 7. For example a service job has allocated channel #10 to a lineprinter. This channel may now be accessed by a pseudo-handler which transforms a request received on device unit number 0 to a RSP unit number 10 by adding the value of 10 to the device unit number. This pseudo-handler may also be defined to the system as a standard lineprinter so that programs and RT-11 utilities cannot "see" the difference between a real lineprinter handler and the pseudo lineprinter handler. Note that the service job may also allocate channels to a DC handler within the same system. In this way devices on all connected

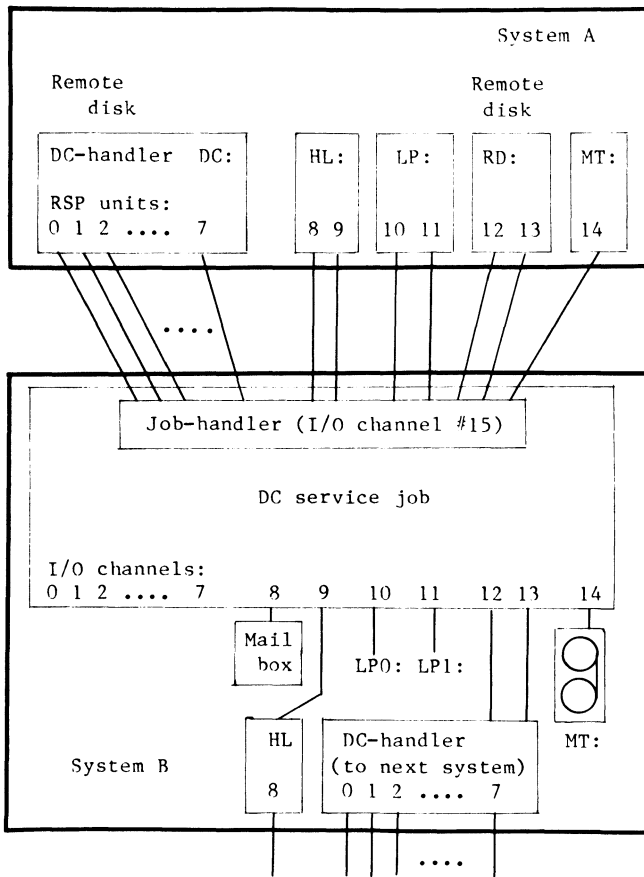


Figure 3. DC Data & I/O channels

systems may be accessed, even when there are intermediate systems.

Currently the following hardware is implemented as the physical data link:

LSI-11: DRV-11 (DEC)
 WBV-11 (Buchholz / Hammond, Germany)
 Qnector (Westvries Systems, Holland)
 (DMA-interface)

PDP-11: DR-11C (DEC)
 DR-11K (DEC)
 WB-11 (Buchholz / Hammond, Germany)

However, as all hardware dependent code is in the DC-handler and job-handler, implementation of new hardware only requires that that these handlers are adapted to drive that hardware. The general purpose parallel interfaces DRV-11, DR-11C, DR-11K require simple electronic circuitry in order to form a handshake connection, which can be used on interrupt bases. Only one FLIP-FLOP and one OR-gate are the needed components [1]. In the first version also DL(V)-11 type hardware was supported. However, this was dropped as transfer rates are low and DEC supports these data links within the RT-11 package (programs VTCOM and TRANSF, handlers XL, XC).

The size of the DC service job is about 1.5 Kw

including it's 256 word default buffer and the job-handler. The size of the DC handlers is about 700 words and that of the pseudo handlers ca. 50 words. Default the DC job handles read, write, special function (.SPFUN) and boot requests on all channels. However, one channel the message channel, is special purpose and used for message transfer to the user on the other system. These messages and news are also put in a mailbox, a file which is present on each system that runs at least one DC job. The message channel is also used for transfer of date&time from system to system and DC job configuration data (list of remote devices, read/write access).

Normally the job tries to open an I/O channel to all devices in a list, the job device list. This requires that these devices are loaded. One exception on this rule are Special Directory devices such as Magtape. In fact, these devices have no directory and require special operations to open a file. Therefore when an "open file" request (.LOOKUP, .ENTER) is received, this is forwarded to the device itself and not processed by the local USR as for disk devices. Handling Special Directory devices requires additional code in the DC job. This code, including handling "asynchronously directory" operations for Magtape (see RT-11 Software Support Manual), is about 200 words in size. This code may be selected at assembly time by setting a conditional in the DC job source. So two versions may be kept at hand: one that supports all devices (DCJOB.SPD) and one that supports all but Special Directory devices (DCJOB.REL). At the other side of the link a pseudo Magtape handler is available which behaves like a normal Magtape handler but is much smaller in size (ca. 180 words).

USING THE DATA LINK

As already pointed out: remote devices are used in the same way as local devices are used! However, pseudo-handlers use the DC-handler and this requires that the DC-handler must be loaded when using a pseudo handler! When it's DC-handler is not loaded a pseudo-handler immediately returns a hard I/O error.

A utility HELLO can be used to send a message to the user at the remote system. It also checks whether there is a difference between remote and local date&time. Further it prints which remote devices are available, the device's characteristics such as size, identifier (helps you to "see through" logical assignments at the DC jobs site), etc. and read/write access to the remote devices. The JBDATE utility is very usefull in the startup command file as it copies remote date&time and sets them locally. When at a site one or more DC jobs have been started, the JSHOW utility should be run. It displays the following data:

.JSHOW

JOB	Hndlr	Nr. I/O requests	Checks errors	Protoc errors	Buffer size	
DCJOB0	(0,QJ)	9K 950	0	0	1024	No SPDIR
DCJOB1	(1,DJ)	12K 372	0	0	256	No SPDIR
DCJOB2	(2,DI)	not running!				

SHOW ALL / r&w / change r&w / exit :

No	Device name	iden.	size	characteristic
0	DLO:SYSII .DSK	FILE	4800.	* FILE *
1	DLO:SOURCE.DSK	FILE	4800.	* FILE *
2	DLO:VER .DSK	FILE	4800.	* FILE *
3	DL1:ERPROG.DSK	FILE	4800.	* FILE *
4	DL1:DATBAS.DSK	FILE	4800.	* FILE *
5	DL1:ERP .DSK	Not in system
6	DLO:DIPOL .DSK	FILE	1200.	* FILE *
7	DL1:ER .DSK	FILE	1200.	* FILE *
8	SY :JBINFO.DAT	FILE	16.	* FILE *
9	HL :	.	377	0. SPFUN
10	SP :	.	LP	0. WONLY
11	DL0:	.	DL	20450. FILST SPFUN VARSZ
12	DL1:	.	VM	384. FILST
13	DM :	.	DM	53724. FILST SPFUN VARSZ
14	MT :	.	MT	0. SPECL HNDLR SPFUN
15	JOB:	.	300	0. SPFUN

Press RETURN to continue

Central device	DCJOB0	DCJOB1
0 LD0: sys 11/23	R W	R -
1 LD1:	R -	R W
2 LD2:	R -	R -
3 LD3:	R -	R -
4 LD4: Datbas	R -	R W
5 LD5:	N	N
6 LD6:	R -	R -
7 LD7:	R -	R -
8 JBINFO Mailbox	R W	R W <=
9 HEL	R W	R W <=
10 SP: Spooler	- W	- W <=
11 RD0: Remote disk	R -	R -
12 RD1:	R -	R -
13 RD2:	R -	R -
14 MT: Magtape	N	R W
15 JOB handler	R W	R W <=

Note: R=read, W=write, N=no device

Show all / r&w / CHANGE R&W / exit :

Change ?

Give JOB nr. :1:
 RC, RS, WC, WS :WS:
 Device no.(0-14):13:

Change ?

Give JOB nr. : :

Show all / R&W / change r&w / exit :EXIT

With this utility the read and/or write access to a device for a certain job can be changed. The number to the left in the device list is the DC channel number (also RSP unit number). Note that although on RSP unit no. 12 the device DL1: is specified, this device in fact is the VM: disk ! This is because the logical assignment ASSIGN VM DL1 has been made before the DC jobs where started!

A job is simply stopped by aborting it:

.ABORT DCJOB1
 .UNLOAD DCJOB1

SPECIAL FEATURES

One special feature is using a remote disk as system disk. When a remote disk is made bootable for the DC-handler (e.g. DC:) and contains the necessary systems components such as monitor file and utilities, it can be booted with the standard command:

.BOOT DC:

The disk may have been made bootable before with the command:

COPY/BOOT DC:RT11FB DC:

However, it could also have been made bootable at the disk's site with the command (assume that the disk's name is DK:):

.COPY/BOOT:DC DK:RT11FB DK:

The ability to use a remote system disk is a very powerful feature because it allows using memory-only (or better: no disk!) systems! In order to serve memory-only systems, the DC job also acknowledges a boot command. When such a command is received, it transmits a block of data (256 words, without protocol header and tail!) : the BOOT program. Sending the boot command to the DC job can be done by a small program which has been put in (P)ROM. However, it could also be typed in (toggle-in boot) using CPU-ODT available on most machines. When the BOOT-program is received and activated, it asks a password. When the correct password has been entered, it asks the unit number of the bootable disk, fetches the bootstrap from that disk and activates it. RT-11 will then come up. When using a disk data cache also multiple systems can use the same (remote) system disk [3,4].

Another feature is parallel processing. By this is meant that data are transferred from the memory of one system directly to the memory of another system. The data can then be processed in parallel by both systems. This feature is realized by using a special purpose, internal queuing, handler (refer to: "Internal queuing handlers" in Chpt. 7, RT-11 Software Support Manual). This special purpose "parallel" handler can accept a next write request before a previous read request is finished [1]. Therefore this handler can transmit data from a buffer within one job to the buffer of another's job.

Display in detail of the activity of a DC job can be realized using a device I/O logging&display package [3]. The packets transmitted and received can in this way be monitored by selecting the job-handler as the device under investigation. The I/O display is activated by loading a special handler and running another system job (SHOWIO or LOGG). Also a test version of the DC job may be generated. This job prints a "C" for each command-packet received, a "R" for each data-packet received, a "S" for each data-packet transmitted and an "E" for each END-packet transmitted. After an "E" the next characters are printed on a new line.

Not a feature but more a problem is that of "job blocking". As the DC job runs as Foreground or System job, it runs at a higher priority than the Background (BG) job. This may be one of the causes

that intermittent problems occur when the BG job does high speed A/D conversion, while the Foreground is also active. The A/D converter may report errors and samples may be lost. In such a situation it would be desired to block the DC jobs until all time critical activity of the BG is stopped. Until now RT-11 has no provisions for such a facility. Therefore the following "trick" is used. A set of subroutines, to be used in a BG program, can block/suspend, unblock/resume DC jobs. When job blocking is required a "no wait" .SPFUN request is sent to the job-handler. When the job-handler receives this request, it accepts it but does nothing. This means that no other I/O requests, those from the DC job, can enter the job-handler and the DC job is thus blocked. The BG can in the meantime process it's critical tasks. When the BG wants to resume the DC job it aborts the "hanging" .SPFUN request. The DC job I/O requests can now enter the job-handler and so can resume it's activity.

INSTALLATION OF THE SOFTWARE

First of all the DC job programs DCJOB.REL and in case of special directory support DCJOB.SPD, the utilities and pseudo-handlers are copied to the system disk. The DC jobs and utilities may also reside on another available disk unit. The pseudo-handlers may also be renamed to a more appropriate name. The DC and job-handlers should be inspected for having the correct I/O page and vector addresses. There is also an option, selected by a conditional, for disabling the checksum calculation. When disabled, a fixed bit pattern is transmitted, instead of the checksum, as the tail of each packet. When a packet is received the bit pattern is checked. Although this procedure assures some minimal error detection, data corruption within a packet is not noticed. However, in practice, there are many physical data links which show up seldom an error. And when it occurs, it comes in bursts so that these errors are detected in any case. When the handlers are assembled (with system conditional file SYSGEN.CND) and linked they are copied to the system device.

The DC jobs require that a list of devices is available to which they should open I/O channels at startup. They expect to find this list within the job's data file SY:JBINFO.DAT . Within this file further are stored: default read/write access settings (may be changed while DC jobs run with JSHOW), which channel is the message channel and which reserved for Magtape, a list of the names of available job-handlers, bootstrap programs for memory-only processors and the mailbox.

The file SY:JBINFO.DAT can be created and the data in it are set by the program JOBS. All the modifiable data mentioned above (device&job lists, read&write default access), are stored in a readable format in the file JOBS.CND. Using an editor they may be changed to the appropriate values. Then JOBS should be assembled, linked and run once. During the assembly phase JOBS.CND is read and processed.

PERFORMANCE

The performance of the data link was measured for all hardware types. For this purpose a dummy handler was constructed comparable to the null-handler (NL:). This dummy handler immediately satisfies any I/O request that it receives, but does not perform any data transfer from or to a buffer. Data were transmitted in records of 1024 words (4 disk blocks). As the protocol allows the transfer of max. 256 words/packet [1], four data packets are transmitted for each record. Before these packets are transmitted a command packet is sent and after the four data packets an End packet is received. The throughput rates in the table are effective rates. This means:

including the protocol overhead just described,
 + transfer from memory to interface by DC handler,
 + transfer over the cable (20 m.),
 + transfer from interface to buffer of DC job,
 + 6 I/O requests by DC job to interface handler,
 + 4 I/O requests to dummy when buffer job is 256 w.
 (1 I/O request to dummy when buffer job is 1024 w.)

TRANSMISSION RATES in Kw./s.

Buffer:	No checksum		Checksum calculation	
	1024	256	1024	256
<u>Qnector:</u>				
11/23	35.7	31.3	22.7	21.3
+	(30)	(28)	(55)	(50)
11/23				
<u>WB(V)-11:</u>				
11/34 ---	12.3	11.8	11.9	11.6
+	(60)	(60)	(73)	(73)
11/23 ---	(74)	(74)	(86)	(86)
<u>DR-11:</u>				
11/34	16.7	15.9	14.7	14.3
+	(92)	(88)	(94)	(92)
11/34				

Note that in the table the machines linked, differ. The PDP 11/34 is in many respects about 20% faster than the LSI-11/23. The Qnector was not set to it's highest speed because of high bus load. However, at it's highest speed a throughput rate of 67.2 Kw./s. (= 1 Mb.) was measured. The hardware specifies max. 250 Kw./s. (= 4 Mb.). Therefore it is demonstrated that the often impressive throughput rates specified by manufacturers do not tell much about the effective throughput under software control! The values between the parenthesis give the CPU load in % during the transmission at the DC handler side. The CPU load at DC job site shows nearly the same values. Note that these values apply to the test situation! Under "normal" circumstances, where I/O's have to be processed by devices, the CPU load measured is considerable lower (20-40%) !

CONCLUSIONS

A collection of programs and handlers realises multiprocessing and high speed data communication with RT-11. Remote devices are used in the same way as if they were local. The well-structured software allows all type networks to be setup. Low cost as well as high performance hardware is implemented. Moreover implementing new hardware is a relative small task as only two handlers have to be coded. Cheap memory-only systems can be put to work due to boot capabilities.

REFERENCES

1. Haenen, H.T.M.
"A Modular Data Communication Package Providing a Multiuser Environment and Parallel Processing"
Proceedings DECUS EUROPE
Coventry U.K., Sept. 1982, pp. 81-88
2. The "Radial Serial Protocol (RSP)".
Microcomputer Interfaces Handbook. DEC 1980, p. 640
3. Haenen, H.T.M.
"Disk Usage Analysis and Disk Data Caching under RT-11"
Proceedings DECUS EUROPE
Zuerich, Switzerland, August/Sept. 1983, pp. 247-252
4. Haenen, H.T.M.
"The Disk Data Cache under RT-11"
Proceedings DECUS U.S.A.
New Orleans, Louisiana, May 1985

RSX-11 SIG

Arnold Scott De Larisch
 Florida Atlantic University
 College of Engineering
 Department of Electrical Engineering
 Boca Raton, FL 33431

ABSTRACT

This paper is written with the new system manager's needs in mind. Topics include creating new user disks, making use of SYSLOGIN.COM file facilities, tailoring a system to provide a more user friendly environment, controlling system access on dial-in lines, and providing safe use of privileged accounts by system operators.

1. CREATING NEW USER DISKS

One of the first post-sysgen necessities on most systems is the creation of public disks. This process has several steps and requires the system manager to make several important decisions. The process starts with the formatting of the disk pack. All new packs should be formatted to insure the media's integrity. The task responsible for the formatting process is called FMT. FMT writes a complete header for each sector of the volume and then by default the address contents of each sector header is verified. The FMT utility as well as some other privileged tasks can be very hazardous to a working system. If a disk is spinning and not allocated, any user can run FMT on the pack and destroy its contents. Therefore great care should be taken when using the FMT utility.

There are various switches available from which to choose many options. The /WLT switch (which stands for Write Last Track) is one such option. This is where the Manufacturer Detected Bad Sector File (MDBSF) is located. When formatting DL type devices this switch is mandatory and is also highly suggested for use with DM and DR type devices. The /WLT switch requires a decimal serial number. If you have your disk packs under a service contract it is very important that you use the serial number located on the bottom of the disk pack when employing this switch. The /BAD switch which attempts to spawn BAD from FMT should not be used when formatting a disk pack. BAD is a privileged utility which can be very dangerous if left installed on a system.

The next step in preparing a user disk pack is to run the BAD utility on the disk pack. BAD is used to test disks for unreliable blocks and updates the last track information about the bad blocks. It writes to another last track file called Software Detected Bad Sector File (SDBSF). Both the SDBSF and the MDBSF are used when the INI command is used to create the [0,0]BADBLK.SYS file. Normally the /LI switch should be used with BAD, this lists the location of the bad blocks on the terminal. It should be noted that BAD does not prompt you before writing on the pack as FMT does, so extra care must be exercised when using this utility.

The final step in creating a user pack is to use Initvolume command which creates a FILES-11, also known as ODS-1 (On Disk Structure level 1), format on the disk pack. The INI command destroys all existing files, writes a dummy bootstrap, writes a home block and finally builds a directory structure. A required parameter is the volume label which is used as the password when MOUNTing the disk pack. The label should normally not be public knowledge since private volumes could be mounted by others if the volume label is known.

As with the other utilities there are some switches which should be considered by the system manager when using INI. The one switch which can greatly effect the disk performance is the /INDX switch. This switch specifies the location on the disk pack of the index file, the storage allocation file and the Master File Directory (MFD). In general, the index file should be placed in the middle of the volume which is the default location. The only noteworthy exception to this occurs on small volumes such as floppy disks in which maximum contiguous space

is sought. In the latter case either the /INDX=BEG or /INDX=END should be used. Furthermore, when initializing a new volume, careful consideration must be made with the default file protection.

The /FPRO=[System,Owner,Group,World] switch should be used to modify the default settings. The default access mask used when creating a new file on the volume is Read, Write, Extend and Delete privileges for the System, Owner, and Group. In addition the Read privilege is given to the World class, which is unsuitable for many settings including the university environment. Since users in the same course are generally placed within the same group there is nothing stopping one student from copying or deleting another's files.

2. CREATING A SYSTEM INDIRECT COMMAND DIRECTORY

The indirect command processor (@) is a rather powerful programming language available to RSX11-M/M-PLUS users. If you are in a development environment, you have probably created many useful command files. There is an easy way to have a system-wide indirect command directory which would be searched whenever a command file was not found in a user's local User File Directory (UFD). A system-wide indirect command directory also allows easy wild-card copying when transporting to a new disk after a new sysgen. It also reduces unnecessary duplication of command files, eliminates typing device and UFD specifications, and allows easy updating of command files.

To incorporate this feature the build file [1,24]ICPBLD.CMD must be slightly modified. The command file search begins in the user's own UFD and concludes in SY:[1,54]. To change the default from SY:[1,54] to another UFD one needs to simply decide which UFD is to be used on the system disk (that's the disk assigned to SY:). This UFD needs to be converted to an octal representation of the binary bit pattern of the Group and Member number. For example, if we wish to use [1,3] for the directory the following procedure can be used.

The group and the member number can each be represented in an 8-bit binary pattern as indicated below.

0	000	000	100	000	011	Binary
G	GGG	GGG	GMM	MMM	MMM	Group/Member
0	0	0	4	0	3	Octal

```
Invoke SYSGEN3 to rebuild ICP
Edit [1,24]ICPBLD.CMD
Find the line GBLDEF=D$CUIC:1
Change to new UFD GBLDEF=D$CUIC:000403
Do not forget to run VMR to
REMOve ...AT. and
INStall [1,54]ICP/TASK=...AT.
```

3. REBUILDING DCL TO FALL THRU TO MCR

To gain the most flexibility from the Digital Command Language (DCL) it should be modified to allow unrecognized commands to fall through to the Monitor Console Routine (MCR) / Command Line Interpreter (CLI). The modification needed to patch DCL involves once again invoking [200,200]SYSGEN3. Only this time DCL must be rebuilt and the CMD file must be edited. The build file to be modified is [1,24]DCLBLD.CMD. Search for the global definition GBLDEF=D\$CAT:0 and change the 0 to a 1 to enable fall through. Please note that when rebuilding this task the command file must taskbuild DCL twice which will take a fair amount of time (on an 11/24 with RK07s it takes about an hour). After the taskbuilding is done, you need to run VMR and REMove DCL and INStall the new version. Do not forget that the INStall must indicate that the task is a CLI thus the command would be:

```
INS [1,54]DCL/TASK=...DCL/CLI=YES.
```

Please note that there are some commands which appear in more than one CLI with different syntax. For example the SET command is valid for both DCL and MCR so if the user types SET /NOBRO=TI: from DCL it will return with a syntax error and not pass through to MCR since the SET command is valid for DCL. To use an MCR command from DCL you must prefix the SET with an MCR (i.e. MCR SET /NOBRO=TI:).

To give further functionality and convenience to your users a catchall program called TDX can be installed. TDX give several new commands and allows "flying installs" from a disk. This can free up valuable pool space since task headers are not in pool until the program is requested. Refer to the RSX-11M Release notes and Sysgen manual for further information.

4. CONTROLLING DIAL-IN ACCESS

Having a modem connected to any system leaves it open to "hackers" who have nothing better to do than to try to get into your system. There are several ways to deal with this problem.

The easiest way to remove the threat of outside intruders is to remove the modem. This, however, is generally not an acceptable action. Another approach is to write a security CLI which requires a password to even get to enter a command line requesting HELLO. Still another approach is to use [1,2]SYSLOGIN.COMD file to restrict access to the system. The mere existence of this file causes HELLO to invoke it, thus setting various parameters and terminal attributes. This happens after the intruder has given a valid username and password. The terminal attributes are then immediately set to slaved and privileged as well as having the MCR CLI forced on the terminal. This allows the system to execute privileged commands in a controlled manner. Furthermore there is a parameter called <LOCAL> which is set false if the line is a remote (modem) line. At this point you can ask for a password to further verify the person's identity. A simple example follows:

```
.ENABLE SUBSTITUTION
.ENABLE QUIET
.DISABLE DISPLAY
.ENABLE CONTROL-Z
.IFT <LOCAL> .GOTO NMOD
.SETS PAS "XXXXXX"
SET /NOECHO=TI:
.ASKS PAS MODEM PASSWORD >
SET /ECHO=TI:
.IFT <EOF> .SETT CTRLZ
.IFT CTRLZ .SETS PAS ""
.IFT CTRLZ .GOTO BADPAS
.IF PAS = "PASSWORD" .GOTO NMOD
.BADPAS: .DISABLE QUIET
;*** BAD PASSWORD ***
BYE
.EXIT
.NMOD: .IF P5 = "P" .GOTO SLAV
SET /NOPRIV=TI:
.SLAV: .IF P6 = "S" .GOTO OVER
SET /NOSLAVE=TI:
.OVER: CLI /UNOVR
.EXIT
```

The command file tests <LOCAL> and skips all the modem restrictions if true. The routine from .NOMOD: through .OVER: is necessary for the SYSLOGIN.COMD. Variable P5 indicates if the user should be privileged whereas P6 indicates if the terminal should be slaved. Finally, the CLI's override bit is cleared which allows the default Command Line Interpreter to be invoked instead of MCR. If the terminal is a remote line, the user is prompted for a password.

If the password is correct the terminal is logged in like any local terminal, otherwise a bad password message is displayed and the user is logged out. This should eliminate most modem line security difficulties.

This is just a small sample what can be done in the [1,2]SYSLOGIN.COMD file. I suggest some caution when initially making the file since all terminals are initially slaved at login. If there is a syntax error prior to the code which sets the terminal to noslave, there is no way of logging back on to the system. So you should always keep a second privileged terminal logged in when ever working on the SYSLOGIN.COMD file so you can unslave the terminal in case of difficulties.

5. AN EFFECTIVE USE OF SLAVED ACCOUNTS

One of the least used but most powerful security techniques is the use of a slaved account with a LOGIN.COMD file. Basically speaking, a slaved account is unable to accept unsolicited input. This allows controlled access to the command line interpreter. Thus operator accounts which need privilege to effectively do their work can be given access only to those commands and utilities in which are needed. Simple menu driven command files can be generated without a great deal of effort. A simple addition to the end (i.e. before the exit) of [1,2]SYSLOGIN.COMD will allow a local LOGIN.COMD to be executed.

```
.SETS FILE P1+P2+"LOGIN.COMD"
.TESTFILE 'FILE'
.IF <FILERR> = 1 .CHAIN 'FILE'/LO
.IF P6 = "S" .XQT BYE
.EXIT
```

The proceeding is an example of such code.

Within the user's UFD a file called LOGIN.COMD is executed. A sample is as follows:

```
.DISABLE DISPLAY
.ENABLE CONTROL-Z
.ENABLE DECIMAL
.ENABLE SUBSTITUTION
.OPEN TI:
.OPTION: .DATA OPERATOR COMMAND FILE
.DATA 0 EXIT
.DATA 1 SHOW USERS
.DATA 2 SET TIME
.DATA 3 SHUTDOWN
.START: .ASKS CMD ENTER NUMBER >
.IFT <EOF> .GOTO OPTION
.TEST CMD
.IFF <NUMBER> .GOTO OPTION
.ONERR OPTION
.GOSUB 'CMD'
.ONERR
.GOTO OPTION
.0: .XQT BYE
.EXIT
.1: DEV /LOG
.RETURN
```

```
.2:      .ASKS HR ENTER TIME HH:MM:SS >  
        TIM 'HR'  
        .RETURN  
.3:      .XQT $SSHUTUP  
        .RETURN
```

Once again this general slaved account concept can be expanded upon in many ways including adding an error trapping routine or even maintaining a audit trail of all commands entered. Using slaved accounts allows the system manager to minimize costly syntax errors typed by system operators. This concept can also be extended to generate captive demonstration accounts and even menu driven accounts for occasional users who are not DCL syntax experts.

As you can see from these few examples, RSX-11 allows a great deal of flexibility which allows the system manager to tailor the system to the needs of the users. This flexibility may be a little difficult for a new system manager to quickly grasp, but it is worth the time and effort learning these techniques.

ETHERNET DATA-ACQUISITION AND CONTROL SYSTEM*

Paul Elkins, MS H821
Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

An ETHERNET data-acquisition and control system has been developed for use on the RSX operating system. This system supports supervisory control, closed-loop control, data monitoring, and data recording. An RSX driver has been written for the DEQNA ETHERNET controller for use with this system.

INTRODUCTION

The Fusion Materials Irradiation Test (FMIT) control system¹ is a distributed system consisting of 12 nodes connected by ETHERNET.² The communication protocol used is locally defined and does not adhere to any of the current network standards. A list of all computer nodes in this system is shown in Table 1. Supervisory control of the FMIT linear accelerator is the primary function of this system; however, it also provides for monitoring of remote equipment, data recording, plotting, alarm reporting, etc. The system can be divided into two levels; the upper level uses the RSX11M (disk-based) operating system, whereas the lower level uses the RSX11S (memory-resident) operating system. Also in the lower level there is one node used as a Falcon system development node. The Falcon system will be discussed later in this paper.

TABLE 1
NETWORK NODES SHOWING CPU AND SYSTEM TYPES

Node	CPU	System
2	11/60	RSX11M
3	11/23(KDF11-A)	RSX11S
4	11/23(KDF11-A)	RSX11S
5	11/23(KDF11-A)	RSX11S
6	11/23(KDF11-A)	RSX11S
7	11/23(KDF11-A)	RSX11S
8	11/23(KDF11-B)	RSX11M
9	11/23(KDF11-B)	RSX11M
10	11/73(KDJ11-A)	RSX11M
11	11/73(KDJ11-B)	RSX11M
12	11/21+	Falcon
13	11/23(KDF11-A)	RSX11M

The functions of the various RSX11M nodes provide for program development, data analysis, control-console operation, and dedicated computing that is concerned with determining beam quality and position.

The two control consoles are identical and are used to remotely control the accelerator. Each console consists of the following equipment: a color-graphic scope with an overlaid touch panel and a set of four control knobs and their associated plasma display panel that is also touch sensitive.

The graphics scope displays menus, schematics, tabular data, plots, etc.; the overlaid touch panel selects new graphic displays, knob displays, and may be used to control remote equipment. Control knobs are used to adjust remote equipment. An operator assigns a knob to a piece of remote equipment; each time the knob is turned, the corresponding knob counts are sent to the appropriate equipment connected to a remote computer node. The plasma display panel is a 24-column by 12-line display that displays data and turns equipment on or off through its touch-sensitive surface.

The instrumentation nodes are CAMAC³ based and each has an LSI-11/23 (KDF11-A) processor plus 124K words of memory. These systems all run the RSX11S operating system down-line loaded from a central host. Each performs a surveillance (limit-checking) function, certain node-specific processes, and allows for some limited amount of local operator interaction through the console terminal. These nodes also process remote requests, such as those originating at one of the control consoles.

ETHERNET DRIVER

An ETHERNET⁴ driver was written for the DEQNA Q-bus ETHERNET controller module to enable connecting the various nodes listed in Table 1. Two versions were written: one for the RSX operating system and one for stand-alone systems used on single-board computers (SBC) such as the Falcon (SBC-11/21). Both drivers use the same formatted packet shown in Table 2 and can therefore communicate with each other. All protocol fields and many other parameters are defined in a set of macros contained in a macro library called XQMAC.MIB.

TABLE 2
PACKET FORMAT

Field name	Field size (bytes)
Destination address	6
Source address	6
Type	2
Word count	2
Destination task name	4
Source task name	4
Subtype	2
Sequence	2
Data	32-1486

This driver is intended to manage all ETHERNET traffic for a given node; therefore, it must never appear busy to the system. It should always be

*Work supported by the U.S. Department of Energy.

ready to receive and dispatch packets to waiting tasks or to save them in its internal queue if a QIO request is not currently pending. Typically, a task will have a read QIO posted, but in a normal control-system environment it may receive one or more messages while it is processing the current request. That is why the driver tries to save all incoming packets until they are requested by the owning task.

The driver occupies 4K words of memory; however, the code only uses about 1.5K, and the last 2.5K is for storing incoming packets until they can be dispatched to the appropriate receiving tasks. Depending on network traffic, if a packet is not requested it may eventually have to be flushed to allow the hardware to reuse the data buffer, which is accomplished by passing the packet to a system task that prints out the header portion of the packet on the console log device and thus frees the buffer for reuse.

The driver must be built for each node it is to be used on and requires three modules for this operation. The first two are standard RSX items and are the driver source code and its database; the third module is a file (NODEX.MAC where X is the node address) that defines the node's 14 possible addresses. The driver departs from standard ETHERNET address protocol and uses only the sixth byte position to indicate a physical address; however it does support broadcast and multicast modes. If the byte (in file NODEX) is positive, it is considered to be a physical address. This condition imposes a limit of 127 nodes for a system, which is sufficient for most control systems. If the byte is equal to -1, it is considered to be a broadcast address; if it is negative but not equal to -1, it is a multicast address. The driver uses the information in this file at power-up to initialize the DEQNA's address fields.

SEND PACKET

A task sends a packet by posting a write QIO after ensuring proper formatting of the packet. The driver first checks the packet for proper size; if it is too large, the packet will be rejected and the appropriate error code returned to the caller. If it is too short, its length will be simply increased to conform with the minimum ETHERNET packet size. After all packet-length checks are successfully completed, the preformatted packet will be transferred directly from user task memory through DMA hardware to an on-board RAM; at this time, the actual transmit operation is initiated. When the transmit operation is complete, the task's event flag will be set and its I/O status will be updated if these parameters were specified in the QIO that requested the send. The driver only initiates one transmit at a time and will not start the next transmit request until the last one is complete.

RECEIVE PACKET

A task receives a packet by posting a read QIO; if a packet is waiting for the task, it will appear as an immediate transfer. If a packet is not waiting, the transfer will take place whenever a packet arrives for that task. In any event, at the completion of a receive-packet transfer, the task's event flag will be set and its I/O status will be

updated if specified in the receive request. If a packet arrives with errors, they are categorized and counted, but a packet containing errors is never transferred to a user buffer. Priority is given to packet receiving over transmitting, where appropriate. In actuality, when a packet arrives, the driver finds the end of it (it may span several data buffers) and saves all status information and then makes an entry in the driver-receive queue. Each entry is four words and contains a link word, buffer descriptor pointer, and the receiving task's name. The entry is placed in the driver-receive queue to minimize the search time required when trying to match multiple received packets to multiple tasks. This procedure also makes it easy to flush a packet, if this should become necessary, by merely changing the task name in the driver-receive queue.

GET STATUS

A get-status request is initiated by issuing a QIO to get-link information (GLI) and a byte count. A task may get its node address by issuing this call with a byte count of 2, or may get all status information by requesting 30 bytes. The remaining 28 bytes consist of a count of all error-free packets sent and received, plus all hardware errors reported by the controller. The errors are divided into transmit and receive errors. A slight variation of this call allows a program to get the link information and to clear the error counters. This call is used by network-management tasks to read and clear network-error counters.

SYSTEM SERVICE

A minimum set of system services are required to make a distributed network useful. The services supported by this system are as follows:

- Program communication
- Down-line system loading
- Operator file transfer
- Remote file access
- Line watcher

Program communication is provided by the basic driver using the RSX standard QIO system directive; however, applications programs usually access this directive through a user-interface routine.

Down-line system loading is required by most memory-resident systems such as the RSX11S and the Falcon systems. This operation requires the cooperation of at least one program in each CPU; this system uses two programs in the host CPU. The loading process is always initiated at the remote node (the one being loaded) either directly, by the operator, or indirectly, by sending a break character etc., and thus forcing the execution of an EPROM-based code called the primary loader. The primary loader sizes and tests memory and requests the initial load from the host CPU. A fresh copy of the loader task is spawned so that multiple nodes can be loaded simultaneously. At the completion of the load process, the system is automatically started.

Operator file transfer is provided, allowing an operator at one node to transfer files to or from a remote node. The SYNTAX is a subset of DECnet and an example of transferring a file to a local

node follows:

```
NFC =NODE::[UIC]FILENAME.EXT
```

A file can be transferred to a remote node by entering the following:

```
NFC NODE::=[UIC]FILENAME.EXT
```

Remote file access is a by-product of the operator file-transfer system's service referred to earlier. It allows a program to read, write, or create files in a remote node. Remote access can be by sequential-, direct-, or block-mode method of access. Again, when a remote-file operation is requested, a fresh copy of the remote-file access code is spawned so that multiple remote-file operations can occur simultaneously.

A line-watcher task (LWT) was provided to support the many miscellaneous network functions:

- LWT processes remote-connect requests to the remote-file access and the down-line loader tasks. This service provides for simultaneous operation of multiple copies of these programs.
- Stale packets are flushed to LWT by the ETHERNET driver. When LWT receives a stale packet, the header is printed on the console log.
- Get and set local system time are implemented to allow a task in any node to read and/or set the system time in any remote node. This technique is used by all RSX11S nodes to synchronize their system time to the host node. It is also useful to synchronize all nodes to a master clock.
- Echoing packets for testing is used for determining the condition of remote nodes. This is a simple method of testing a remote CPU and communication hardware during normal operation.
- LWT sends and accepts ID packets and maintains the status of any nodes that are active in the network. This information is available to any tasks in the node through the RSX send/receive directive.
- LWT can also "get" or "get and clear" local error statistics, then sends the results to a remote node. This capability allows a task to read and/or clear a selected set of error counters in remote nodes if desired during testing operations.

FALCON SYSTEM

The Falcon system was developed to satisfy a need for a simple, efficient computer system to be used at the data-acquisition and control level, in a distributed network environment. Most tasks required at the lower level of such a system are not very complex and therefore do not require a sophisticated operating system; in fact, more emphasis should usually be placed on performance. A system of this type needs to be able to handle a wide range of CAMAC I/O modules that are most easily handled by using a set of hardware modules (HM) (software subroutines) for the more complex types. Many modules (if well designed) can be accessed by a single command such as reading an ADC, easily handled by including the necessary command in the channel database record. More complex modules can sometimes be accessed through a string of commands

in the database that defines a particular module. This same approach can be applied to equipment modules that are a collection of HMs connected to the remote equipment. The objective here is to develop a general system that is database driven, requiring few if any changes as the project develops.

A major program, developed for this system, is a surveillance program. It is responsible for scanning its database list(s) of channels, checking for out-of-limit conditions, and (if found) reporting them to a designated host. The channels are all accessed using only the information in the database and a set of macros developed for accessing CAMAC.

Two programs required (not currently developed) are a remote request handler and a local process module. Both programs are driven by command strings, the first from a received packet and the latter from strings in its node database. Primarily, these programs include calls to system functions such as read or write to a channel, wait for an operation to complete, etc.

A set of HMs is required for a few, more-complex module types; however, all CAMAC accesses are made through the CAMAC macros, which makes the system independent of a particular manufacturer's CAMAC controller.

The Falcon system is a memory-resident (can be EPROM based) multitasking system with a linked-list, active task list (ATL). It supports the executive and data-buffer pools, which are simply two sections of pool, each composed of different but fixed-length blocks. The executive pool is used by the system for building executive lists such as the ATL and the delayed task list (DTL), but is also available to programs for temporary use. The data-buffer pool is used by the XQ driver for storing ETHERNET messages, and their length is as specified for the ETHERNET driver (64 bytes at present). The data-buffer pool may also be used by programs for building up reply messages, etc.

The Falcon-system services allow for requesting a program, requesting a program and passing arguments to it, or delaying a program. When a packet is sent to a program, it is requested by the ETHERNET driver along with its arguments, that is, the received packet. The program can request a packet transfer to its own buffer or, if the packet fits in one data buffer, it can be used as is without the need for transfer because the driver merely gets another buffer from pool. The delay function saves R0-R5, along with the program restart address; at the end of the specified time interval, the DTL block is inserted into the ATL. The task is now ready to run if another task is not running; in any event, when the task runs, its registers (R0-R5) will be restored first. This allows a task to be re-entrant; therefore, many copies of the same task may be running at the same time. This capability is particularly useful if a task needs to execute for awhile and then wait for some event to occur.

The ETHERNET driver is functionally equivalent to the RSX driver; however, because all of the memory is addressable by all tasks and the system, some simplifications are possible. One is that there is no need for a task to post a receive; the driver simply makes an entry in the ATL along with the packet address for each received packet, which will

result in the task running when it reaches the head of the ATL. Also, this mode of operation allows more than one request to be outstanding for the same task. Another simplification allows the task to use the buffer where it arrives, instead of transferring it to a local buffer.

These services, along with numerous system subroutines, make it fairly easy to create applications on the Falcon.

SYSTEM PERFORMANCE

When the Falcon system was first brought on-line, it became apparent that it had a "real time" feel and would allow an operator to control the accelerator easily. Some measurements were made, and they indicated that the system still provided a knob update rate of 10 updates per second at full load. This rate is certainly fast enough to do the job required by the system.

Performance is always difficult to measure and can be somewhat subjective. However, some measurements were made with the network loaded and some with it unloaded; the results are shown in Table 3 and Figures 1-5.

TABLE 3
UNLOADED NETWORK RESPONSE FOR 64-BYTE PACKETS

CPU Type Master/Slave	System Type Master/Slave	Loop Time (ms)
11/73 ^a - 11/73	RSX/RSX	6.8
11/60 - 11/73	RSX/RSX	7.6
11/73 ^a - 11/23	RSX/RSX	8.1
11/60 - 11/23	RSX/RSX	8.9
11/23 - 11/23	RSX/RSX	10.8
11/73 ^a - 11/21	RSX/Falcon	5.3
11/60 - 11/21	RSX/Falcon	6.1
11/23 - 11/21	RSX/Falcon	8.8

^aKDJ11-BC CPU Module

First, the unloaded network response will be discussed. To make this measurement, all network traffic was removed and all nonessential tasks were aborted. The remaining network-overhead traffic had a minor (unmeasurable) effect on response; however, the individual tasks initiating this traffic added some CPU loading for the CPUs involved. These measurements were made with a variety of CPUs and two operating systems: the RSX system and the Falcon stand-alone system. All measurements were made using a master task (packet originator) and a slave task (ECHOer). The master sent a packet with an embedded sequence number, the slave received it and ECHOed it back to the master. The master checked the packet sequence number and sent another packet; at the end of this process, the average loop time per packet was recorded. All measurements were for 1000 packets, each 64 bytes in length. The results from the various tests are shown in Table 3.

Figure 1 is a plot of three different tests showing throughput versus packet size in bytes. All tests used the master/slave relationship referred to earlier. Curves 1 (11/73s) and 2 (11/23s) illustrate

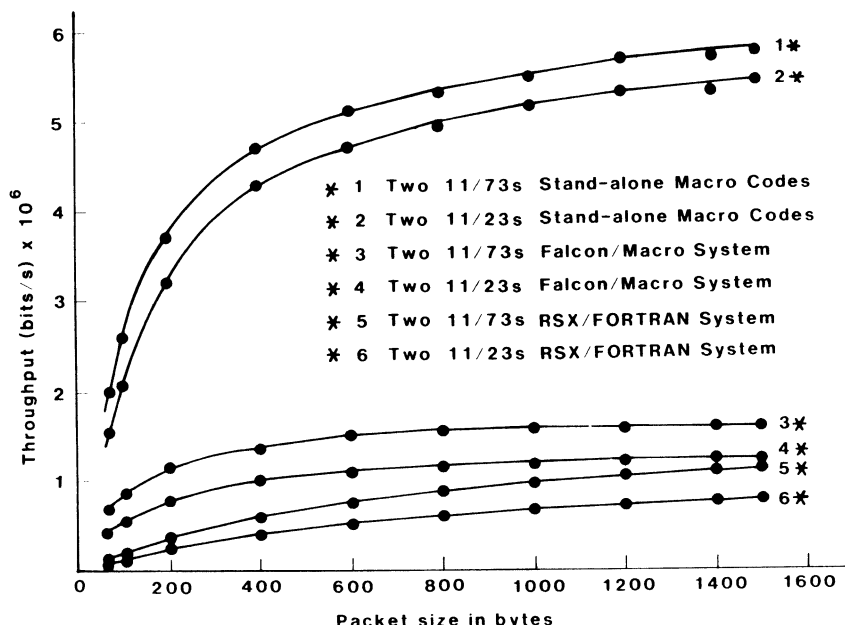


Figure 1. Ethernet performance versus packet size.

the maximum throughput for two individual nodes using a stand-alone macro program in each. It should be noted that this test is of academic interest only; it is unlikely that any operating system could support this rate. This is really an indication of the system hardware limit.

Curves 3 (11/73s) and 4 (11/23s) were derived using the CPUs indicated on the Falcon system; even this low-overhead system significantly reduces throughput. Curves 5 (11/73s) and 6 (11/23s) use two FORTRAN codes on the RSX operating system in the master/slave relationship indicated above.

For all tests, it is obvious that (for increased packet size) the throughput goes up accordingly because there is some amount of fixed overhead in both the hardware (controller) and software. When this overhead is amortized over longer packets, it has a decreasing effect on the throughput. Also, as the operating system complexity increases, the throughput decreases.

Unfortunately, most control systems must operate with short packets, which is the least efficient area on any of the curves. Furthermore, control systems most frequently require a sophisticated operating system to be able to handle the changing demands of the system being controlled. Although the throughput appears small (compared to 10 megabits/s), it is not that small when considering the volume of data that can be transferred, even at these reduced rates. For an 11/73 pair for instance, the rate for a 64-byte packet is 140 kilobits/s; for an 11/23, it is 95 kilobits/s.

The tests⁵ results shown in Figures 2-4 were run with all conditions held constant except the number of CPUs contending for network access. These tests were run on a fiber-optic (FO) network consisting of a four-port passive star and a FO transceiver at the end of each of the four FO cables. The tests covered by Figure 5 had one

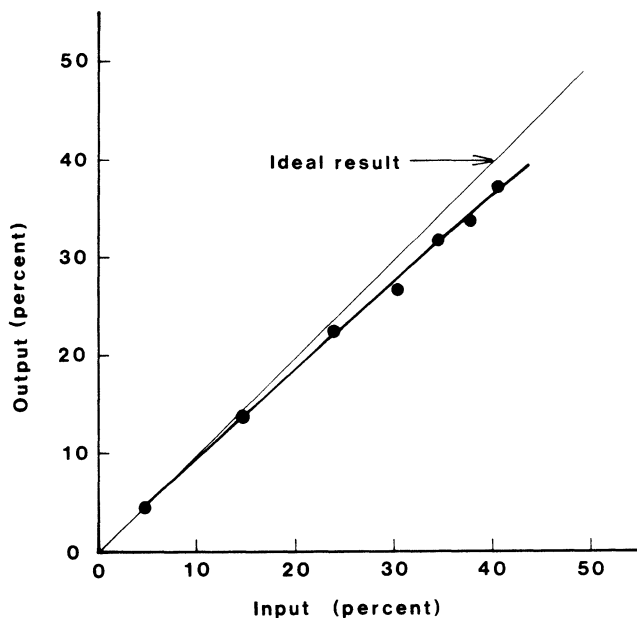


Figure 2. Fiber-optic network performance for 10 nodes in percent of bandwidth.

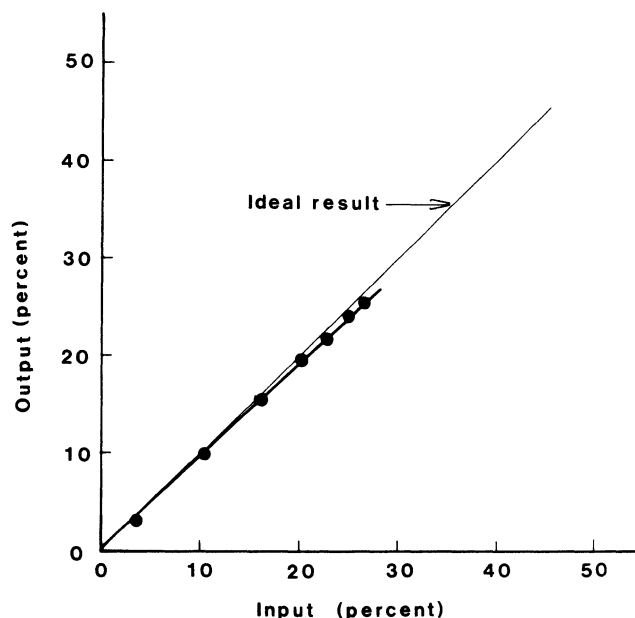


Figure 4. Fiber-optic network performance for six nodes in percent of bandwidth.

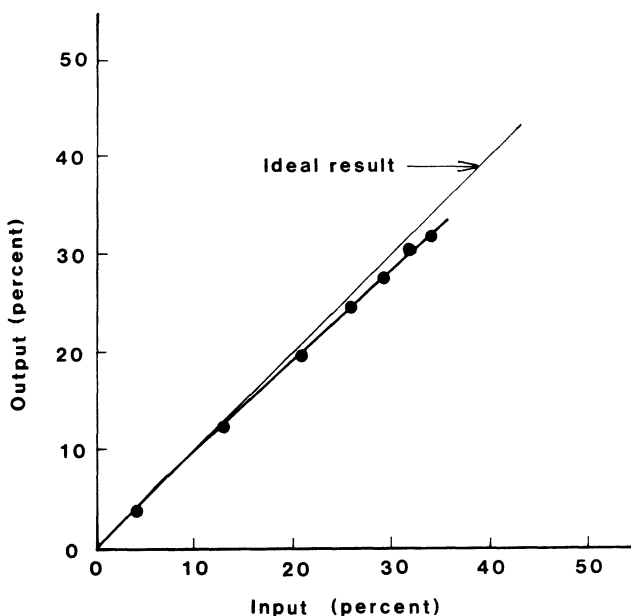


Figure 3. Fiber-optic network performance for eight nodes in percent of bandwidth.

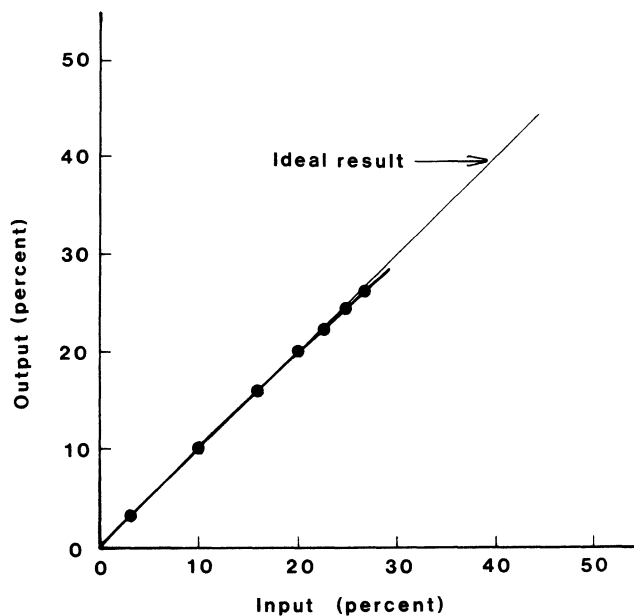


Figure 5. Nonfiber-optic network performance for six nodes in percent of bandwidth.

additional difference: they did not use the FO network but instead used a local DELNI. This technique was used to determine what, if any, effect the FO system had on network throughput. Tests were run with 10, 8, and 6 CPUs vying for network access. In all cases a master/slave relationship existed; that is, for the 10-CPU case, there were 5 masters and 5 slaves. As described in earlier tests, the master originated the packet, sent it to its slave (ECHOer), and checked it for validity upon receipt from the slave. If a packet did not return within a specified time (10 clock ticks or 167 ms), it was marked lost and retransmitted. During the course of each load test, the number of

CPUs was held constant, with only the packet size being varied to increase the load. All CPUs for a given test were started almost simultaneously (± 25 ms) by a test manager program and were run for 10 s. At the end of the time interval, the CPUs reported their results back to this test manager program when polled. This information reported back was the number of packets sent and received, as well as the number of packets lost. At the conclusion of a test series, the unloaded input was measured by running each master/slave pair, without network loading, for each packet size used during the load test. The packet sizes used during all of these tests were 64, 250, 500,

750, 1000, 1250, and 1500 bytes for each test series.

The results shown in Figure 2 were for the case where 10 CPUs (5 pairs) were vying for network access. During the 70-s test (7 runs of 10 s each), 36.4K packets were sent and 140 were lost. The total input (unloaded) was 40.4% of the ETHERNET bandwidth; whereas the output was 37.4%, thus showing a loss of 3% or 300K bits/s. There was a total of 1459 collisions for this test series, not one was fatal. The master/slave node pairs for this test were as follows (refer to Table 1 for the CPU type and system):

<u>Master Node</u>	<u>Slave Node</u>
11	7
10	6
9	5
8	4
2	3

The results shown in Figure 3 depict the case where eight CPUs (four pairs) are contending for network access. During this test series (70 s), a total of 32.4K packets were sent and 44 packets were lost. The maximum input was 33.9% for an output of 31.9%, which shows a net loss of 2%, or 200K bits/s. The master/slave node pairs for this test were as follows:

<u>Master Node</u>	<u>Slave Node</u>
11	7
10	8
9	6
2	3

The results shown in Figure 4 were for the case where six CPUs (three pairs) were vying for network access. During this test series (70 s) a total of 25.9K packets were sent and 26 were lost. The maximum input was 26.5%, corresponding to an output of 25.4% for a loss of 1.1%, or 110K bits/s. The master/slave node pairs were as follows:

<u>Master Node</u>	<u>Slave Node</u>
11	5
10	4
2	3

The results shown in Figure 5 were also for the case of six CPUs (three pairs) contending for network access. The major difference was that the FO system was not used; instead, the CPUs were all on a single DELNI. During this test series (70 s), a total of 26.0K packets were sent and no packets were lost. The maximum input was 26.4%, corresponding to an output of 26.1% for a net loss of 0.3%, or 30K bits/s. Because no packets were lost, the drop in throughput was due to collisions. The physical location of the CPUs in the network made it necessary for the slave nodes to be different from those in the previous test. They were as follows:

<u>Master Node</u>	<u>Slave Node</u>
11	9
10	8
2	7

It would appear that the major drop in throughput was due to packet loss in the FO system; however, this is not totally conclusive. A better test would be to configure the network using standard ETHERNET coaxial cable and H4000 transceivers and repeat the test shown in Figure 2. This type of test could not be done because of lack of standard ETHERNET equipment. It also appears that the packet loss increases with increasing load (see Figures 2-4), which is what one would expect.

CONCLUSIONS

This system has met or exceeded our requirements for speed of response and system flexibility (expandability). Our control-knob update rate of 10 per second under full load is certainly fast enough to give a real time feel for accelerator operation. It would appear that future systems would use 11/73s and Falcons, where possible, to improve system response. Also the FO system should be improved or replaced with a standard ETHERNET coaxial cable system. With an ETHERNET system, nodes can be added easily as requirements arise; this capability facilitates adding new functions or splitting off functions from an existing overloaded system. The ETHERNET system should be easily upgraded to any new bus or ring network architecture if one should prove to be superior in the future.

ACKNOWLEDGMENTS

I would like to acknowledge the efforts of the entire Instrumentation and Control Section of AT-4 (the Accelerator Technology Division Systems Integration Group at Los Alamos) for their contributions toward the design and implementation of the original FMIT prototype control system. Also, I would like to give special thanks to Jim Johnson for his encouraging words and continued support during the process of developing the ETHERNET system.

REFERENCES

1. R. M. Suyama, D. R. Machen, and J. A. Johnson, "The FMIT Facility Control System," IEEE Trans. Nucl. Sci. 28 (3), 2252 (1981).
2. E. P. Elkins, "Use of ETHERNET for the FMIT Control System," Los Alamos National Laboratory memorandum AT-4-82:368 (December 27, 1982).
3. "IEEE Standard Modular Instrumentation and Digital Interface System (CAMAC)," ANSI 583-1975 (American National Standards Institute, Inc., New York, 1975), IEEE Std. 583-1975, published by The Institute of Electrical and Electronics Engineers, Inc., 345 East 47 Street, New York, NY 10017.
4. "The ETHERNET, A Local Area Network Data Link Layer and Physical Layer Specifications," DEC order number AA-K759A-TK, Version 1.0 (compiled by Digital Equipment Corporation, Maynard, Massachusetts; Intel Corporation, Santa Clara, California; and Xerox Corporation, Stanford, Connecticut. Version 1.0, September 30, 1980).
5. J. F. Shoch and J. A. Hupp, "Measured Performance of an ETHERNET Local Network," Communications of the Association for Computing Machinery 23 (12), 711 (1980).

UPGRADING A USER-WRITTEN I/O DRIVER
FROM RSX-11M TO RSX-11M-PLUS

Al Tyrrill
Digital Consulting
Garden Grove, California

ABSTRACT

This paper describes the modifications that must be made to an I/O driver written for RSX-11M to port it to an RSX-11M-PLUS system. It presents an overview of the I/O data structures for both systems and describes the entry points that must be implemented and the macros that are available. It explains how to modify the RSX-11M driver, gives some driver hints that apply to I/O drivers for either system and discusses some of the advanced features available for drivers in RSX-11M-PLUS. It concludes by explaining how to incorporate the modified driver into an RSX-11M-PLUS system.

INTRODUCTION

Most application programs can be ported from RSX-11M to RSX-11M-PLUS without modification. In fact, for nonprivileged programs, the .TSK file from the 11M system can usually be copied to the 11M-PLUS system and executed. Privileged programs must be recompiled/assembled and retaskbuilt, but can sometimes be executed without further change.

If this is attempted with an I/O driver, however, the 11M-PLUS system will crash, usually when the driver is loaded or the associated controller or device is configured online.

This paper describes the changes that must be made in an RSX-11M I/O driver to make it run under RSX-11M-PLUS. If it is necessary that a driver run under either operating system, the changes must be conditionalized. The conditional assembly symbol R\$\$MPL is defined in RSX-11M-PLUS but not in 11M, and can be used for this purpose.

The remainder of this paper will cover the following.

An overview of the I/O data structures for RSX-11M and RSX-11M-PLUS.

Required I/O driver entry points and available macros for drivers.

Modifying the RSX-11M driver.

Driver hints, applicable to either operating system.

Advanced features for the support of I/O drivers in RSX-11M-PLUS.

Incorporating the modified driver in an RSX-11M-PLUS system.

RSX-11M I/O DATA STRUCTURES

The database for an RSX-11M I/O driver consists of these structures.

Device Control Block (DCB)

The DCB contains the static characteristics of a device type. It contains information such as the device name, range of unit numbers and valid function masks. There is one DCB for each device type and they are linked in a list headed by executive symbol \$DEVHDD.

Status Control Block (SCB)

The SCB contains the state of a device controller. It contains the CSR and interrupt vector addresses, device priority, space for a fork block and the I/O queue listhead. There is an SCB for each controller.

Unit Control Block (UCB)

The UCB contains the state of a particular device unit. There is one UCB for each unit in the system. It contains, for example, unit status, device characteristics and buffer parameters.

The I/O packet, not part of the driver's database, is the interface between the application task and the driver. The executive constructs the packet on behalf of the task and queues it to the driver. It describes the specifics of the I/O operation.

Figure 1, below, illustrates the relationship of the I/O data structures. The DCB has a pointer to the UCB for the first device (usually unit 0) and also contains the length of the UCB. Since all the UCBs must be the same size, the locations of all the UCBs can be calculated.

Each UCB contains a pointer to the SCB for the controller associated with that device. There is also a back-pointer to the DCB. Figure 1 illustrates the case of two controllers, each with two units.

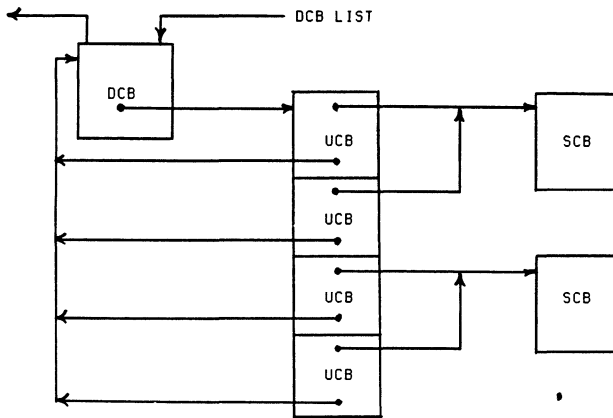


Figure 1) RSX-11M I/O Data Structures

RSX-11M-PLUS I/O DATA STRUCTURES

In RSX-11M-PLUS, the DCB, SCB, UCB and I/O packet exist as in RSX-11M, but with not exactly the same structure or interpretations. Two new data structures exist in 11M-PLUS, the Controller Table (CTB) and the Controller Request Block (KRB).

Certain special situations also require additional structures. A Common Interrupt Table is provided when different device types can be attached to a single controller, as is the case with the RH11/RH70. A Subcontroller Request Block (KRB1) is required when a subcontroller can be connected to a controller, as with the TM02/03 tape formatter. These situations are discussed in more detail in the section on advanced I/O driver features in RSX-11M-PLUS.

Controller Table (CTB)

The CTB defines a unique controller type, there is one CTB for each controller type in the system. They are linked in a list like the DCBs, the CTB list is headed by executive symbol \$CTLST. The CTB contains the following.

Two-letter ASCII controller code. This must be unique across the system, but can be the same as the device code in the DCB.

Link to first Interrupt Control Block (ICB). The ICB is the code block, built within the executive when the driver is loaded, that first receives control when an interrupt for the driver occurs.

Pointer to the associated DCB. This pointer references the Common Interrupt Table when it is present (in that case there will be multiple DCBs).

List of KRB addresses.

The format of the CTB is illustrated in figure 4-15 of reference 1.

Controller Request Block (KRB)

This structure defines the status of a particular device controller. There is one KRB for each controller, and it contains the following information, most of which is in the SCB in RSX-11M.

CSR and interrupt vector addresses, device priority, controller index, status.

I/O count. This is a measure of activity on the controller, used for load balancing with dual-controller devices.

List of UCB addresses. The address of the UCB for each device to which the controller is connected appears in this list.

Pointer to the owning UCB. This is the UCB of the device for which the controller is currently performing I/O.

Controller request queue. This queue is used to queue fork blocks of processes waiting to use the controller (distinct from the queues of I/O packets in the SCBs).

The format of the KRB is illustrated in figure 4-12 of reference 1.

Status Control Block in RSX-11M-PLUS

The SCB has a somewhat different format and interpretation in RSX-11M-PLUS. The CSR and vector addresses, priority and controller index fields have been moved to the KRB. The SCB is now the context for an I/O operation on a unit by a controller.

There will be an SCB for each parallel operation by a controller that can be occurring, for example, parallel seeks on disks. In the case where a device is connected to the system through more than one controller (e.g. multiport disk) the SCB contains a list of addresses of the corresponding KRBs. Figure 4-9 in reference 1 specifies the format of the SCB in RSX-11M-PLUS.

SCB - KRB Interrelationships

The relationship of the KRBs and SCBs will differ, depending on which of three possible situations apply.

1. When the controller operates serially, as with, for example, a printer, the KRB and SCB are contiguous (parts of the same control block). See figure 4-14 in reference 1.
2. When the controller operates in parallel, as with overlapped disk seeks or mag tape positioning commands, there is an SCB for each unit. The KRB has a list of UCB addresses and a pointer to the UCB for which a data transfer operation is currently occurring. The UCB has a (static) pointer to the corresponding SCB.
3. When the device is connected to the system through several controllers, the SCB has a list of the addresses of the associated KRBs, and a pointer to the KRB of the currently active controller.

The RSX-11M-PLUS I/O data structure relationships are illustrated by figures 2-1 through 2-4 of reference 1.

Other RSX-11M-PLUS Changes

The DCB is unchanged from 11M to 11M-PLUS. The UCB has certain device dependent changes for accounting and error logging.

The I/O packet has two attachment descriptor block pointers added (used by the executive for making a region containing an I/O buffer uncheckpointable) and this changes the packet's length. Drivers that allocate blocks of pool the same size as I/O packets in order to take advantage of the fast allocation for packets of this size will need to be modified accordingly.

RSX-11M-PLUS Executive Memory Mapping

In RSX-11M, kernel mapping registers 0 thru 4 are used to map the "resident" executive, which includes some of the executive code, the resident I/O data base and the dynamic storage region, "pool". These cover the low 20K words of physical memory.

Registers 5 and 6 are used to map I/O drivers, system routines in the executive commons and privileged tasks. Register 7 maps the I/O page.

Processors that have separate mapping registers for instruction and data space have the I-space and D-space registers overmapped, as RSX-11M does not support separate I and D space.

In RSX-11M-PLUS, on processors that support separate I and D space, both I and D kernel registers 0 map the first 4K of physical memory. Kernel I-space registers 1 thru 4 map the executive code in the next 16K of memory.

D-space registers 1 thru 4 map the resident I/O data base and pool, which is located in physical memory right above the executive code. The virtual addresses of I/O data structures in the resident data base or pool now no longer are the same as the physical addresses. Beware when using the MCR OPEN command!

I and D registers 5 and 6 map the drivers, executive commons and privileged tasks just as they do in RSX-11M, and I and D are overmapped. Thus 11M-PLUS drivers can contain embedded data, just as 11M drivers do. Both I and D registers 7 map the I/O page.

COMMON I/O DRIVER ENTRY POINTS

The following entry points must be supported by drivers in both RSX-11M and RSX-11M-PLUS. xx is the 2-character device name in the DCB.

xxINI: I/O initiation
\$xxINT: interrupt service routine
(one or more)
xxCAN: cancel I/O

xxOUT: device timeout
xxPWF: power failure
carry set - controller
carry clear - unit

In RSX-11M-PLUS, the following additional entry points can be provided by a driver.

xxKRB: controller status change
xxUCB: unit status change
xxCHK: block number check and conversion
(for seek optimization)
xxDEA: deallocation
(buffered I/O)
\$xxLOA: entry from MCR LOAD command
\$xxUNL: entry from MCR UNLOAD command

Controller/Unit Status Change

If the controller status is changed with CON ONLINE/OFFLINE, entry is made to the driver at xxKRB:. Likewise, if the unit status is changed with CON, entry is made at xxUCB:. The direction of change is indicated with the carry bit, as follows.

carry set - online to offline
carry clear - offline to online

The driver indicates success or failure to the executive by storing a value in location \$SCERR in kernel data space, as follows.

<0 - rejection
=1 - success

The driver can return immediately to the executive via the address on the top of the processor stack, or can make a delayed return. The latter is to accommodate devices like disks, which may take a while to become ready (due to spinning up).

In the delayed return, the driver saves the top address on the stack, then returns immediately to the next stack address. It then returns to the top address sometime in the next 60 seconds. The driver must arrange for its own activation in this interval, e.g. with a device interrupt or timer expiration.

Block Number Check and Conversion

Certain disk drivers support ordering of disk accesses so that seeks occur more optimally than the random order of arrival. This is described in more detail in the paragraph on advanced RSX-11M-PLUS features, below. The code at entry xxCHK: must determine if the operation is a data transfer, and if so, convert the disk logical block number to cylinder, track and sector for the optimization algorithm.

Buffer Deallocation

A driver that transfers data through an intermediate buffer is entered at xxDEA: when the data transfer is complete. The code at this point must deallocate

the intermediate buffer. This is also described in more detail in the paragraph on advanced RSX-11M-PLUS features.

MCR LOAD/UNLOAD Entry Points

If it is necessary that a driver be entered when it is loaded or unloaded, the entry points \$xxLOA: and/or \$xxUNL: can be implemented. The presence of these global symbols in the driver's .STB file causes LOAD or UNLOAD to call the driver after loading or before unloading. Drivers with these entry points cannot be loaded with VMR, because VMR modifies a system image on disk, not a running system. VMR gives an error message if this is attempted.

DRIVER DISPATCH TABLE

In RSX-11M, the driver dispatch table is a simple 4-word structure, containing the addresses of the initiator, I/O cancel, device timeout and power failure entry points.

In RSX-11M-PLUS, there are two parts to the driver dispatch table. The first part is a 6 to 8 word entry point address list, as shown.

	Optimization Entry Point	xxCHK:
	Deallocation Entry Point	xxDEA:
\$xxTBL:	I/O Initiation Entry Point	xxINI:
	Cancel I/O Entry Point	xxCAN:
	Device Timeout Entry Point	xxOUT:
	Powerfail Entry Point	xxPWF:
	Controller Status Change EP	xxKRB:
	Unit Status Change EP	xxUCB:

The fields for the optimization and deallocation entry point addresses have negative offsets from the head of the table and need not be present if the feature is not used.

The second part is a list of interrupt entry point blocks. In RSX-11M-PLUS, an I/O driver can support multiple controller types, and there must be an interrupt entry point block for each controller type. The full duplex terminal driver TTDRV (in the RSX-11M-PLUS distribution kit) services multiple controller types.

Each interrupt entry point block contains the generic 2-character controller name from the CTB, a list of interrupt entry point addresses (terminated by a zero word) and a pointer to the KRB list in the CTB.

The interrupt entry point address list also contains offsets so that the executive can calculate the locations of all the interrupt vectors, using the address of the first vector contained in the KRB. Even numbers in the address list are assumed to be interrupt entry points and odd numbers are offset codes.

This mechanism is described in detail in paragraph 4.5.1 of reference 1. The driver dispatch table format is illustrated in figure 4-18 of that reference.

RSX-11M-PLUS DRIVER MACROS

Three macros are provided to simplify coding of certain tables and interfaces.

DDT\$ - Generate Driver Dispatch Table (DDT)

GTPKT\$ - Get Next I/O Packet

INTSV\$ - Interrupt Save
(modified from RSX-11M)

Generate Driver Dispatch Table

The DDT\$ macro generates a driver dispatch table for an RSX-11M-PLUS driver. It is limited to a single interrupt entry point block and only handles the case where the interrupt vectors are contiguous. The format of DDT\$ is as follows.

DDT\$ dev, nctl, ints, iny, tbl, new, opt, buf

The parameters have the following interpretation.

dev - xx, the 2-character device name from the DCB.

nctl - number of controllers, usually a symbol of the form x\$\$x11.

ints - interrupt entry points. NONE means there are none, leaving this parameter blank means there is one, named \$xxINT. Otherwise, a list of 3-character suffixes enclosed in braces is coded. For example, if there are two interrupt entry points, named \$xxINP: and \$xxOUT:, this parameter is coded <INP,OUT>.

iny - the 3-character suffix of the initiator entry point, if it is named something other than xxINI:.

tbl - the name of a UCB address save table to be allocated. This is the CNTBL from RSX-11M.

new - if anything is coded here, the entry points xxKRB: and xxUCB: are created, and the power fail entry point is called on both controller and unit power failures. Otherwise, code is generated that calls the driver's powerfail entry point only on unit power failure, and on a unit status change from offline to online if the device's UC.PWF (unconditional call at power fail) bit is set.

opt - if anything is coded here, the xxCHK: entry point address is built into the DDT.

buf - if anything is coded here, the xxDEA: (deallocation) entry point address is built into the DDT.

Get Next I/O Packet

The GTPKT macro implements the call to the \$GTPKT executive routine. In RSX-11M, this is done directly by the user. GTPKT\$ has the following format.

GTPKT\$ dev, nctl, nopk, tbl, suc

The parameters are defined as follows.

dev - xx, the 2-character device name from the DCB.

nctl - the number of controllers, usually a symbol of the form x\$\$x11.

nopk - the address to which a jump is taken if no packet is available or the device is busy. If left blank, a RETURN is generated.

tbl - the name of the UCB address save table coded in the GTPKT\$ macro. Usually this will be CNTBL. The macro generates code to save the UCB address in this table upon returning from the \$GTPKT routine.

suc - anything coded here indicates the controllers serviced by this driver control a single unit.

Interrupt Save

This macro is coded at a driver's interrupt entry points. In resident drivers it extracts the controller index from the PSW, saves R4 and R5, and calls the \$INTSV routine. In loadable drivers it just fetches the UCB address from the save table or the KRB. INTSV\$ in RSX-11M-PLUS has a different format of that in RSX-11M.

INTSV\$ dev, pri, nctl, psws, tbl

The parameters have the following interpretations.

dev - xx, the 2-character device name from the DCB.

pri - device priority, PR4 - PR7.

nctl - the number of controllers, usually a symbol of the form x\$\$x11.

psws - cell for saving the PSW while initially saving R4 and R5. This is only necessary in resident drivers and is provided for compatibility with RSX-11M. It is not necessary at all in RSX-11M-PLUS, now that register saves without modifying the PSW are done with JSR.

tbl - the name of the UCB address save table coded in the GTPKT\$ macro. Usually this will be CNTBL. The macro generates code to fetch the UCB address from this table.

The INTSE\$ macro, with the same parameters as INTSV\$, is used for error logging devices. DEC disk seek overlap drivers do not use INTSV\$/INTSE\$. They use a different interface which is described in the paragraph on advanced features, below.

MODIFYING A DRIVER'S DATABASE

The following describes the modifications necessary to the database of a "conventional" RSX-11M driver, i.e. one whose controllers operate serially and do not use special features such as queue optimization, in order for it to be used with RSX-11M-PLUS. If the driver is to be used on both operating systems, the changes will have to be conditionalized on symbol R\$\$MPL.

Create the Controller Table (CTB) - allocate space for the ICB link, next CTB link and generic controller name. It can be the same as the device name in the DCB, but not as any other controller name. Use the CTB macro to establish the link to the next CTB in a resident database (CTB is a no-op in a loadable database). Allocate space for the DCB pointer, the status byte and number of KRB addresses.

Allocate a word for each KRB address. Figure 2 below is an example of the CTB for a loadable driver.

```

;
; CONTROLLER TABLE
;
      .WORD 0           ; L.ICB, LINK TO 1ST ICB
      CTR              ; MACRO THAT GENS LABEL
      .WORD 0           ; LINK TO NEXT CTB
      .ASCII /UR/      ; L.NAM, GENERIC CONTROLLER NA
      .WORD .URDCB     ; L.DCB, DCB ADDRESS
      .BYTE 1          ; L.NUM, NUMBER KRB ADDRESSES
      .BYTE 0          ; L.STS, CONTROLLER STATUS
SURCTB::
      .WORD SURK0      ; L.KRB, KRB ADDRESS
;
;SUREND::           ; END OF UR DATABASE
;
      .END

```

Figure 2) Controller Table

Extract Controller Request Block (KRB) - most of the fields of the KRB will come from the old SCB. Allocate the KRB and new SCB contiguously, with the SCB following. The KRB address fields in other data structures should reference the CSR word (K.CSR/S.CSR). The SCB address fields in other structures should reference the I/O queue listhead (S.LHD/K.CRQ). Figure 3 below is an example of a combined KRB/SCB.

```

;
; KRB AND STATUS CONTROL BLOCK
;
      .BYTE PR5        ; S.PRI, DEVICE PRIORITY
      .BYTE URO$VC/4  ; S.VCT, VECTOR ADDR/4
      .BYTE 0,0       ; S.CON, K.CON, K.IOC, CONTROLLER
                        ; INDEX, I/O COUNT
SURK0::
      .WORD KS.OFL    ; K.STS, CONTROLLER STATUS
                        ; KRB LABEL
      .WORD URO$AD    ; S.CSR, RECEIVE CSR ADDR (XMIT IS
      .WORD 0         ; K.OFF, OFFSET TO UCB TABLE
      .WORD 0         ; K.HPU, HIGHEST PHYSICAL UNIT
      .WORD .URO      ; K.OAN, OWNER UCB
SURSO::
      .WORD 0,-2      ; URO STATUS CONTROL BLOCK
      .WORD 0,0,0,0  ; S.LHD, I/O PACKET QUEUE LISTHEAD
                        ; S.FRK, $FORK AREA
      .WORD 0         ; EXTRA WORD FOR S.FRK
      .WORD 0         ; S.PKT, I/O PACKET ADDR
      .BYTE 0,16     ; S.CTM,S.ITM, CURR & INITL TIMEOUT
      .BYTE 0,0      ; S.STS, S.STS, CONTROLLER STATUS
      .WORD S2.CON    ; S.ST2, STATUS EXTENSION
      .WORD SURK0     ; S.KRB, KRB ADDRESS
      .WORD 0         ; EXTRA SPACE

```

Figure 3) Combined KRB/SCB

MODIFYING A DRIVER'S CODE

This paragraph describes the modifications necessary to convert the code of a "conventional" RSX-11M driver, as defined above, to RSX-11M-PLUS.

DDT\$ Macro - replace the handcoded driver dispatch table with the DDT\$ macro, as shown.

DDT\$ xx, x\$\$x11, , , CNTBL

Use the UCB save table CNTBL, and delete or conditionalize the handcoded CNTBL that most likely exists in the RSX-11M driver. Figure 4 below is the expansion of DDT\$ coded as indicated.

GTPKT\$ Macro - replace the call to \$GTPKT with the GTPKT\$ macro, coded as shown.

GTPKT\$ xx, x\$\$x11, , CNTBL, suc

suc = 1 if controller has a single unit
null otherwise

Figure 5 illustrates the expansion of GTPKT\$ coded as indicated.

of which is run by a different driver. For example, the RH70/RH11 controllers supports DB (RP04/05/06) and DR (RM02/03/05/80) disks, the EM "semiconductor disk" and MM magtape devices.

The driver interrupt interface is different from normal, in that interrupts are handled by executive routine \$RHALT, rather than code in an interrupt control block.

The database differs from normal in that there is one Controll~ Table for all the devices. The DCB pointer in the CTB points to a new structure called the Common Interrupt Table. It contains pointers to various entry points in the executive interrupt handler (\$RHALT for the RH70/RH11) and a DCB address list. There is still one DCB for each device type. There is one KRB for each RH controller, and one SCB for each unit.

When loading drivers for the devices on such a controller, the first is loaded normally, but subsequent drivers are loaded using the /CTB switch on the LOAD command. This tells LOAD that the CTB is already loaded. The parameters on /CTB tell LOAD which controllers the device is attached to.

If a device on the controller is in fact a subcontroller, as is the case with the TMO3 tape formatter on the RH70, a Subcontroller Request Block (KRB1) is required for each subcontroller. The KRB1 is a subset of the KRB. Each unit to which the subcontroller is connected points to it and the KRB1 has a pointer to its "master" unit.

Overlapped Disk Seeks

RSX-11M-PLUS supports overlapped operations on devices connected to controllers that are capable of it. For example, disk seeks and some tape positioning commands can be overlapped.

To support overlapped operations of this sort, there must be an SCB for each unit, which cannot be combined with a KRB. At any given time, the KRB points to the active unit for data transfer.

There must be an interrupt handler in the executive that distinguishes data transfers from control operations. For data transfers, the executive interrupt handler drops to fork level and calls the driver at its interrupt entry point. For control operations, the driver is entered at interrupt entry point + 2, at device priority. The drivers do not use INTSV\$/INTSE\$ at their interrupt entry points.

For RH controllers the executive interrupt handler is routine \$RHALT. For RK06/07 disks, it is executive routine \$MHALT. A user wishing to add disks with controllers that do not look to the system like either of these will need to implement an interrupt handler modelled after \$RHALT/\$MHALT.

Dual-Access Disks

This feature supports devices that are connected to the system by more than one controller, where the executive selects the controller to be used for a particular operation. It is not supported by RSX-11M, and supported by 11M-PLUS only when both controllers are on the same system.

The problem with connecting two systems with a dual-access disk comes when both systems have parts of the index file and the block allocation bitmap memory resident, and each system is unaware of changes made by the other.

With this feature, SCBs are dynamically assigned to KRBS for the duration of an operation. There are three executive routines which a driver will use to implement the dynamic assignment, as follows.

- \$RQCND - Request Controller
- \$RLCN - Release Controller
- ASKRB - Assign KRB For Dual-Access Device

The executive attempts to balance the load between controllers. A measure of I/O activity is maintained in each KRB for this purpose.

Full-Duplex I/O

No special executive support is provided in RSX-11M for full-duplex I/O. This is sometimes implemented with two drivers, one for receiving data, the other for transmission.

In RSX-11M-PLUS, the driver's initiator routine calls executive routine \$GSPKT rather than \$GTPKT. It passes the address of an acceptance routine to \$GSPKT in R2. \$GSPKT dequeues a packet and calls the acceptance routine.

If the acceptance routine rejects the packet. \$GSPKT dequeues the next packet and calls again. This continues until the driver accepts a packet or all are rejected. Note that \$GTPKT in the executive is just \$GSPKT with an acceptance routine that always returns true.

A driver using \$GSPKT does not use the normal UCB and SCB busy flags, and in fact always looks "ready" to RSX. The driver must be sure not to call \$GSPKT if it is in fact busy in both directions.

The driver is responsible for maintaining the context of the second I/O operation in augmented data structures, for example in a UCB extension.

Buffered I/O

Buffering an I/O operation through an intermediate buffer in primary or secondary pool allows a task to be checkpointed during the I/O operation. This is useful for slow devices like terminals.

Three executive routines support implementation of buffered I/O.

\$TSTBF - checks for the legality of buffered I/O. The task must be checkpointable, stoppable and not fixed in memory.

\$INIBF - initiates buffered I/O and stops the requesting task.

\$QUEBF - handles buffered I/O completion. It schedules a kernel AST to transfer the data and unstops the task.

The driver is responsible for allocating and deallocating the intermediate buffer. The code for

the deallocation must be at entry point xxDEA: and this entry must be defined in the driver dispatch table.

The full duplex terminal driver TTDRV and the virtual terminal driver VTDRV both use buffered I/O. The code for VTDRV, in the 11M-PLUS distribution kit, is the easier to read.

I/O Queue Optimization

RSX-11M-PLUS optimizes the order of execution of disk accesses such as to minimize average seek time. Three algorithms have been implemented.

Nearest Cylinder
Elevator (triangular wave)
Cylinder Scan (sawtooth wave)

Nearest cylinder sometimes leaves an operation for data at the edge of a disk for long periods without performing it, but there is a "fairness count" to limit the severity of this effect. Elevator gives preference to data at the center of the disk, which is where the index file should be located if using this algorithm. Cylinder scan gives more equal preference to all data. Measurements have suggested that cylinder scan should be tried first.

The driver must provide code at entry point xxCHK: to determine if an operation is a data transfer, and if so, to convert logical block number to cylinder, track and sector numbers for the disk. \$GTPKT scans the controller queue and performs this optimization.

This feature is enabled and the algorithm selected with the MCR SET /OPT command.

INCORPORATING A DRIVER IN RSX-11M-PLUS

Incorporating a user-written I/O driver into the system is generally easier than with RSX-11M.

At SYSGEN

When the driver is to be incorporated at SYSGEN, the sources for the code and database need to be stored in SY:[11,10], and the device mnemonic give to SYSGEN when it asks. SYSGEN builds and executes the necessary MAC and TKB command files and inserts a LOAD command for the driver in the VMR command file.

After SYSGEN

When the driver is to be loaded into a running system, it must be assembled with RSXMC.MAC and EXEMC.MLB, and taskbuilt with RSX11M.STB and EXELIB.OLB, just as with RSX-11M. The symbol LD\$xx must be included in RSXMC.MAC or in the driver files if the driver is loadable.

The driver is loaded with an MCR command like

```
MCR LOAD xx:/PAR=GEN/HIGH
```

then the controller and device are configured online.

Generally, one would want to build loadable drivers under RSX-11M-PLUS, because they are much more flexible to debug and because of the many consistency checks in the LOAD and VMR utilities. Resident drivers should only be used when the faster interrupt response time they provide is required.

Device Reconfiguration

RSX-11M-PLUS supports changing the CSR and interrupt vector addresses of loaded drivers. The controller must be offline when this is done.

```
CON SET xxy CSR=177440  
CON SET xxy VEC=210
```

When a user-supplied driver has been included in the SYSGEN, the controllers and devices are configured online with the CON ONLINE ALL command that appears in the startup command file. When a driver is loaded via MCR into a running system, the controller and device are explicitly configured online, as illustrated.

```
CON ONLINE URA,URO:
```

REFERENCES

1. RSX-11M-PLUS Guide to Writing an I/O Driver, Order No. AA-H267B-TC, Digital Equipment Corp., 1982
2. RSX-11M-PLUS System Generation and Installation Guide, Order No. AA-H431C-TC, Digital Equipment Corp., 1983

SUPER LIGHTS:
ARENA LIGHTING CONTROLS AT THE LOUISIANA SUPERDOME

Bill Heidler, Jeff Thompson, Tom Cirello,
John Decker, Brent Naseath, Steve Pollman

Johnson Controls, Inc.
Special Systems Operation
San Diego, California

ABSTRACT

The Superdome lighting control system allows its operators to control more than 2200 stadium lights, either individually or in a pre-defined configuration ("mode") suitable for each type of stadium event. Lighting modes are constructed, archived, and executed by using a set of user-friendly tools and commands. The system consists of a distributed, three-level network, co-resident with an energy management system. The top level computer in the network is a PDP-11/24 running RSX-11M, with applications code written in FORTRAN 77.

SYSTEM OVERVIEW

In September 1983, Johnson Controls contracted with the State of Louisiana to develop and install a computerized facility management system at the Superdome sport and convention facility. The system was to include the capability to monitor and control the heating, ventilation, and air conditioning (HVAC) equipment; in addition, it was to provide the capability to control the main arena lights.

The Superdome arena lights are arranged in four concentric rings on the roof of the dome. Each ring contains banks of six to twenty light fixtures, in two or four horizontal rows. The lights illuminate the entire arena floor.

The Superdome is used for a variety of functions, including professional football, baseball, conventions, and trade shows (including the Spring 1985 DEXPO). The major requirements for the lighting control system were that it be fully automated, highly reliable, flexible (allowing lights to be turned on and off individually), and reconfigurable for new lighting effects and light fixture types.

The JC/84 lighting control system replaced a set of manual controls, which consisted of switch panels in each of four lighting control rooms. The lighting control rooms are located around the perimeter of the dome, at the very top level (just above Bob Uecker's seat). To manually change

the lighting configuration a technician had to go to each of these rooms and flip a switch for each light that needed to be turned on or off. Synchronization of lighting changes from the different rooms was nearly impossible.

The computerized system allows control to be centralized in a much more convenient location. It also permits light adjustments to be synchronized, and for a "blackout" effect to be achieved by turning out all the lights rapidly. Since the system combines both lighting and HVAC control functions it eliminates the need for personnel dedicated to the job of controlling the lights during events. Timing of lighting effects is an important factor in many events: the lights may need to be adjusted before and after half-time of a football game; they must be dimmed or turned off on cue for some concerts (rock star Prince refused to come onstage unless all of the lights were out).

At the time of the contract bid, much of the software and other components for the HVAC controls were already developed and configured into the JC/84 system. However, due to the special requirements of the facility, the lighting controls had to be custom designed and retro-fitted into both the JC/84 and the existing lighting contacts and wiring.

The JC/84 control system is a three level distributed processing network consisting of a headend mini-computer system, field interface microprocessors (FIDs), and

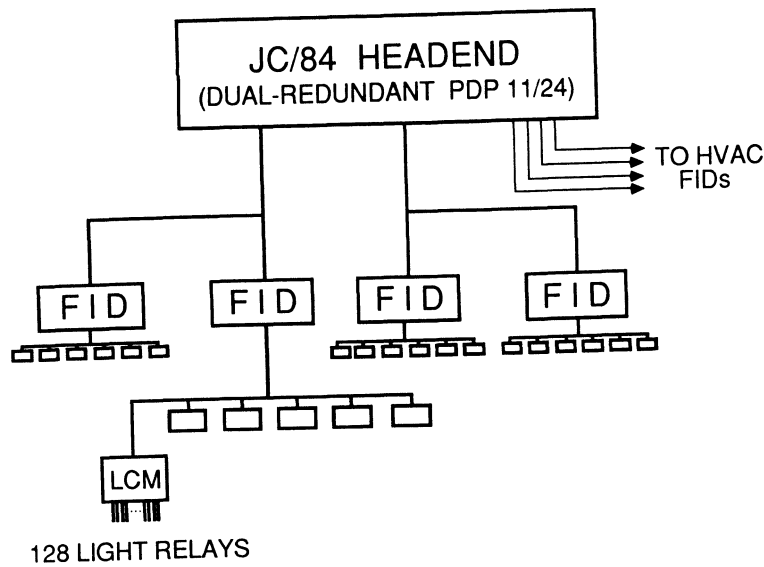


Figure 1. JC/84 System Architecture

multiplexors (MUXs), as shown in Figure 1. The headend provides the centralized monitoring and control functions, system-wide database, and operator consoles. The FIDs act as concentrators of messages from the field, and provide limited stand-alone monitoring, control, and diagnostic capabilities. The MUXs interface directly with the field hardware (sensors and actuators).

The lighting control system includes modifications to the headend software and FID firmware, and an entirely new MUX (the Lighting Control Multiplexor, or LCM). The headend lighting software is co-resident with the existing HVAC code, so that the lighting and HVAC systems can share the same operator consoles and disk media. Each FID is dedicated to either the HVAC or lighting control application, depending on the firmware that it contains.

There is one lighting FID in each geographical quadrant of the Superdome. Each FID communicates with six LCMs, and each LCM interfaces to 128 light relays. The system controls a little more than 2200 lights, with a total capacity of 3000 lights without adding more hardware.

Headend Computer

The headend system, the top level of the distributed network, includes two PDP-11/24s, operator consoles and printers (VT100, ISC 8500, and LA120), dual 10MB disk storage subsystem (RL02), serial I/O (RS-232-C) communications ports (DZ11) for peripherals and field communications lines; and the software to support the monitor, control, and communications functions for the top level of the

network. The second PDP-11/24 provides full redundancy. A manual failover switch, which re-routes the peripheral devices and communication lines, allows the operator to transfer control between the computers. The manual failover allows one computer to run the standard HVAC system while the other computer runs the lighting control system -- this was useful during the installation and field checkout of the lighting control system.

Field Interface Device

The FID, a MULTIBUS-based system, contains an 8088 microprocessor, serial I/O communications ports to support headend and MUX communications, and a diagnostic console. The firmware includes the VRTX real-time, multi-tasking executive and diagnostic facility, as well as the application code. There is no non-volatile memory (e.g., disks, tapes, NV-RAM) to maintain the FIDs memory-resident database; however, an uninterruptable power supply provides up to eight hours of battery backup to maintain system integrity.

The HVAC and lighting FIDs have the same hardware configurations; however, two separate sets of applications software support the HVAC and lighting functions.

Lighting Control MUX

The lighting control multiplexer (LCM) is a new device designed to interface to an existing relay panel at the Superdome. The LCM, a 6802-microprocessor system, is a binary output controller capable of controlling a maximum of 256 momentary contacts.

Table 1 - JC/84 Communications

MASTER	MEDIA	BAUD RATE	SLAVE
Headend	Bell 202	1200bps	HVAC FID
Headend	Bell 202	1200bps	Lighting FID
HVAC FID	proprietary	4800/9600bps	HVAC MUX
Lighting FID	EIA RS-422	9600bps	LCM

Communications

The communications between the distributed network levels uses a broadcast multipolling interface protocol. The protocol is a half-duplex master/slave relationship where the "master" device polls its "slave" devices. Table 1 shows the communications characteristics of the JC/84 HVAC and lighting system.

SOFTWARE DESIGN

The lighting control system provides the operators with five basic capabilities: to define and modify the database of light fixtures; to define and store lighting "modes" (lists of light on/off states along with time delays between actions) to be used for arena events; to activate a mode; to turn single lights on and off; and to examine the status of each light. All of these capabilities are provided as input and output options at the headend computer consoles.

The Light Database

Each light has a label by which it is known to the control system and operator. The label is a character string consisting of three four-character fields. The first field identifies the geographical quadrant (NE, NW, SE, SW) and ring in which the light is located, the second the bank, and the third the individual fixture. The data kept for each light includes the type of lamp (metal halide, quartz, or incandescent); its location in the control network (FID, LCM, and relay address); its current status (on, off, burned out, or no lamp present); the last date and time when the light was turned on or off; and the total accumulated burn time.

During system installation, the above information was entered manually into the database. The system automatically updates the current status (except for burned out lamps) and burn time data. Normally, the other information won't

change; however, the operator does have commands available which allow him to modify the characteristics of a light, or to add or delete lights.

Light Mode Definition

The most important capability of the system is the ability to define and activate a complete configuration of the arena lights for an event. We have termed such a configuration a "mode". A mode contains instructions specifying the lights to be turned on or off, in a particular sequence, and may contain time delays anywhere in the sequence. A fairly extensive set of user-friendly software tools was developed to allow the operators to "program" any possible lighting mode, to translate the user-written mode into a form suitable for machine processing, to store the mode in the database, and to cause the mode to be executed (activated).

The mode definition "language" consists of eight types of statements: mode name, ring declaration, bank declaration, light control action, delay, operator message, halt, and end of mode. These statement types are described below. Figure 2 shows some examples of mode definition statements.

The "mode name" statement associates a four-character label with the mode; this label is then used in all operations associated with the mode.

The "ring and bank declaration" statements allow defaults to be set for the first two levels of each light ID used in subsequent light control action statements. These statements save the operator from repetitive typing.

The "light control action" statement specifies a particular light (using its ID label), and two on/off actions that are to be taken. The two actions correspond to activation and de-activation of the mode. Usually, a light control action statement will specify that a light is to be turned

```

Light mode: FTBL ; { Televised Pro Football }

message: S = "Turning on Stadium work lights",
         R = "Turning off Stadium work lights" ;

ring: NWR4 ; { Inner ring, NW quadrant }

bank: BK02 ;

      fixture: LF01, S = ON, 0.1 sec,
              R = OFF ;
      .
      .
      .
halt: S, R ; { End of work light sequence }

```

Figure 2. Sample Lighting Mode Instructions

on when the mode is activated ("mode startup"), and off when it is de-activated ("mode reset"), but this does not have to be the case. Associated with each on/off action is a time delay from zero to twelve seconds before the next statement in the mode is executed.

A "delay" statement allows a time period from one second to nine hours (with one second granularity) to be specified. This time is to elapse before the next statement in the mode is executed.

An "operator message" statement contains character strings that are to be displayed at the operator consoles during mode execution. One message is specified for mode startup, the other for mode reset.

The "halt" statement causes execution of the mode to stop. This is the only mechanism within the mode definition language that affects the "flow of control" during execution. The operator can also halt mode execution by entering a command at the terminal. After a mode has been halted, the operator can command execution to begin at any statement number in the mode, or to resume where it left off. Thus, a mode may contain many "sub-modes" separated by halt statements, which the operator may activate in any order.

The "end of mode" statement acts like a "halt" statement, and is required as the last statement in a mode file.

To construct a new light mode, the operator exits the standard console screen handler and calls up a menu of mode definition options. The EDT screen editor is provided for writing the file of mode statements. When this file is complete, it is translated into a form suitable for storage in the database and for execution. The translator checks the validity of all light IDs, and for proper syntax; it allows free insertion of whitespace, and

allows comments anywhere. If errors are found, it identifies the statement in error in a listing file. Once a mode is successfully translated, it can be entered in the database.

The mode which sets the lighting for televised football games contains 1210 instructions divided into 3 sub-modes, with 1200 light control actions.

Light Mode Execution

Although light modes are defined and stored in the database at the headend computer, they are actually executed by the FIDs. Thus, the first step in activating a mode is to download its executable form from the headend to the FIDs. The FIDs do not have mass storage devices, so they can contain only one mode at a time.

The operator initiates the download of a mode with a single command. Due to the fact that there are four FIDs which must receive the download, and that other processing (for the HVAC controls) is going on concurrently, up to 20 minutes are required to download a large mode such as the football mode.

Once a mode is downloaded, the operator can cause it to be activated by entering a single command. The command causes the FIDs to (simultaneously) begin to execute the sequence of translated instructions. The operator can specify an ending, as well as starting, instruction number; otherwise, the mode will be executed until either a halt instruction or the end of the mode is encountered. The operator can abort mode execution at any time with a single command.

During the course of executing a lighting mode, each FID keeps track of the lights that it has commanded to start or stop. Once a minute, it reports this data to the headend computer, which uses the data to update the burn time history in the database.

Single Light Control

The operator can command either an individual light, or a bank of lights, to be turned on or off by entering a single command. To turn a single light on or off, its entire ID label is specified; to turn a bank on or off, the first two levels of the IDs of the lights in that bank are specified.

The single light control feature is not normally used during arena events; it is useful for maintenance and for developing new lighting configurations.

Displaying Light Status Information

The operator can obtain information about the status of the lighting system by several means. Reports which list current data for the lights can be generated on either a video console or a printer; these reports list on/off status, burn time, lamp type, etc. (The status of a light is determined from the last command issued -- there are no sensors to provide feedback information.) The operator can obtain a list of all stored lighting modes, and can print out a formatted listing of all of the statements in a mode.

There is also a color graphic display showing a quadrant, with the status of each light indicated using a color code.

SOFTWARE IMPLEMENTATION ISSUES

The headend software used in the lighting control application was added onto the existing base of software in the JC/84. This situation had the advantage of allowing us to use facilities provided by the existing code. However, it also posed two problems: the additional code required for the lighting control application would add to the processing burden; and the lighting code needed to fit "neatly" into the system from a configuration management standpoint.

The JC/84 headend software runs on a PDP-11/24 or 11/44, using version 4.1 of the RSX-11M operating system. The installation at the Superdome uses the slower 11/24 processor. The system must maintain communications with ten FIDs, performing various forms of processing in response to the data received; it must also service requests from four operator consoles and two printers. The system contains about 90 tasks (executable images), of which as many as 20 may be requesting service from the CPU at a time.

The two areas where we have experienced the greatest limitations on performance are database accesses and requests for dynamic system memory ("pool"). We found several years ago that the standard database access calls generated by FORTRAN (using the record manager utility RMS and Files-11) would result in unacceptably slow response, so we wrote our own database management utilities. When we added the code to support the lighting control application, we began to experience system crashes due to pool becoming depleted. Our solution to this problem was to provide a way to install, run, and remove tasks from FORTRAN.

Database Manager

The database manager in the JC/84 supports two simple file structures: fixed record

size, non-expandable, randomly-accessed binary data files; and tree-structured file directories. The files used by the system are created and initialized "off-line" (when the application code is not running). The application code typically accesses the database to read or write data elements from a particular record; less frequently the need arises to add or delete records from a file.

A database access is initiated from application code by means of a FORTRAN call to a standard database manager routine. This call results in a queued I/O request (QIO) being directed to a pseudo-device. The pseudo-device is simply a convenience to allow all database requests to be serviced sequentially by a single entity (the actual database manager code). The database manager takes the form of an Ancillary Control Process (ACP), which is accessed by a QIO from the pseudo-device driver.

The database manager contains code which maps to the requesting task space to obtain and return the calling arguments in the access request. The database manager also supports a set of cache buffers containing the most recently requested records; thus, an access request results in a disk access only if it is a write request, or if the record requested does not currently reside in the cache.

About a 300% improvement in maximum throughput for the HVAC control system was obtained by this implementation of the database manager. In addition to providing a performance improvement over RMS, this scheme minimizes the amount of code that must reside within the address space of each application task. And by providing a mechanism to ensure sequential servicing of database requests, it greatly simplifies the implementation of record locking and other database integrity functions.

Dynamically Installing and Removing RSX Tasks from FORTRAN

The JC/84 contains 21 tasks which process operator commands involving more than a single line of input. Under RSX-11M, an installed task seems to require about 50 words of pool space -- when we had all 21 of these tasks installed during the operation of the system, they reduced the available pool space by over 1000 words. This is a very significant amount in an RSX-11M system, which does not provide a secondary pool area.

To free up this pool space, we needed to find a way to dynamically install, run, and remove these tasks. This has to be accomplished from the top-level command processing task, which handles the initial command line entered by the operator. In addition, the data that the operator

enters in the command line has to be passed to the task which is being installed.

We accomplished these objectives by using the FORTRAN-callable SPAWN directive (see Sections 4. and 5.3.71 of the RSX-11M/M Plus Executive Reference Manual). The top-level command processing task spawns the MCR command line interpreter, passing the following command line:

```
RUN LB:[ , ]taskfile/TASK=taskname/PRI=60.
```

(In the actual command line, "taskfile" and "taskname" are replaced by the name associated with the separate command processing task.)

Before the spawn directive is issued, the command line data is placed in a global data area and tagged with the name of the task which is to be activated. The activated task gets its task name, then scans the global data area for the proper command line data. Included with this data is the unit number of the console from which the command was issued. This allows the activated task to prompt for and accept further operator input.

Custom Software Configuration Control

The JC/84 headend software consists of over 1200 FORTRAN modules, plus over 100 assembler routines and operating system object modules. During the time the lighting control system software was being developed, modifications were also being made to the "standard" JC/84 system to support other contracts. The lighting control application code had to be added into the rest of the JC/84 system to be delivered to the Superdome. But since lighting controls are not required for other contracts, the standard software had to allow the lighting code to be added or omitted during a software manufacture, without affecting the operation of the rest of the system.

The lighting control system added 125 new code modules to the system, and 13 new database files. We were able to merge this software into the standard system by modifying only ten of the "standard" modules, besides those which are normally modified to accommodate the configuration at a particular site.

We provided for a separation between the "standard" and "custom" software by identifying five areas of interface between the HVAC control system and any additional control functions that might be added: operator command processing, network (FID) message handling, report generation, initialization of global data, and the database. We then used several mechanisms of FORTRAN and RSX-11M to isolate the interface to one or two modules.

In the case of operator command and network message processing, separate tasks are used to process lighting commands and messages. Control is transferred to these tasks (by intertask queue invocations) when a message or command number corresponding to the lighting application is encountered.

In the case of report generation, a separate task could not be used; instead, the modules generating the reports on the lighting data were isolated into a separate overlay factor (".ODL") file. This file is read by the RSX-11M Task Builder, and defines the overlay structure of the modules named in it. A version of the factor file containing only a "null" factor is used for manufactures for sites other than the Superdome.

The global data area is initialized by means of a "block data" task, which defines the values that each entry is to take at load (application startup) time. Separate components (data structures) within the global data area are defined and initialized in separate files, which are incorporated into the block data module by means of the FORTRAN "INCLUDE" mechanism. Thus, modifications and additions to global data for lighting control were isolated to a few "INCLUDE" files.

FIELD INTERFACE DEVICE FIRMWARE

The applications code residing in the FID microprocessor is written mostly in the high-level language PL/M. This code relies on a vendor-supplied executive (VRTX) to accomplish task management, intercommunication, and other system utility functions. The entire firmware occupies about 56K bytes of EPROM memory.

The FID's main responsibility is to store and execute light modes, under control of the headend computer. To do this, it must manage a buffer of up to 36K bytes for storage of a mode, and must perform the processing required for the four mode instruction "primitives": light start/stop action, time delay, operator message, and halt.

A FID starts or stops a light by sending the light's relay address to the proper LCM. Commands for up to ten lights can be packed into a single message sent to an LCM. Such messages must be separated by a minimum of 0.1 seconds, due to the length of the pulse which the LCM must use to activate the relays. Other processing considerations limit the throughput of the system to 67 light actions per FID per second; this allows all the lights in the arena to be commanded on or off in ten seconds.

Despite the fact that the four lighting FIDs cannot communicate with each other,

they remain closely synchronized when executing a mode. They do this by replicating each other's actions for each instruction in the mode, up to the point of actually sending a light start/stop message to an LCM.

If communications are lost with the headend while a light mode is being executed, a FID will continue to execute the mode until either an "operator message" or a "halt" instruction is reached. At this point, the FID will wait until communications are re-established before executing any further mode instructions.

If communications are lost with an LCM, the FID will continue to execute a light mode and will queue all messages containing light control actions that were to be sent to that LCM. When communications are re-established, the messages are transmitted and the light control actions are then performed.

Each FID must also maintain and transmit data on light control actions to the headend. This is accomplished by using a circular buffer of time-tagged light start/stop data. A task periodically scans the buffer and constructs a message for the headend, removing the data from the buffer once acknowledgement of the data is received from the headend.

LIGHTING CONTROL MULTIPLEXER

Each LCM contains eight "relay interface" boards and one "CPU/Communications" board. The LCMs have to provide optical isolation from the existing relay panel, while providing an interface to the relay panel. They use the existing power source to control the relays. The design of the LCM needed to be simple and rugged as the ambient conditions in the lighting control rooms often reached temperatures of 100 degrees F (air conditioning is shut down between stadium events).

The application code for the LCM is written in 6800 assembly language. The code is used for communicating to the lighting FID and controlling the momentary binary output contacts. The entire firmware occupies about 1000 bytes of EPROM memory and 128 bytes of RAM.

The control of the binary output solid-state relays is achieved through programmable logic located on the CPU board. The programmable logic provides each binary output contact, an optically isolated solid-state relay (SSR), with a "register" to control the SSR's contact state (open or closed). Enable/disable control logic forces the SSRs to the open state when the logic is disabled, and allows the register to control the SSRs when enabled.

CONCLUSIONS

The computerized lighting control system's first event was a circus which was held in July 1985. The system worked flawlessly during all five performances.

The Superdome lighting system demonstrates the flexibility and customizability of a large control system hosted on a PDP-11 using RSX-11M. The lighting system provides the Superdome with a major additional capability and cost savings by combining lighting controls with an energy management system.

NOTES

PDP-11, VT100, LA120, RL02, DZ11, RSX-11, and RMS are all products of Digital Equipment Corp.

PL/M and MULTIBUS are products of Intel Corp.

VRTX is a product of Hunter & Ready, Inc.



VAX SYSTEMS SIG

VAX/VMS SYSTEM MANAGEMENT:
A PROBLEM IN RESOURCE ALLOCATION

Robert S. Branchek
ERI Training
New York, New York

ABSTRACT

The major resources of VAX/VMS are controlled by system mechanisms which allow tailoring of the system to meet different environments. Several of these mechanisms for the allocation of the CPU, main memory and disk space are discussed including software priorities, IPLs, priority boosting, automatic working set adjustment, disk quotas, and selected parameters and quotas.

The reason that there is a problem in the allocation of the various resources of a VAX/VMS system has nothing to do with its organization as a virtual memory operating system. It is simply due to its nature as a multi-user operating system: its virtual memory is a strategy used to solve the problem of the allocation of one of the resources of the system, main memory. All multi-user systems face the problem of how to best allocate the resources of a computer system. To solve this problem, mechanisms have been implemented by system designers, which are designed to be flexible, to tailor the system for particular user environments.

The job of a system manager, which is to manage the VAX/VMS system, must begin with the understanding of the various mechanisms implemented for allocating resources. Essentially all manager's responsibilities except for backup (we will assume that although operator's may perform backups, managers have responsibility for them) involve the use of mechanisms which solve problems in resource allocation.

For a manager of a VAX system to understand the designed allocation strategies available to them, they must first remember the resources of the system to be allocated. The three major resources of the system are the central processing unit (CPU), main memory, and the storage on the major peripheral storage device, the disk space. Though not one of the topics of discussion of this article, the allocation of other resources is also controlled by various mechanisms. For example, the process of spooling on the VAX is accomplished through the use of queues of requests for the use of designated printers, terminals, or devices viewed by the system as printers or terminals, e.g., plotters.

The word spool, as an acronym for shared peripheral output on-line, alone supports the notion of a solution to the problem of the use of some output devices on a computer system. The selection of print queue characteristics during the creation of print queues on VAX/VMS is an effort to tailor the use of these devices to the demands of the user environment of a computer system.

Although there are clearly other resources for allocation on a computer system, such as the printers or terminals already mentioned, much more effort is expended in solving the problem of the allocation of the CPU, main memory and disk space, and our discussion will center on these three resources. Both the unchangeable and changeable mechanisms of the system which control these resources will be discussed as will the solutions provided by VAX/VMS to the problem of resource allocation.

Allocation of The CPU

The demand basis for the allocation of the CPU is implemented on VAX/VMS through the interrupt priority level (IPL) mechanism. In this mechanism, almost all system operations are prioritized by the system designers according to the scheme shown in Figure 1. If we examine this figure, we quickly note that, since this is the overall priority scheme for use of the CPU, the first real work of the system is done to handle part of the I/O processing, performed by device drivers. The highest of all system priorities, the reaction to a loss of power, at an IPL of 31 (and within the group of system errors from IPL 25 - 31) is to put the system to a known state, a software crash, if a catastrophic error should occur.

This is within the priority range of routines which execute only to prevent the CPU from being used when reliable results to operations can't be assured. But this is not use of the system for actual user or system operations, only a mechanism to disallow use of the system when it might fail to function properly. Additionally, the highest priority of CPU use, other than for system errors, is at IPL 24 for the purpose of acknowledging a signal, delivered every 1/100 sec., to be used to update the stored system date and time. But this is a rather small amount of work. So, the operation of the device drivers, at IPLs 20-23, are effectively the highest priority system operations that are of major significance.

The use of the CPU for user requests is scheduled at an overall system priority of 7, but user code doesn't execute in the CPU until the IPL is lowered to 0, the lowest level. At this level of system use, only processes which are capable of executing code immediately can be selected to use the CPU (those in the process state computable or COM), and these are prioritized in the priority range shown in Figure 2.

It must be noted that the allocation of the CPU for use by users is granted only when there are no high priority system operations to be done. These high priority system operations are operationally defined as those that are initiated at IPLs greater than 0. User processes, the reflection of the user on the system, are scheduled by the system component, the scheduler, who chooses the highest priority process, with the priorities ranging from 0 - 31 (same range as the IPLs), from the list of all processes who may use the CPU immediately.

Partly as a result of the handling of the major component of I/O processing as a group operation, processes will be normally designated as being in a non-executable or wait state during which time they cannot compete for use of the CPU. Processes waiting for the system to perform data transfer for them are in the state local event flag wait (LEF). Local event flags are the synchronization mechanism used to signal that the requested data of a process has been transferred in either direction between a user's buffer and a device.

Additional characteristics are granted to a process by the assignment of one of the scheduling priorities, shown in Figure 2, in either the range 0 - 15 or 16 - 31. Processes assigned the same priority in the the range 0 - 15 do round-robin scheduling, attempting to share use of the CPU

equally. Each process will be granted use of the CPU up to an interval of time determined by a system parameter, QUANTUM.

Processes with the identical priority within this priority range, by design, should, if all other factors are equal (and they're not), receive equal use of the CPU time. Appropriately, a process in this priority range is termed a timesharing process. Processes in the priority range 16 - 31 do not share use of the CPU at the same priority. By design, at the interval of time defined by QUANTUM, the CPU will not switch use of the CPU to another process with an identical priority to the process which has use of the CPU currently (and is referenced as being in the CURrent process state). Processes in the priority range 16 - 31 should be assigned a priority within a well-defined hierarchy or ranking, one to a priority level, with the certain realization that conditions on the system, which would allow the CPU to be given to a process at a higher priority level in the range 16-31 (perhaps because the second process just entered the system), must be allowable. It becomes the responsibility of the site manager to make correct assignments of priorities in the range 16 - 31.

Our understanding of the assignment of different priorities, and their affect on processes use of the CPU, gives us a better understanding of the system parameter defined interval, quantum. Quantum becomes the interval of time at which the system attempts to switch use of the CPU to another process, with the identical priority to the process currently using the CPU, but only in the scheduling priority range of 0 - 15. It must also be remembered that the system would always switch use of the CPU immediately to a process that has a higher priority, throughout the entire priority range 0 - 31, than the one that is now using the CPU - it doesn't wait until the interval quantum to make a switch.

The assignment of the CPU is greatly affected by the states of processes. As noted earlier, a process must be in a state in which it is capable of using the CPU, the state COM, to be assigned it's use. A problem occurs, however, in the equal distribution of CPU use among processes having the identical priorities within the priority range 0 - 15, because some processes remain in wait or non-runnable states for different periods of time.

For example, if we had only two processes at the priority level of 4, and one was in the COM state 90 percent of the the time, while the

other process was in the COM state only 3 percent of the time, the first process, let's call it process A, should be chosen for use of the CPU much more often than the second process, let's call it process B. Process B is effectively overlooked for use of the CPU often simply because it is not in the state COM, or computable, from which the scheduler chooses processes to use the CPU. Yet, both processes were assigned the same priority level in the range 0 - 15, and, on this basis alone, were designated to receive equal use of the CPU.

The solution to this problem on VAX/VMS is the mechanism of priority boosting which is designed to even out the allocation of CPU use among processes assigned identical priorities in the range 0 - 15, which remain in the COM state for very different periods of time. Through the mechanism of priority boosting, a process such as process B, which remains in a wait state, would receive an elevation of its scheduling priority so that on the next occasion it is in the COM state, and can be selected for use of the CPU, it will have a better chance to receive use of it (by competing against other processes with a higher priority that haven't been waiting) in order to make up for not being able to compete during earlier selection intervals.

Table I illustrates two of the different reasons that processes receive boosts in priority and what the associated boost is for each event. Although there are other

operations which also have associated priority boosts for processes, the overwhelming majority of the time, process receive boosts based on previously requested I/O. From Table I, we can see that boost associated with direct I/O is 2, while the boost associated with buffered I/O is 4 or 6. Direct I/O is data transfer to and from fast devices such as disk and tape drives. Buffered I/O is data transfer to and from terminals and line printers. The higher priority boost associated with slow I/O seems appropriate, because a process that has requested I/O to slow devices should remain in a wait state for a longer period of time before it is capable of contending for use of the CPU than if it has requested I/O to a fast device like a disk. So a larger boost for slow I/O would be appropriate in relation to fast I/O.

The boost is always applied to the starting, base or pre-boosted priority, and lowered back to the base priority each time the process is subsequently scheduled to use the CPU. So a process like either process A or B in our example, starting with an assigned priority of 4, which receives

a boost of 6, has its current priority decremented by one the first and each time it's scheduled to use the CPU again (unless it's already back at it's base priority). In our example, it is scheduled to use the CPU at a priority of 9 the first time it is able to use the CPU after having requested the buffered I/O.

The decrementing of the priority of the process occurs so that the process receives a scheduling advantage for an appropriate duration only to make up for the time it was in a wait state and ineligible to use the CPU. To leave the process at an elevated priority after it had requested I/O and received a boost would give the process an unfair advantage over all other processes on the system for its entire existence. Decrementing its priority each time it is scheduled to use the CPU, allows it to get back to its original designated priority when an advantage is no longer necessary.

The boosted process is eligible for repetitive boosts even before it's returned back to its base priority, though a second boost would not be

applied if the resulting priority of the process would be lower than the priority resulting from the effects of the previous boost alone.

For example, if process A, after receiving a boost of 6 applied to it's original priority of 4, is first scheduled at a priority of 9, and subsequently requests I/O to a disk, it will be eligible for a second boost of 2 applied to its base priority of 4. If the second boost were applied, the priority of process A would next be 4 plus the boost of 2 but minus 1, or a priority of 5, rather the 8 it would have been scheduled at next as a result of the previous boost. So the second boost, if it would result in a lower priority for the process, is not applied because it would have the effect of penalizing the process unfairly. The second (and third and fourth etc.) boost is only applied if the process would have a higher priority than leaving it at the value determined by the previous boost.

System events, other than I/O completion (considered a system event because data transfer is handled as a group by VAX/VMS), also make a process eligible for a priority boost. All events which cause processes to receive priority boosts are mechanisms which serve to aid the allocation of the CPU as a resource among processes that, as designated by their original priority, should receive equal use of the CPU.

Main Memory Allocation

The allocation of the system resource main memory is dynamically adjusted after an original allocation of memory is given to process. The allocation of main memory is controlled both on a process basis and on a system basis. On a process basis, processes receive an initial allocation of main memory which is controlled by a quota, a mechanism which determines how much of some system resource a process can use. The quotas for most processes are set by fields in a system user authorization file, referenced for all interactive processes when a process is first created, at login time. Of the over 20 quotas for a process, three directly control its allocation of main memory.

WSQUOTA sets the maximum number of pages a process may grow to any time on the system. Although the size of a page is fixed at 512 bytes, the pages a process has in memory for its use varies as it references more and more pages (we will assume for our discussion here that these are pages of code from the runnable copy of the program on the disk, the image) until the limit set by WSQUOTA is reached, the point at which the process is normally prohibited from exceeding.

There must be times on the system when there are many more pages available for use than at other times. These pages will be listed on the free page list. The value of WSQUOTA, if it were used by itself to determine the maximum number of pages for a processes memory allocation, could prevent processes from using additional pages during the times when the need for pages of main memory is less than usual. So a second system parameter is used to allow processes to increase their use of pages during times of surplus.

This second quota, WSEXTENT, serves as the maximum number of pages a processes may allocate when there is a declaration of extra or surplus pages. The declaration of a surplus occurs when the number of pages on the free page list exceeds the number of pages (pages not specifically allocated for use) set by the value of the system parameter BORROWLIM. When the number of free pages is greater than the value set by the parameter, processes can use many more pages than would normally, without adversely affecting other processes or system components because these surplus aren't being used anyway.

Some other control mechanisms are necessary to avoid problems than can result from an uncontrolled use of the surplus pages. For example, there

must be a way of controlling the growth of an individual process so that it doesn't grab too great a share of the surplus pages. This is controlled by the system parameter GROWLIM. If, even when a process has been allowed to grow beyond its normal limit of WSQUOTA pages toward a maximum of WSEXTENT pages, the number of pages that are being taken by the process, the free pages, drops below the number set by the parameter

GROWLIM, the process is not allowed to get any additional pages. This system parameter serves as a fast cutoff to prevent a too rapid depletion of the surplus pages by an individual process.

Since each image a process executes may require different amounts of memory allocation, and it would be difficult to predict what each of their needs for main memory would be, the process quota WSDEFAULT is used to control the starting allocation of main memory for each new image a process runs. It would also be wasteful to leave the amount of memory a process had just used for the last image it ran for the next image since it may require far less pages. Resetting the amount of memory back to WSDEFAULT before the next image executes saves available memory for other processes which may require it. Also, the process is still free to request additional pages up to WSQUOTA.

Another problem, that can result in the allocation of these additional pages to processes, is that once the surplus has diminished to normal levels, the processes are now, effectively, over their normally designated allocations and may be considered to be too large for the total amount of available space in main memory. A mechanism to control this uses another system parameter, FREELIM, to set the threshold for the minimum number of pages which must be available for dynamic use (free pages). When the number of free pages drops below the number set by FREELIM, the surplus pages granted to processes beyond their WSQUOTA values, can be reclaimed by the system to satisfy its need for pages. The use of the name, BORROWLIM, for the threshold at which processes are permitted to gain extra pages, denotes that they are only borrowed and can be reclaimed by the system when necessary.

Perhaps the most interesting mechanism that affects the number of pages a process uses is automatic working set adjustment. In its original implementation, VAX/VMS version 2.5, the system monitored the

page fault rate of processes during a sample period of time and added or deleted pages based on the fault rate. Since page faults are a measure of the number of times a process references a page of information that is not in its allocation of space, or working set, they're a good basis for determining whether a process has either enough pages of memory and no adjustment is necessary, has more pages than it requires and can give up a few (with no detrimental effect on the process), or, a process, with a high fault rate, doesn't have enough pages and would function better with some additional pages. Using the sample interval set by AWSTIME (which should be equal to or a multiple of quantum), a process is granted WSINC number of extra pages if its page fault rate is greater than the value set by the parameter PFRATH. If the processes page fault rate is less than PFRATH during the interval set by AWSTIME, then WSDEC pages are taken away from the process. This mechanism allows the dynamic adjustment of the allocation of main memory for a process based on its past activity in a manner very much like the adjustment of the scheduling priorities of a process through priority boosting.

However, the automatic working set adjustment (AWSA) mechanism has not been fully implemented since VAX/VMS version 3.0. The default value of PFRATH is set to 0 by default, turning working set decrementing off and leaving working set page addition on. Pages are still removed from a process through a mechanism that removes pages from processes only after a system need for those pages has already been determined, rather than taking extra pages from a process which could be better used in another way.

This system mechanism for memory allocation is controlled by the system process, the swapper, scheduled to use the CPU at a priority of 16. Since version 3.0 the swapper does more than swap or remove working sets of processes in memory, to allow a larger number of processes to work on the system (by using the disk as an extension of main memory). It will now also reclaim space from a process by taking away first, the borrowed pages beyond WSQUOTA, and if necessary, taking even more pages, until the number of pages in the process is lowered to SWAPOUTPAGCNT, a system parameter. The swapper will also

attempt to write out the contents of the modified page list to the page file since this action would probably occur relatively soon anyway, and it (the swapper) may gain the needed space by this last action alone.

Allocation of Disk Space

These mechanisms will be described only briefly due to the time constraints of this presentation. The most obvious mechanism is the use of an optional file, QUOTA.SYS, which when created on any disk through the use of the utility program DISKQUOTA, enables direct control over the total number of blocks allocated on that disk by users of the computer system. The quota file must be created on each disk of the system that is to have control of the allocation of its space, and it can be subsequently disabled (turned off with no control of the amount of space allocated on the disk by users) or re-enabled. Both a permanent quota allocation, the total number of blocks allowed on the disk normally, and a one time overdraft additional allocation is allowed in order for users not to be trapped in a situation where writing out additional blocks of data would penalize them too severely.

A second mechanism used to control the allocation of disk space is the size of the index file that is established (but not necessarily preallocated) at the time the disk is initialized. The number of headers or records for files that are established within this file controls the total number of files that can be created on the disk, unless it is reinitialized.

A third mechanism controls the total number of entries in a directory of files with the same filename, filetype but different version numbers is set on the directories with a version limit specification.

Quotas, some of which were discussed earlier, set the amount of system resources which can be used by processes or jobs (a group of related processes and subprocesses). In addition, many of the parameters, shown in Figure 3, in addition to WSQUOTA and WSEXTENT, indirectly control the allocation of main memory space used for setting up dynamic

data structures by the system in a system work space known as nonpaged pool, the size of which is itself controlled by the system parameters NPAGEDYN and NPAGEVIR. The quota ENQLM, controls the number of 160 byte plus one longword (32 bits) locks, a synchronization mechanism primarily applied to files, that a process may allocate out of the nonpaged pool space.

A second quota, FILLM, controls the number of files that may be opened simultaneously, each of which allows a creation of a data structure generally known as a file control block which is also allocated out of nonpaged pool space. Several other quotas also

indirectly control the allocation of space of nonpaged pool space just as some system parameters and process privileges also control the allocation of other system memory space.

Allocation of the resources of the system, main memory, CPU and disk space is accomplished through the mechanisms of software priorities and interrupt priority levels, working set quotas and various system parameters and other quotas, and quota files and other indirect disk allocation mechanisms respectively. A fundamental knowledge of the workings of these mechanisms of VAX/VMS system is a prerequisite to the management of resources on VAX/VMS. An good understanding of VAX/VMS allows both quicker solutions to problems to be arrived at when they occur, and the anticipation and avoidance of other problems before they arise.

SYSTEM PRIORITY LEVELS (INTERRUPT PRIORITY LEVELS)

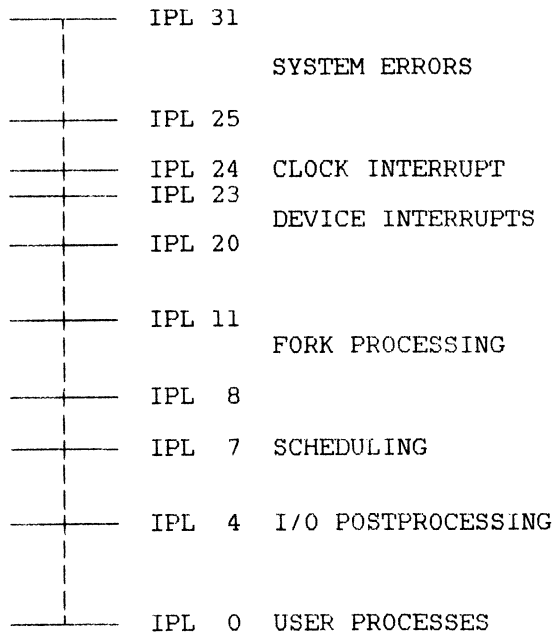


Figure 1

SCHEDULING PRIORITIES FOR PROCESSES

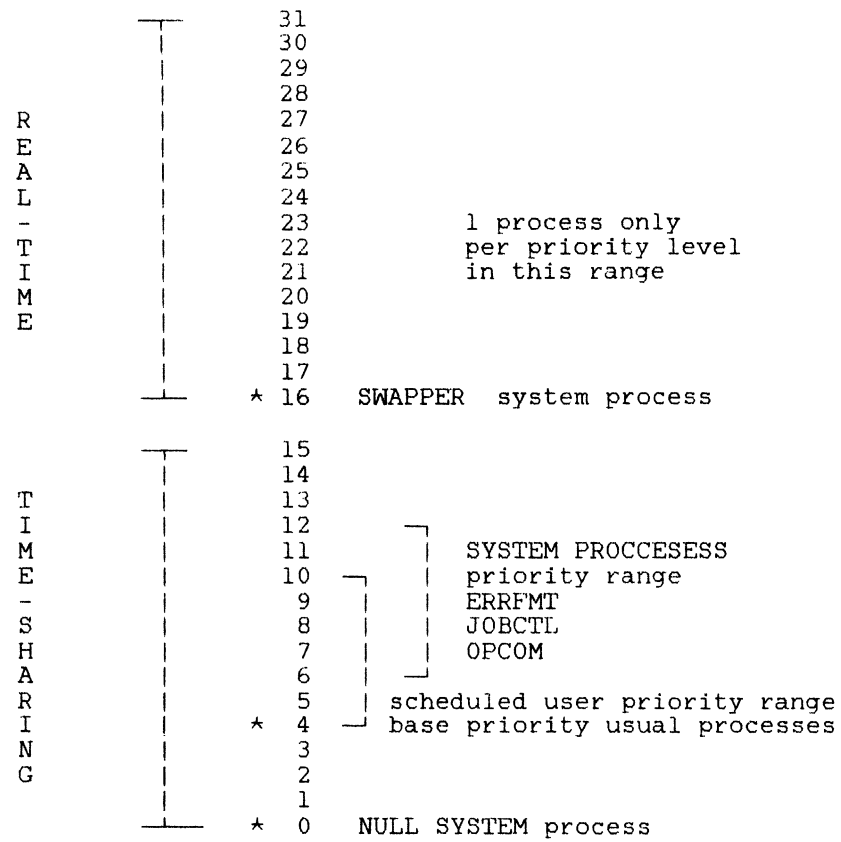


Figure 2

PARAMETERS AND QUOTAS AFFECTING WORKING SET SIZE

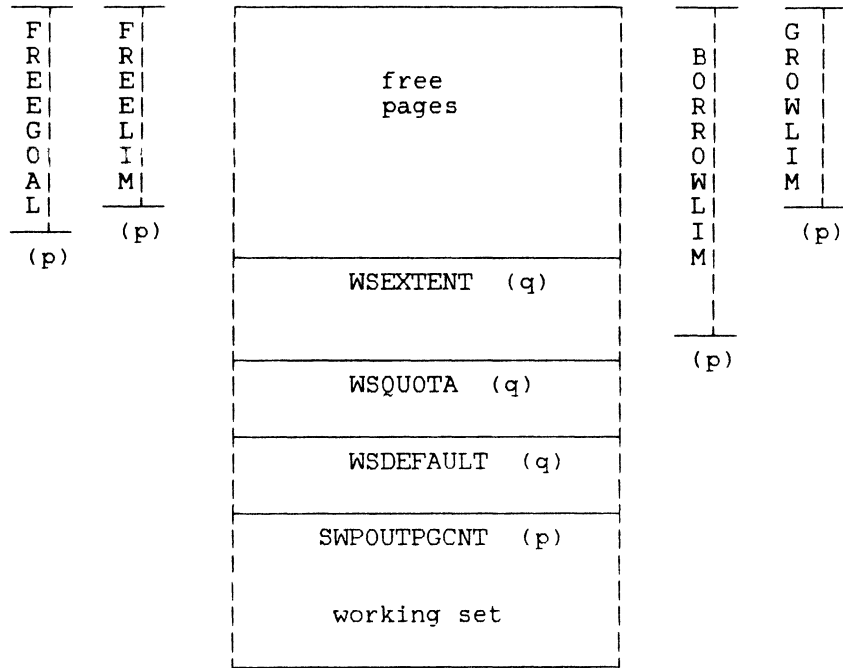


Figure 3

PRIORITY BOOSTS

reason	boost
BUFFERED I/O - SLOW I/O	6 (ACTUALLY 4 or 6)
DIRECT I/O - FAST I/O	2

Note: There are many other reasons a process can receive a boost in priority.

Table I

DESPERATELY SEEKING ACCESS

Identifiers, ACL's, and Alarms

Steven Szep
Chase Manhattan Bank
1 New York Plaza
New York, New York 10081

Abstract This paper begins with an overview of the new security features of VAX/VMS 4.x. It then explores identifiers, access control lists (ACL's), and security alarms as extensions to the traditional system management strategies for resource allocation. Finally, a tutorial on ACL design and maintenance demonstrates the power of the new features.

Disclaimer The information in this document is for informational purposes only and is subject to change without notice. Neither the author nor his firm assumes any responsibility for the material or its use.

Introduction to system security The material in this paper is intended for VAX managers who wish to protect their systems from accidental damage, illicit observation, or outside tampering -- without discouraging authorized users.

This paper will deal primarily with access controls. It will also treat data integrity as a most important "side-effect" of security management.

Design for security Our major goals will be to:

- o Protect those areas where unauthorized access is least acceptable;
- o Design software to minimize the opportunity for abuse and misuse;
- o Restrict users to those activities and resources needed to perform their assigned work;
- o Hold users accountable for their actions;
- o Prevent browsing through private messages;
- o Prevent theft of proprietary programs, plans, and computer time;
- o Protect systems from damage, both intentional and unintentional;
- o Permit certain users restricted access to certain areas in the system.

There is an inherent conflict between user friendliness and effective security.

Security problems Security breaches can be categorized as follows:

- o The user does the "wrong thing" and causes harm.
- o The user exploits insufficiently protected parts of the system.
- o The user breaks through existing controls.

Security needs Virtually every organization is a potential victim of computer crime. To prevent theft of computer files and services, there are three major tools available to the security manager: access controls, integrity checks, and encryption.

Review of user account management

Privileges restrict the performance of certain system activities to certain users. We should grant privileges on the basis of two factors:

- o whether the user has the experience to use it without endangering your system;
- o whether the user has a legitimate need.

If a user needs privileges in order to execute a program, we shall install a privileged image. Once installed, this program has specified privileges -- eliminating the need for the user to have them. To avoid security problems, we must prevent this image from displaying TRACEBACK information. Before installing this image, we link it using the /NOTRACEBACK qualifier of the \$LINK command.

A user's privileges are recorded in the user's UAF record as a 64-bit privilege vector. When a user logs in, this vector is stored in the header of the user's process. In this way, the user's privileges are passed on to the process created for him by LOGINOUT.

Users can use \$SET PROCESS /PRIVILEGE to control the privileges available to the images they run. Any user with SETPRV privilege can enable any privilege.

Login checks performed by VAX/VMS

When a user activates a terminal which is not already allocated to a user process, the system prompts for a name and a password. The person must type a username - password combination located in a UAF record, or the system will deny further access.

If the name and password are accepted, the system performs several operations:

1. Examines the login flags, beginning with DISUSER. If DISUSER is set, the login attempt fails.
2. If DISUSER is not set, verifies primary or secondary day restrictions.
3. Determines whether hourly login restrictions are in effect (via ACCESS, DIALUP, INTERACTIVE, LOCAL, and REMOTE qualifiers). If the current hour is restricted, then login fails.
4. If the login is successful, passes control to the command interpreter named in this user's UAF record.
5. If SYS\$SYLOGIN is defined, the logical name is translated and that procedure is executed.
6. Searches for the name of a login command procedure in this user's UAF record. If so and it exists, it is executed; otherwise, the user's command file named LOGIN.COM is executed, if it exists.

After a successful login, the command interpreter prompts for user input -- for example, DCL displays the familiar "\$". The user responds with commands acceptable to the command interpreter. The system prohibits activities which violate the user's privileges or exceed his resource quotas.

Each user is limited in the consumption of such resources as system memory. We set limits when we define the user to the system via creation of a UAF record. These limits control the way in which a process shares its allotment of a resource with any subprocesses it may create.

File protection VAX/VMS offers two primary protection mechanisms:

1. UIC's, which control access according to the user categories SYSTEM, OWNER, GROUP, or WORLD;
2. ACL'S, which specify the access granted or denied to specific users for each object in the system.

Access determination VAX/VMS employs the following scheme:

1. If an object has an associated ACL, the system uses it.
 - If the ACL specifically grants access to this user, it is granted.
 - If the ACL does not specifically permit or deny access, then the system uses UIC - based protection to make its determination.
 - If the ACL denies access, the system uses only the SYSTEM and OWNER fields of UIC - based protection to finally decide the access issue.
2. If an object does not have an ACL, the system uses UIC - based protection to determine access rights.
3. GRPPRV, SYSPRV, READALL, and BYPASS privileges amplify a user's access to system objects.

UIC-based protection Each account is established with a standard default protection code for all files the user may create. (This code can be changed with \$SET PROTECTION.) Each user is a member of a group.

Each user's UIC is defined in the SYSUAF. (This can be modified via \$SET UIC.) Each system object has an associated UIC -- identical to that of its owner, and a protection code which defines who is allowed what type of access. The relationship between the user's UIC and the object's UIC controls access to this object.

UIC - based protection controls access to objects such as files, directories, and volumes.

The system also provides overall volume protection, which is coded into the home block of a disk or mag-tape, as well as for record - oriented devices: terminals, line printers, mailboxes, etc.

Categories of users

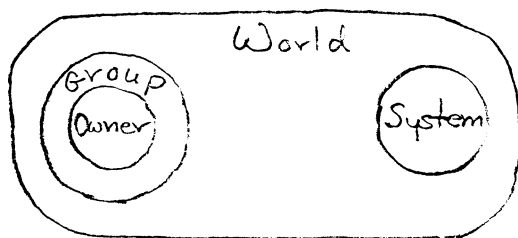


Figure 1 - UIC protection

Categories of access For \$SET PROT, we have

- o READ : examine, print, copy
- o WRITE : modify
- o EXECUTE : execute
- o DELETE : remove.

For \$SET ACL,

- o CONTROL : change protection and file characteristics.

Note: CONTROL access grants the user all of the privileges of the object's actual owner.

Special users Four user privileges can affect the access a user actually receives, regardless of his UIC or any current ACL: SYSPRV, GRPPRV, READALL, and BYPASS. Whenever we define ACL's or UIC - based protection codes, we should realize that users with these privileges are entitled to special access to "protected" objects throughout the system.

A user can access a resource as soon as the system finds a user category he fits into which grants the access he has requested. Thus, in order to deny access to a user category, we must deny access to all of the outermost categories.

Access to files We should avoid obvious names for our directories and important files. Purge regularly, and delete often.

MAIL files should only be accessible to yourself and the system (for MAIL delivery and backups). In addition, it is important to review revision dates on personal files, in order to reveal illicit tampering.

Naturally, be wary of running a command procedure or program image from an undependable source.

Note: To protect a file totally, we must protect both the file itself and the directory in which the file is stored.

Access control lists and identifiers

ACL's are used in conjunction with the standard UIC - based protection scheme in order to "fine - tune" protection whenever necessary.

UIC's are unique to one user and can be numeric -- for example, [300,-100] -- or alphanumeric -- as in [GROUP1,SAM].

General identifiers can be selected by the security manager; they are used to identify a specific subset of the userbase. They are alphanumeric -- for example, PERSONNEL.

VMS itself defines certain system - wide identifiers:

- o BATCH for batch jobs
- o NETWORK for DECnet tasks
- o DIALUP for dialup users
- o INTERACTIVE for interactive users
- o LOCAL for local-terminal users
- o REMOTE for network users

When a user logs in, the identifiers he holds in the rights database (RIGHTSLIST.DAT) -- including UIC and system - defined identifiers -- are copied into a "rights list", which is part of his process. This rights list is the structure which VAX/VMS uses to perform all protection checks. (Additional identifiers may appear in his rights list; they are placed there by the Login software or by in stal lation - specific software.)

You must specifically define which access to grant or deny to the holder of an identifier, for each object requiring this level of protection. If a few identifiers are required to represent differing access needs for each object, the software creates a list of multiple entries: the so - called "access control list". (Each entry in an ACL is called an "access control list entry", or "ACE".)

ACL's may be created by VMS by default, by the system manager for specific system objects, or by users to protect their own files.

There are two very good reasons for using ACL's:

- o to provide access to an object in a way which differs from UIC - based groupings;
- o to set off security alarms when access to an object succeeds or fails.

Access control list entries There are three types:

1. Identifier, which controls access allowed to one user or a group of users
2. Default, which defines the DEFAULT protection for a directory, so that the protection can be propagated down to its files and subdirectories
3. Security alarm, which provides an ALARM message when an object is accessed in the designated way.

Note: Security alarms are always placed at the beginning of an ACL.

Managing ACL's VAX/VMS provides two levels of protection:

- o access control lists (ACL's);
- o protection masks.

ACL's provide the first level of protection and are optional. When no ACL exists, the object is protected as specified by its (UIC - based) protection mask.

A protection mask consists of four fields, each with four indicators. Each field applies to one category of file ownership. Each indicator within a field applies to one category of file access.

The fields and indicators are as follows:

Field	Own'ship
SYSTEM	system
OWNER	owner
GROUP	group
WORLD	world

Indic.	Access
R	Read (allocate)
W	Write
E	Execute (create)
D	Delete

Formats of ACE's An ACL consists of one or more ACE's. Each ACE can have one of three formats:

- 1) (ALARM__JOURNAL=name)
- 2) (DEFAULT__PROTECTION, - ownership[:access],...)
- 3) (IDENTIFIER=identifier, - OPTIONS=option - +...,ACCESS=access+...)

General system access Here are some "hints and kinks" for the security - conscious system manager.

1. Do not assume that specifying ACCESSNONE for an identifier will absolutely prohibit the holder(s) of the identifier from accessing the object.
2. Watch for errors in the order of ACL entries. Place the ACE's which grant the widest access rights immediately after the most - restrictive ACE's.
3. Do not place ACE's on everything: performance will suffer. Apply ACL's upon objects with discretion. Be aware of how the object will be used or may be abused.
4. Use general identifiers to create practical groups of users -- in order to avoid long ACL's.

5. Update the SYSUAF and relevant ACL's whenever a user leaves your firm.
6. Maintain the shortest and most current ACL's possible.
7. Use \$DIRECTORY/SECURITY to display current information about file security.
8. Use the ACCESS field in the SYSUAF when there is only one set of primary and secondary hours for all types of logins. (Otherwise, use LOCAL, REMOTE, DIALUP, INTERACTIVE, BATCH, or NETWORK, as appropriate.)
9. Build a separate CLI for certain users, and set CLITABLES and DEFCLI in the SYSUAF accordingly.
10. Impose ACL's on the system program files in SYS\$-SYSROOT:[SYSEXE].
11. Restrict account duration, via the EXPIRATION field in the SYSUAF.

The problem with UIC-based protection
That a firm is organized into several departments suggests that individuals in the same department perform many of the same functions. Employees in a specific functional area communicate frequently with one another and, in general, share the same data.

A second reason for the departmental scheme is that everyone in the company should NOT have access to all company data. The purpose is to keep confidential information hidden from those not meant to have access to it, AND to protect the integrity of this data.

If we set up UIC groups according to this "natural" departmental organization, you can partition, and protect, data according to these groupings. Users who typically share data and influence one another's activities should be placed into one group. As a consequence, the users who should not have access to these objects will be assigned to other, distinct groups.

But, what do you do for that information to which many users, across departmental lines, request access? If many groups of users are able to access such data, the important protection feature is effectively destroyed. Thus, UIC-based protection does not offer reliably secure file protection.

The best solution is not to repeatedly restructure your UIC groups in an endless effort to find some "workable" situation, but to extend the identity of these groups via ACL's.

Tutorial on ACL design As manager, you must add new user accounts, assign them their initial passwords, and modify their accounts in order to amplify or reduce their permitted activities and access to system resources.

Your primary tools for these tasks include:

1. AUTHORIZE, the VAX/VMS user account management program;
2. DISKQUOTA, the VAX/VMS disk - space allocation program;
3. EDIT/ACL, the editor for creating and modifying access - control lists.

Your site's operation affects how you set your user's accounts. Who requires which computer resources? Try to develop several templates which work for certain "classes" of users.

First, you should map out the functions which you expect the users to perform on your system. Next, look for common groups of users who are involved with a particular function -- say, quality assurance -- or engaged in a specific activity -- say, a program development team. You will thereby gain a certain perspective on the logical groupings in your organization. This information forms the basis upon which you can place users into groups.

You should keep in mind that such groupings will have an impact on file protection and will influence the granting of privileges -- particularly, GROUP, GRPNAM, and GRPPRV.

Case #1: "Fine - tuning" file protection

Miami Devices is a manufacturer of peripherals for sixth - generation computer systems.

MD's payroll is one component of this firm's central computer system. Alice Atlas, executive secretary, has drawn up a list of individuals who require access to PAYROLL.DAT, the payroll file. Alice decides to make UIC groups correspond to her firm's departments, as shown in the following table.

Username	Dept.	Access
HHIGGINS	Exec.	RWED
AATLAS	Exec.	R
JCHEEZEE	Acctg.	RWED
PNICHOLS	Acctg.	R
JDONNE	Mfg.	R
BJONES	Mfg.	R
MSHELLEY	Mfg.	R
TTURKEY	Shpg.	R
BCANTO	Shpg.	R
SPEREZ	Sales	R
DPERIGNON	Sales	R

Figure 2 - Departments in Miami Devices

President Higgins decides that dialup terminals cannot be used for accessing payroll information at his company.

Alice notices that many of the prospective users on her list share the same access requirements and, indeed, fall into one of two groups. The determination is made to define two general identifiers:

1) PAYROLL, for all users who need "RWED" access to PAYROLL.DAT ;

2) PAYREAD, for those who only require "R" access to this file.

As a result, the following actions are taken:

```

$SET DEFAULT SYS$SYSTEM
$RUN AUTHORIZE
UAF> ADD/IDENT PAYROLL
UAF> ADD/IDENT PAYREAD
UAF> CREATE/RIGHTS
UAF> GRANT/IDENT PAYROLL HHIGGINS
UAF> GRANT/IDENT PAYROLL JCHEEZ
UAF> GRANT/IDENT PAYREAD PNICHOLS
UAF> GRANT/IDENT PAYREAD JDONNE
UAF> GRANT/IDENT PAYREAD BJONES
UAF> GRANT/IDENT PAYREAD MSHELLEY
UAF> GRANT/IDENT PAYREAD TTURKEY
UAF> GRANT/IDENT PAYREAD BCANTO
UAF> GRANT/IDENT PAYREAD SPEREZ
UAF> GRANT/IDENT PAYREAD DPERIGNON
UAF> EXIT
$SET DEFAULT MD$PAY
$SET ACL/ACL=(-IDENTIFIER= -
$_PAYROLL+LOCAL,ACCESS=READ+ -
$_WRITE+EXECUTE+DELETE), -
$_(ID=PAYREAD+LOCAL,ACCESS=READ),-
$_(ID=DIALUP,ACCESS=NONE)) -
$_PAYROLL.DAT

```

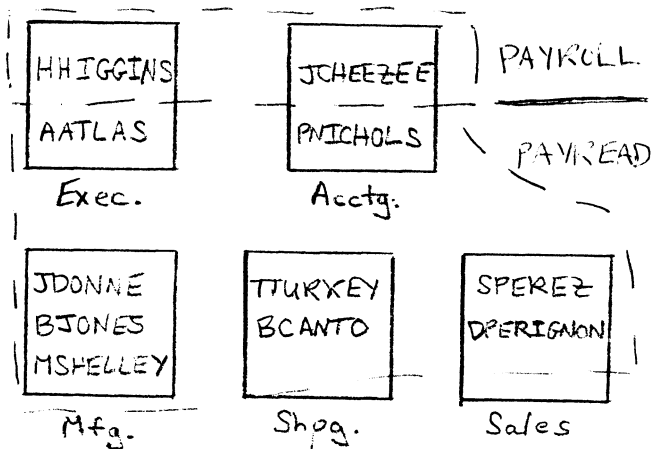


Figure 3 - Access to PAYROLL.DAT

Case #2: Chargeback, via resource control

One of the more interesting ways in which you can control the ownership of files -- which, naturally, also effects the outcome of protection checks, as well as the charging of disk space against preset quotas -- is the use of the RESOURCE attribute with identifiers.

The RESOURCE attribute attached to an identifier forces file - space to be owned by, and charged to, that identifier.

Jean Dark works as a consultant for both the Computer Science and Comparative Literature departments at Rambo University. To charge file usage properly, Colonel Mustard, the VAX systems manager, sets up the project identifiers "COMPSCI" and "COMPLIT", and grants both to Jean with the RESOURCE attribute.

```

$SET DEFAULT SYS$SYSTEM
$RUN AUTHORIZE
UAF> ADD/IDENT COMPSCI /ATTR=RES
UAF> ADD/IDENT COMPLIT /ATTR=RES
UAF> GRANT/IDENT COMPSCI JDARK/ATTR=RES
UAF> GRANT/IDENT COMPLIT JDARK/ATTR=RES
UAF> EXIT
$RUN DISKQUOTA
DISKQ> ADD COMPSCI
DISKQ> ADD COMPLIT
DISKQ> EXIT

```

When Jean logs in, she is told to create two subdirectories -- relating them to these identifiers, as follows:

```

$CREATE/DIR [JEAN.COMPSCI] -
$_/OWNER=[COMPSCI]
$CREATE/DIR [JEAN.COMPLIT] -
$_/OWNER=[COMPLIT]

```

Thereafter, when Jean writes files into the [JEAN.COMPSCI] directory, they receive the owner identifier "COMPSCI". When Jean writes files into [JEAN.COMPLIT], these belong to "COMPLIT". And, when Jean writes files anywhere else, they receive the owner UIC of [JEAN] -- if her default directory is not owned by "COMPSCI" or "COMPLIT", of course.

Case #3: Resource allocation

Third Rail Software is best known for its Ivory Tower Management Program, "ITMP" -- which is written in Pascal and runs on the VAXintosh mega - microcomputer.

Charlie Cola, President, Founder, CEO, and Director of Software Development, hears from his programming staff that their development system is often bogged down when many users perform interactive compiles.

Johnnie Lee Programmer is given the task of rectifying this situation. Johnnie Lee decides to use an identifier to disallow interactive access to the Pascal compiler:

```
lm 10
$SET DEFAULT SYS$SYSTEM
$SET ACL/ACL=(IDENTIFIER= -
$_ INTERACTIVE,ACCESS=NONE) -
$_SYS$SYSTEM:PASCAL.EXE
```

Displaying the rights database It is important to regularly scan the rights database for correctness and completeness. Two AUTHORIZE commands are used for this purpose: SHOW /RIGHTS and SHOW /IDENTIFIER.

Auditing access to sensitive files If one of your users has important files which she feels may have been improperly accessed, we can audit access to those files via a security alarm ACE in the ACL for those files.

If user STONES wants to know when her file REVIEW.MEM is being accessed, STONES would add an ACE to the ACL for this file:

```
$SET ACL REVIEW.MEM /ACL = -
$_ (ALARM=SECURITY,ACCESS = -
$_ READ+WRITE+DELETE+CONTROL -
$_ +FAILURE+SUCCESS)
```

Note: Enabling too many alarms results in failure to monitor each alarm and fosters a lax attitude about alarms.

Conclusions I will end with a few cautionary remarks and my wishlist for improvements to VAX/VMS security.

Security cautions

"Do not do unto others as you would have them do unto you. Their tastes may not be the same." - G.B. Shaw

A user cannot specify a secondary password in an access - control string for file access over a network.

There are no restrictions on files and directories to which a detached process can gain access. -- This is still true for UIC - based protection, but NOT for ACL protection!

Prohibit the general use of SDA and PATCH.

A menu restricting a user to certain operations is the only true security in a VAXcluster?

Security "wishlist"

"A journey of a thousand miles must begin with a single step." - Chinese proverb

There should be a special option which permits no one except the security manager to override an ACL ... similar to MOUNT's request - for - operator - assistance.

When a user possesses some privilege which entitles access regardless of ACL protection, give us the option to query a security terminal -- in order to permit or deny actual access.

Disallow \$EDIT/ACL of a file UNLESS the user has SECURITY privilege.

The system should retain three passwords for each user: the current (active) one and the two previously used by a user. \$SET PASSWORD should not allow the reuse of these passwords.

References

1. Guide to VAX/VMS System Security
2. Guide to VAX/VMS System Management and Daily Operations
3. VAX/VMS DCL Dictionary
4. VAX/VMS Utilities Reference (ACL Editor)
5. Guide to VAXclusters
6. Guide to Networking on VAX/VMS
7. Introduction to VAX/VMS, Terry C. Shannon (Springhouse, PA: Professional Press, 1985).
8. "Securitrieve", Al Cini, in VAX Professional (Vol. 7, No. 4).
9. "Defending Against Trojan Horses", Larry Kilgallen, in Pageswapper (Vol. 7, No. 5).

A Central Message Processing Facility

Georgia A. Pedicini
Los Alamos National Lab
PO Box 1663, Mail Stop H810
Los Alamos, NM 87544

Jamii K. Corley
EDS
Albuquerque, NM

ABSTRACT

The control system for the linear accelerator at the Los Alamos Meson Physics Facility (LAMPF) requires a centralized message processing capability. This requirement led to the development of a Message Dispatcher process to log anything from informational messages to hardware failures and application program crashes, for a variety of users. Since the Message Dispatcher must be available at all times, error recovery was also required. This paper describes our experiences in creating such a process. Features discussed include the system wide message facility, the condition handler and the exit handler. Problems discussed include aspects of mailbox communication and file organization, process quotas, and the condition and exit handlers.

1.0 OVERVIEW

The control system at the Los Alamos Meson Physics Facility (LAMPF) is a collection of programs, routines, and device drivers run from consoles to control and monitor devices along a linear accelerator. The two main sub-systems are the Console System and the Data System. A console consists of a color CRT and a collection of touch panels, graphics devices and knobs. For each console there is a Job Controller that interprets operator commands and starts other processes as necessary. The Data System provides standard software interfaces to the control and monitoring devices. The main structure of the LAMPF Control System (LCS) consists of the processes and routines required to interact with these two sub-systems. One of the requirements of this system was to have standard handling of all error, warning and informational messages. The Message Dispatcher (MSD) provides this capability while still allowing for flexible treatment of different types of information.

The VMS message facility was chosen as the vehicle to centralize the location of messages. This also minimized the traffic between routines, and contributed to the uniformity of error handling. In addition to VMS messages, many site-specific messages are defined in include files. To simplify the use of message codes by application programs, all LCS messages are collected into a shareable message image, and a pointer to this image is put into the

LCS run-time library. Programs linked to the run-time library are automatically given access to LCS messages. In addition, the SET MESSAGE command is executed at login for all LCS accounts, providing the same linkage for users.

The Message Dispatcher acts as a central processor for incoming messages, using optional routing information to decide where the messages are to be sent. Five lines of each color CRT, called the scratch pad, are reserved for the most important messages. There are also on-line log files and hard-copy units. In the present configuration, one file and hard-copy unit are used for system messages. This is also the default route if none is specified. A second file and hard-copy unit are used for accelerator messages. Output to these hard-copy units is presently coupled to the log files, so that all important messages are available in the on-line files.

Messages are passed to the Message Dispatcher through one of series of mailboxes. MSD uses a special QIO to advise the console driver of the permanent mailbox (DRIVER MBX) for logging hardware errors. All LCS-specific drivers can then use DRIVER MBX to communicate with MSD. Each console has a mailbox associated with it, called MSDMBXx, where the last letter indicates the console name (ex. MSDMBXA for console A). A process permanent file is created by transparent initialization, and equated to both SYS\$OUTPUT and SYS\$ERROR, resulting in a single copy of each error

message. This is then assigned to the appropriate mailbox when a process is started by a console Job Controller. Since we are trying to standardize on MSD for all logging purposes, a general purpose mailbox (GEN DEL MBX) is also available. This mailbox is used by processes not running through the Job Controller, and by DCL routines. MSD itself uses this mailbox to send a self-diagnostic message.

Expansion capabilities are built into the program, allowing for a total of eight consoles with color CRTs, four hard-copy units, and twenty-four files. The original configuration consisted of a single console and two log files and hard-copy units. Two more consoles have since been added.

The Message Dispatcher program is written in FORTRAN, and runs on a VAX 11/780 under VMS. The original version of the program was put into production under version 3.4, and has survived upgrades through version 4.2 with very little trouble. The next major modification planned for the Message Dispatcher is to run it on two VAX 11/780s in a cluster environment. Each VAX will have least two consoles, and access to system and accelerator hard-copy units. Since these devices are not shareable, the plan is to run a copy of the Message Dispatcher on each VAX to communicate with the privately owned devices, while sharing access to the log files.

2.0 PROGRAM ORGANIZATION

All real work in this program is done at AST level. The main program simply sets up everything, and hibernates. "Everything" consists of opening files, assigning hard-copy units and CRT lines, and creating mailboxes and setting up attention ASTs on them using SYS\$QIO. The appropriate routine or set of routines is then triggered whenever something is written to a mailbox. When processing is complete, the message-processing routines reset their own ASTs. This required some special handling, and is discussed later. Control then returns to the main program, still hibernating, unless an AST is pending, or until another AST is delivered.

A subroutine is used to open the log files, since they require a great deal of specialized information. This will also make it easier to add files as needed. The assumption is made that a copy of the file already exists, so the file type OLD is used. If there is an error on the first try, the open is retried using a file type of NEW. Only after the file open has succeeded is the unit marked as available to receive messages. The CRTs and hard-copy units are also not marked as available

until they have been successfully assigned.

Error recovery is vital to this application. This has been implemented by including an exit handler and a condition handler. The exit handler's job is to restart the Message Dispatcher, helping to ensure that message processing is always available. The need for a condition handler developed for two reasons. First, most errors detected by MSD are not fatal to the application, but should not be completely ignored. Having a condition handler allows MSD to treat these errors as informational, log them, and continue to function normally. Secondly, if MSD does stop, a traceback is written to the files assigned as SYS\$OUTPUT and SYS\$ERROR. When the Message Dispatcher is restarted, it latches onto the old copies of these files. At the time of the original implementation, these files could not be read while MSD was running. This problem could be avoided to some extent by determining the reason for failure before restarting MSD. However, this can take time, and if the Message Dispatcher fails, the first priority is to get it running again. This led to the file WARN.LOG, written by the condition handler, and readable while MSD is running.

The mailboxes can be written in several different ways, including QIOs, FORTRAN writes, LIB\$SIGNAL/STOP, and a locally written library routine to pass status codes to SYS\$OUTPUT. Messages sent from drivers have a different format from process messages. A driver message is a packed 30-byte array, containing the message type (code), the unit number of the device in error, and the device name. Process and DCL messages consist of optional routing information and text. The routes are parameterized for process messages; system symbols are being instituted for DCL route parameters.

3.0 MESSAGE DISPATCHER INPUTS AND OUTPUTS

When the console Job Controller starts a new process, it makes the connection between SYS\$OUTPUT, SYS\$ERROR, and the correct mailbox for that console, automatically routing a traceback through MSD. Standard FORTRAN writes to unit 6, with optional routing information, also go through MSD.

```
WRITE (6, fmt) [MSD_K_route, ] text
```

Non-console processes can open GEN DEL MBX for standard FORTRAN writes. They can also use the system service SYS\$CREMBX to identify the mailbox channel, and use SYS\$QIO for writes.

```
OPEN      (UNIT=n,      FILE='GEN_DEL_MBX',  
TYPE='OLD')
```

```
WRITE (n, fmt) [MSD_K_route, ] text
DCL routines can open GEN_DEL_MBX for
writes.
```

```
$ OPEN outfile GEN_DEL_MBX
$ WRITE outfile "[route codes] text"
$ CLOSE outfile
```

There are also several ways to pass through status messages. For processes running under the Job Controller, LIB\$SIGNAL and LIB\$STOP automatically send their status messages through MSD. Any messages received in this manner are written to the system log and hard-copy unit, since no routing is possible. For those status messages that need to be sent to some other destination, a locally written routine called LCS_PUTMSG is available. This routine performs the functions of the system service SYS\$PUTMSG to format a status message, and sends the message, including optional routing information, to MSD. Generally, this is a simple matter of writing the message to SYS\$OUTPUT. For programs which are being run from a console, the mailbox connection for SYS\$OUTPUT has already been made by the Job Controller. In order to have messages from non-console processes logged, GEN_DEL_MBX must be used. This mailbox is only opened for a non-console process if the optional routing is specified. Once GEN_DEL_MBX has been opened for a process, all further messages from that process are written to both the mailbox and SYS\$OUTPUT.

The output messages are formatted differently depending on the type of message and its destination. All driver messages use the following 2-line format (<time> is dd-mmm-yyyy hh:mm:ss.xx), and are always output to the system log and hard-copy unit.

```
<time> Device in error is <dev>, unit <#>
<time> %FAC-S-MSGID <message text>
```

Other messages sent to the files and/or hard-copy units have the following 1-line format, where nnnnnnnn is the PID. When the console Job Controller creates a process, it appends a 3-character console identification code to the process name (ex. A0, C1). The process name can have up to 12 characters, including the console identification code. Originally, a message length of 132 characters was used, so that each message would fit on a single output line. Unfortunately, some status messages weren't entirely readable, so an arbitrary message length of 150 characters is now being used.

```
<time> nnnnnnnn PROCNAME text
```

CRT messages consist of the process name

and the message text. Due to space limitations, these messages must be less than 48 characters. The five-line scratch pad area is scrolled whenever a new message is written, and the most recent message is flagged, effectively limiting the CRT message to 47 characters. If a significant part of the message will be lost when it is truncated to this length, the Message Dispatcher will also send the complete message to the system log file and hard-copy unit. For CRT messages, the process name is limited to 9 characters, and the console identification is not used.

```
>PROCNAME text
```

All the information used in the driver message is contained in the 30-byte input message. The status code is extracted and sent to SYS\$GETMSG. The resulting string along with the unit number, name, and the FAO directive for the present date and time are sent to SYS\$FAO to be formatted. The complete 2-line message is then written to the system hard-copy unit using SYS\$QIO, and to the system log file using a FORTRAN write.

Process messages require some information in addition to that contained in the message text. First, the first word in the mailbox I/O status block (IOSB) is checked for the status of the mailbox write. If the status is SS\$ENDOFFILE, it indicates that a null message was sent, and no further processing is done. Other than this, a success status is assumed. These blank messages frequently turned up as termination messages. The second word in the IOSB is the byte count. This turns out to be whatever was requested in the call to SYS\$QIO to set up the attention AST, so it is not useful in detecting blank messages, or for any other purpose in this application. The second longword in the IOSB is the process id (PID), which is used to look up the process name and also appears in log file messages.

Route codes, if any, are always at the beginning of the message text. These codes are stripped from the text, and checked against the available devices. All messages received from a process cause resolution of a possible MYCRT route. If invalid (i.e., the process is not running at a console, or the message actually came from a DCL routine) MYCRT is identified with the default route. The message text is also stripped of any non-printing characters. Any requested CRTs receive the short form of the message, which may contain special color formatting sequences interpreted by the color CRT driver. These color formatting codes, if any, are then removed from the text, and the long form of the message is sent to any requested files and hard-copy units.

4.0 FILE ACCESS AND ORGANIZATION

Since MSD is a permanent process, on-line access to the log files had to be provided. The program PRTLOG, using the Command Definition Utility, allows access to MSD files with several sorting options. These are:

- o /FILE= one of the on-line log files, or MSD's own error log file; defaults to system log if not specified
- o /OLD previous version of requested file; MSD re-opens new file periodically to keep size manageable
- o /PROCESS= process to search for; defaults to ALL; wildcarding available; special "process name" of HW selects all hardware errors
- o /PID= overrides PROCESS if both selected
- o /AFTER= defaults to midnight today
- o /BEFORE= defaults to current time
- o /OUTPUT= defaults to terminal

This naturally brings up the question of file organization. Sequential files had no advantages for this application. Indexed files would have provided the search features required by PRTLOG. However, neither indexed nor sequential files flush the internal buffers immediately. This means the most recent information written by MSD would not necessarily be available to any program reading the file (PRTLOG). This arrangement was not acceptable, so we were forced to use relative files. An alternate solution, using the RMS service \$FLUSH, might have solved the problem, but only at the cost of considerable overhead per file write.

For relative files, the records must be fixed-cells of a specified length, achieved by using RECORDTYPE='FIXED' and RECL=n. In this case, RECL=175. In order to allow PRTLOG to access the files, the SHARED specifier is also required. In order to have all messages in the file appear in sequential order, the USEROPEN specifier references a locally written routine that positions an existing shared, relative file to its "end". The IOSTAT specifier is

used to check the status of the open.

Initially, the file type was declared to be 'UNKNOWN', on the assumption that either the existing file would be found, or a new file would be opened if needed. However, the USEROPEN routine requires an existing file. The first way around this was to have the command procedure that started MSD first run an initializing program that opened new copies of all the files, using all the same specifiers except the USEROPEN routine. This worked fine for a while, but resulted in extreme proliferation of files under some circumstances, as when testing drivers at 1:00AM and having to boot the machine several times. The next solution was to change the file type to OLD, and delete the initializing program. As the weeks went by and the files got larger, the time required for PRTLOG to find a subset of information increased. The present solution is a command procedure that stops MSD once a week, purges its files, renames the "current" files to "old" files, and restarts MSD. If the first attempt to open a file fails, MSD tries again using a file type of 'NEW' and not using the USEROPEN procedure. Only if the open finally succeeds is the unit marked available to receive messages.

5.0 CONDITION HANDLER

The VMS manuals have a great deal of information scattered throughout them about writing a condition handler. In particular, the Run Time Library reference manual devotes an entire chapter to the subject. In the Version 3 manuals, this was Chapter 6; it is Chapter 7 in the Version 4 manuals. The Message Dispatcher's condition handler is a close copy of the sample in section 6.2.5, "Logging Error Messages to a File" (Version 3). Section 7.2.5 in the Version 4 manuals describes this routine, but does not give the code. Once the condition handler has been declared, the program can use the services LIB\$SIGNAL and LIB\$STOP for errors which it detects. The user's condition handler is also automatically activated first for any system-detected errors.

The condition handler uses the system service SYS\$PUTMSG. The second argument to this service is an optional action routine, which must be a function. After SYS\$PUTMSG has translated the status code and formatted the message, control is transferred to this routine, and the formatted text is passed as the first parameter. The suggested code for the action routine is to write the text to

a file, and return. In this application, some additional formatting is needed so that records in the error log file conform to records in the other log files, allowing the use of PRTLOG on the error file. If the function's return value is .TRUE., SYS\$PUTMSG will also write the message to SYS\$OUTPUT and SYS\$ERROR. If the return value is .FALSE., the error message will not be echoed.

When control is returned to the user-written condition handler, also a function, it must determine its return value, generally either SS\$_CONTINUE or SS\$_RESIGNAL. The manual suggests that user-written condition handlers always re-signal the error, so that a traceback is generated. This implies that a process will die if the error is re-signaled. Due to this program's requirements, this is not desirable, so MSD's condition handler returns the status SS\$_CONTINUE. If the error is truly catastrophic, the exit handler will eventually be invoked, and the program will be restarted.

The action routine comes in handy for MSD to log additional information about an error status. MSD, knowing the context of the error, creates its own text message and passes it to the logging function to be logged. For example, suppose some interaction with a color CRT has failed. MSD checks the status, and calls LIB\$SIGNAL to log the error. Since the status message may simply amount to "something bad happened", MSD then sends a line of text to the logging routine, identifying the CRT name and unit number and mentioning the context, for example, could not assign, or could not read.

One problem with the condition handler was with one of the required parameters. A user-written condition handler has two required parameters, the signal arguments vector and the mechanism arguments vector. The references are somewhat misleading about the standard format of the signal arguments vector (SIGARGS). The first longword in this array is the number of additional longwords in the vector. This is followed by one or more message sequences, each consisting of a message code, the number of FAO arguments required, if any, and the FAO arguments. Following all the message sequences, the PC and PSL are appended by system, and these two longwords are included in the count in the first longword. Some messages require one or both of these items, some ignore them. In some

cases, an attempt is made by SYS\$PUTMSG to interpret these two items as additional messages. All SIGARGS vectors do indeed have this format, but system, RMS, and system exception messages all interpret the vector using different formats. System messages expect a message code only. RMS messages expect not only a message code, but an additional RMS status code. System exception messages expect a message code and FAO arguments, but no FAO count. In each of these cases, any FAO arguments that are present, as well as the PC and PSL, will be treated as chained messages. Internal errors detected by the Message Dispatcher are almost always either system or RMS messages. The program now calls a locally written MACRO routine that adjusts the "number of additional longwords" parameter depending on the status type.

Although this condition handler always continues, rather than re-signalling, there are some cases when continuing is not allowed. If the Message Dispatcher has checked for the error, it uses LIB\$SIGNAL, on the assumption that the program can still function. However, if the system is allowed to trap the error, a call to LIB\$STOP may result, and the condition handler continuation status is not allowed. In most cases, this will simply trigger the exit handler and the program will be restarted. However, if this occurs in the exit handler, the program exits at that point without having started a new copy. One example of such an error is a fatal error during a write to one of the files, while not using the IOSTAT or ERR keywords. The only solution to this problem is to have the exit handler trap all of its own errors. While this routine is doing more error checking than it once did, it is probable that all surprise errors have not yet been anticipated.

6.0 EXIT HANDLER

When the exit handler is triggered, it outputs a termination message to all the available log files and hard-copy units, and starts a new copy of the Message Dispatcher process, using SYS\$CREPRC. The Message Dispatcher name toggles between MSD1 and MSD2, since multiple processes with the same name are not allowed, and the created copy uses whichever name is not the name of the current process. Since doing a force exit on the current MSD process triggers the exit handler, this provides a

convenient way to bring up a new version of MSD.

The references on writing an exit handler are scattered throughout the manuals and are often vague or ambiguous. An exit handler is declared by using the system service SYS\$DCLEXH with an exit handler control block of at least 4 longwords. The first longword of the control block is a forward link, for VMS usage. The second longword is the address of the exit handler. The third specifies the number of additional arguments in the control block. The fourth is the address of the location where the system will fill in the reason for the exit, followed by any additional arguments to send to the exit handler. The fourth item is required, but so far we have found no useful information deposited as the "reason for exit".

One of the most irritating bugs was the question of whether the log files were still open or not, once the exit handler was invoked. At first, the exit handler was writing the termination message to some FORXXX.DAT files instead of the real log files. If an INQUIRE statement was used, the real log files were identified as open, and the writes would then go to the correct files. If the INQUIRE statement was removed, the files were again not found. We decided to re-open the log files in the exit handler just to be safe. Recently, another program was written that accidentally did not re-open its files in its exit handler. Since it worked correctly, we removed the file opens from MSD's exit handler, and it is now working correctly. The only explanation suggested for this anomaly is a minor release of VMS.

A caution on the use of the system service SYS\$CREPRC: all of its optional arguments are not all optional. If the INPUT and OUTPUT arguments are not supplied, for example, the system service completes successfully, but the process is not actually created.

Since the program restarts itself when exiting, we must be careful to avoid a crash loop. In case of catastrophic failure early in the program, the exit handler is not declared until after most of the initial setup and before the attention ASTs are set up. Only once has a crash loop occurred, during a period when a number of other system errors were happening. The program repeatedly failed when trying to set up the AST

on one of the CRT mailboxes. The error message that was being logged was RMS-?-xxxx, and suggested a hardware problem at the console, a surprising error to get from the RMS facility. The question mark indicates that the system is for some reason not quite sure if this is the correct message, and in fact no hardware error was found. Without making any changes, the program was recompiled, the executable image was replaced, and the problem disappeared.

As previously discussed, as much error trapping as possible should be done in the exit handler, or the program may fail. Additional unexpected errors show up as time goes on, usually file-related. Twice, when the exit handler needed to extend a log file in order to write the termination message, it has failed because the disk was full. The solution currently being tested is to use the IOSTAT keyword on all file writes, give MSD its own quota, and mark the files as unavailable for messages once overdraft is exhausted.

7.0 ADDITIONAL PROBLEMS

Several additional problems encountered with the program have been FORTRAN-related, the most notable of these being that FORTRAN is not AST re-entrant. Normally, an AST routine must be declared external before it can be set. FORTRAN doesn't allow a routine to be EXTERNAL to itself, considering that to be recursion. MSD employs two different methods to get around this problem.

The driver AST is given the AST address as the AST parameter when the attention AST is first set by the main program. All subsequent ASTs for this routine are set using this parameter as both the AST address and its parameter.

```
STATUS = SYS$QIO( , %VAL(DR_CHAN),
%VAL(IO$_READVBLK), , DR_AST,
DR_AST,
%REF(DR_BUFFER), %VAL(30), , , )

SUBROUTINE DR_AST (ADDR)

.

.

.

STATUS = SYS$QIO( , %VAL(DR_CHAN),
%VAL(IO$_READVBLK), , DR_AST,
DR_AST,
%REF(DR_BUFFER), %VAL(30), , , )
```

The other AST routine (PROCESS MSG) requires some additional information, namely the "unit number" of the triggering mailbox, the mailbox channel, and the mailbox IO SB. This information is packed into a four longword array (IDENT) and sent as the AST parameter. However, the AST is actually set on a dummy routine (DO PROCESS) which does nothing but call the real processing routine with the array of useful information. The real message processing routine can then declare DO_PROCESS EXTERNAL and reset the AST.

```
STATUS = SYS$QIO( , %VAL(IDENT(2)),
%VAL(IO$ READVBLK), IDENT(3),
DO_PROCESS, IDENT,
%REF(BUFFER), %VAL(150), , , )
```

```
SUBROUTINE DO_PROCESS (IDENT)
CALL PROCESS_MSG (IDENT)
```

```
SUBROUTINE PROCESS_MSG (IDENT)
EXTERNAL DO_PROCESS
```

```
.
```

```
.
```

```
.
```

```
STATUS = SYS$QIO( , %VAL(IDENT(2)),
%VAL(IO$ READVBLK), IDENT(3),
DO_PROCESS, IDENT,
%REF(BUFFER), %VAL(150), , , )
```

Other FORTRAN-related problems have been minor. Some messages were truncated to 80 characters, which was fixed by opening unit 6 in transparent image initialization using RECL=132. FORTRAN also includes carriage returns and line feeds in some of its error messages. These non-printing characters are removed from the message text during processing.

The LA50s used for the system and accelerator hard-copy units have a "ready" state that periodically changes to "unready" for unknown reasons, and must be manually reset. If MSD continues to write to the device while it is not "ready", the buffer will eventually fill up and MSD will enter MWAIT state. The program now uses a routine which checks the state of the device before attempting the write. The message is not written if the device is not "ready". The next time the device is "ready", a message warning of possible lost messages is written.

Some quota-related problems have also arisen. The buffered I/O limit (BIOLM) defaults to 6. Using this limit, the Message Dispatcher sometimes ran very slowly, particularly when processing a large number of messages at once. This was usually apparent in processing a traceback, when the dying process could take several seconds to disappear. This limit has been boosted to 40, and is no longer a problem.

A more serious problem was related to the AST limit (ASTLM). MSD would start up normally, and then become stuck in MWAIT state. Oddly enough, the program would work correctly when run under the debugger. It turns out that ASTs are not returned to the available pool until the AST routine exits. Since this program resets its ASTs within the AST routines, a "spare" AST is needed. MSD thus needs 2 ASTs for each console, 1 for the driver mailbox, 1 for GEN DEL MBX, and 1 spare. The original configuration, with only one console, used 4 ASTs actively. The default for ASTLM is 6, leaving 2 spares. When the second console was added, MSD needed 6 active ASTs plus the spare, leaving it one short. Once the first message had been processed and the AST needed to be reset, it would have to wait for another AST to become available, something which could never occur. When run under debug, however, it was given the user's quotas, which happened to include 9 ASTs. The program is now using an AST limit of 20, which will allow it to run with the maximum configuration of 8 consoles with no more alterations.

8.0 ACKNOWLEDGEMENTS

We would like to thank Eric Bjorklund for his contributions to this project.

VAXCLUSTER ON A BUDGET

Rochelle Lauer
Yale University
Physics Department
P. O. Box 6666
New Haven, Connecticut 06511

ABSTRACT

Upgrading to VAXcluster hardware can be prohibitively expensive in low budget environments. This paper presents a user's experience with the minimum hardware configuration necessary to run a VAXcluster under VMS version 4.

In particular, the paper highlights functionality which can be achieved without the HSC50 controller. Experience with common system disk, shared queues, shared system and user files, implemented through MSCP served, UDA50 and MASSBUS, locally attached disks, is presented.

1. INTRODUCTION

The task of providing high technology computer facilities to scientific researchers with almost non-existent hardware/software budgets, poses a unique challenge. Each hardware purchase is carefully considered to meet the following criteria.

- Provides maximum functionality for the given type of equipment.
- Supports the current state-of-the-art software systems.
- Can become part of a migration path to emerging technologies, to be acquired as the budget permits.

The Yale High Energy Computer Facility exemplifies this need to maximize available funding. The facility supports research in high energy, experimental and theoretical physics. Yearly funding is minimal, allowing at most, integration of one significant piece of hardware and supporting software.

We started with a core system which could be expanded as technology and funding dictated.

This core, consisting of a VAX 11/750 system (CPU, system disk, data disk, tape drives, printer) and supporting software (VAX/VMS, FORTRAN compiler, advanced editor, text processor) provided sufficient functionality for one year. Our increasing need for CPU power became critical just as DEC announced VAXcluster availability. VAXcluster would suit our needs, but not our budget.

2. THE PLAN

Our major hardware acquisition was to be a CPU (VAX 2), with some as yet, undefined network connection to the existing system. Funding for peripheral equipment was limited to the system disk necessary to boot and maintain VMS. Therefore, the network connection would be instrumental in providing access to the existing (VAX 1) tape drives, printer, word processing devices and data disk.

The straightforward application of DECnet would suffice, if we were willing to live with the following constraints.

- A set of command procedures would be needed to provide user transparent access to devices and files which were not physically connected to the host VAX.

- Without tape drives on VAX 2, VAX 1 would become an intermediary for VAX 2 backups.

Disk_{VAX 2} → Saveset_{VAX 1} → tape_{VAX 1}

Cumbersome at best !

Although viable, the DECnet solution provided no migration path to our desired VAXcluster goal.

As knowledge of VAXcluster became available, the MSCP server emerged as the answer to our problems. Could we have total cluster functionality without the (unaffordable)HSC50? It appeared that with MSCP served disks we could achieve :

- Shared print and batch queues.
- Generic queues , spanning both CPU's.
- Direct backup of VAX 2 system disk to VAX 1 tape drive.
- Shared site-specific system files.
- Common mail files.

We found no-one to contradict or confirm our assessment. We were told that in principal the idea was sound, in reality, it had never been tried. We decided to put it to the test.

3. THE CONFIGURATION

Hardware

The new 750 purchase, completing the VAXcluster, consisted of

- 750 CPU
- RA81 disk
- Cluster upgrade package (2 CI750's; with cabinet, star coupler, cables)

The total configuration is pictured in figure 3-1.

Software

With the spirit of full VAXcluster functionality in mind, system software procedures were designed for maximum shareability. Ignoring the current

restrictions, VMS version 4 was implemented assuming the presence of an HSC50, compromising only when the MSCP server proved inadequate.

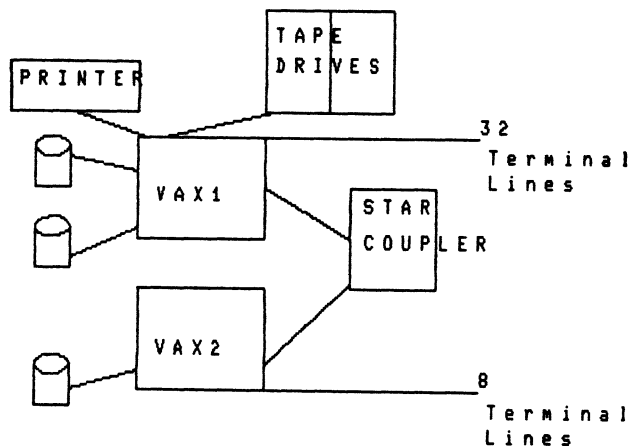


Figure 3-1: VAXcluster configuration without HSC50

In this manner, future hardware upgrade would have minimum impact on the running system.

The following steps were taken to ensure smooth transition to the homogeneous VAXcluster goal.

- A set of common system procedures(systartup.com, sylogin.com, startq.com) were designed for execution on any node in the cluster.
- A common system directory structure was created on each node-specific system disk. Although an MSCP-served disk is not supported as a bootable cluster-common system disk, we were preparing for future transition to the HSC50 hardware.
- (Shared) common files were created whenever possible.

Allowable shared files.

- * Queue file
- * Mail database

Files not to be shared with MSCP disks.

- * SYSUAF.DAT
- * NETUAF.DAT
- * Files in SYS\$SYSTEM

Files to experiment with

* Site-specific libraries/vendor software.

4. THE IMPLEMENTATION

We adopted a step-wise, conservative approach to VMS version 4 installation.

1. The new VAX hardware was installed, while continuing to run VMS 3.7.
2. Assured that the hardware was functioning properly, we upgraded to version 4.0, only after making a backup copy of version 3.7 on an alternate system root. Our production system continued to run version 3.7, with a CI-DECnet connection between CPU's. We now had two versions of VMS (3.7 and 4.0-4.1) on each system disk.
3. We tested the experimental non-HSC50, VMS version 4.0 cluster at our convenience.
4. Satisfied that the cluster would actually run, it was transferred from experimental to production status.

Common System Startup Procedures

The system startup procedures are shown in tables I-1 through I-4. Common procedures are used whenever possible. Node-specific procedures are in general, named nodefunction.com; where node is the DECnet node name, and function indicates the procedure's purpose. In this manner, node-specific procedures can be called generically from common procedures.

Queues

A single queue file on SITE\$SYSDSK is used to maintain all queues. This queue file is resident on a VAX1 local disk, accessible to VAX2 through the MSCP server.

The single queue file mechanism provides transparent sharing of local devices. With the definition of cluster wide print queues for the high speed printer and 3 word processing devices attached to VAX1, the devices become available to VAX2 users. A print command executed on either node, is automatically routed to the proper device.

The single shared queue file is also the vehicle for implementing generic batch queues across the cluster. Generic batch queues provide load balancing, with the queue manager directing traffic to the least loaded CPU.

The consequences of having a single MSCP-served queue file are discussed in section 5.

Shared Files

Single shared files provide the data integrity required in a cluster environment. The difficulty in maintaining multiple copies of the same files, unavoidably causes the files to diverge. Updates applied to one file do not always get merged into the other. After a remarkably short time period, the files are noticeably different. We therefore aimed for maximum file sharing.

On the system level, we started with two shared files, the queue file, and the mail database. VMS documentation warned of sharing system files through MSCP served disks, and questions on MSCP performance kept us from implementing a single set of site-specific software files.

Encouraged by the successful running of the cluster, we decided to implement a shared site-specific library directory. As seen in

table I-1 a directory on the VAX1 disk is designated as our site-specific software directory (SITE\$SYSDSK:[UTILITY.]). Files in this area are accessed constantly by users on both nodes. A single copy of the screen editor, text processor, graphics package, mathematical libraries, are resident on the MSCP served disk. This consolidation freed us from updating multiple copies of software, and released a considerable amount of disk space as well. However, the more important consideration was that it work, and it did ! The non-HSC50 cluster was coming very close to the real thing.

5. Success with Restrictions

The software implementation of VAXclustering (i.e. through the MSCP server) was highly successful. However, the following restrictions proved serious enough to justify the purchase of an HSC50 controller.

- Tape drives are accessible only to the local node. MSCP served disks, transparently accessible to each node, solve the backup problem. However, user tape jobs must execute on the "tape drive" node, implying that users must either log onto the required node, or submit jobs to a specific batch queue. The former option was, at times, inconvenient, the latter defeats the purpose of generic, cluster wide batch queues.

- Our experience has shown that maintenance of two system disks (two complete sets of VMS system files) in a consistent manner requires more than twice the time of maintaining a single set. For example:

- * Updates to VMS must be done twice, thereby providing you with two chances to do things wrong. In reality, you will probably make different mistakes with each installation.

- * Changes in command procedures must be copied to each node. If not copied immediately (one forgets, gets interrupted, or the node is down), time is lost in ironing out inconsistencies. More serious consequences result if noticeable differences cause confusion among users.

- The most serious restriction is the dependency of VAX2 on VAX1. Common files on an MSCP-served local VAX1 disk (queue file, mail file, site-specific software library) require that VAX1 be up, if VAX2 is to be operational. Therefore, in this configuration, a single node controls the uptime for the whole cluster.

To alleviate the problem, one could restrict the use of common files. However, such a solution would preclude the use of generic queues, and would also compound the consistency problem caused by multiple copies of the same file. We aimed for maximum commonality, and learned to live with the added cluster downtime.

6. Adding the HSC50

With the successful implementation of a non-HSC50 VAXcluster, we achieved our major goal. In addition, the following configuration features prepared us for eventual migration to an HSC50 based VAXcluster.

- Common system disk structure.
- Node-specific procedures, called from common procedures.
- Shared files.
- Cluster wide queues.

We were surprised and pleased when, as planned, the installation of an HSC50 controller caused little changes to the software structure. In general, all procedures remained intact. Changes were limited to the actual names of physical devices.

The system device on each node was already structured as a common system disk. One of these disks became the cluster wide, bootable, system disk with the addition of a node specific root using MAKEROOT. We deleted all system files from the other disk, leaving us with a single (therefore consistent) system area for the cluster.

The cluster wide system disk is at this time, a single point of failure. The addition of a new disk, and a future version of VMS, will allow us to volume shadow this disk, thereby boosting the uptime and integrity of the cluster.

7. Conclusion

The MSCP server can be used to provide VAXcluster functionality without HSC50 hardware, thus maximizing the availability of peripherals and files in a dual CPU VAX configuration. Such a system can perform most, but not all tasks of an HSC50 based cluster.

The demonstrated advantages of shared files and generic cluster wide queues, combined with the ease of migration to HSC50 hardware, provide more than adequate justification for implementing VAXclusters in this manner.

Table I-1: SITE-SPECIFIC STARTUP PROCEDURE

```
$ RU sys$system:sysgen
CONNECT CNAO/NOADAPTER
MSCP
EXIT
$ this_node = f$getsysi("NODENAME")
$ define/system YALE$NODE 'this_node
$ disk_proc= "@sys$manager:" + this_node + "dsk" + ".com"
$ term_proc = "@sys$manager:" + this_node + "trm" + ".com"
$ node_proc= "@sys$manager:" + this_node + ".com"
$!
$ 'disk_proc
$ 'node_proc
$ 'term_proc
$!
$ define/system/exec /tran=(concealed) -
    SITE$SYSDEVICE YALPH1$DUA0:
$ define/system/exec -
    SITE$SYSDSK SITE$SYSDEVICE:[v4common.]
$ define/system/exec vmsmail -
    site$sysdisk:[sysex]vmsmail.dat
$!
$ define/system/exec sys$sylogin site$sysdisk:[sysmgr]sylogin.com
$ define/system/exec MAIL$SYSTEM_FLAGS 3
$!
$ define/system/exec -
    site$utility site$sysdevice:[utility.]
$!
$!
$ @sys$manager:startq
$ exit
$! end of systartup.com
```

EXPLANATION:

The procedure is common to all nodes

The CNAO device allows DECnet connection over the CI bus.

Loading the MSCP server permits local disks to be shared across the cluster.

Node-specific procedures are derived from the node name, and then called.

The definition of SITE\$SYSDISK provides a common area for storing site-specific cluster wide software. The logical name was originally defined to be node-specific, and two copies of software were maintained. As we gained confidence in the cluster, SITE\$SYSDISK was redefined to be a common area (as seen here) on a VAX1 local disk. The software was automatically available to VAX2, as the VAX1 disk was MSCP served to VAX2.

Table I-2: START QUEUES

```

$ start/queue/manager SITE$SYSDSK:[sysexe]jbcsysque.dat
$!
$ YALPH1_start = "/NOSTART "
$ YALPH2_start = "/NOSTART "
$ this_node = f$getsyi("NODENAME")
$ 'this_node'_start = "/START"
$!
$! ALL devices are on YALPH1
$ if this_node .nes. "YALPH1" then goto init_queues
$ Set device/spooled=sys$print LCAO
$!
$INIT_QUEUES:
$ Initialize /queue/enable_generic_printing/on=YALPH1::LCAO: ~
  'YALPH1_start' YALPH1_LCAO
$ set queue/default=(FEED,FLAG=ONE,NOTRAILER,NOBURST) YALPH1_LCAO
$!
$ Initialize/queue/generic=(YALPH1_LCAO)/start sys$print
$ set queue/default=(FEED,FLAG=ONE,NOTRAILER,NOBURST) sys$print
$!
$! sys$fast only on local nodes
$ Initialize/queue/batch/on='this_node'::/job_limit=2/base_priority=4~
  /cpudefault=NONE/cpumaximum=0-00:15:00.00~
  /wsdefault=0/wsquota=200/wsextent=1000/start 'this_node'_fast
$ define/system sys$fast 'this_node'_FAST
$!
$ Initialize/queue/on=YALPH1::/batch/job_limit=2/base_priority=3~
  /cpudefault=NONE/cpumaximum=0-02:00:00.00~
  /wsdefault=0/wsquota=200/wsextent=1500 'YALPH1_start' YALPH1_batch
$ Initialize/queue/on=YALPH2::/batch/job_limit=2/base_priority=3~
  /cpudefault=NONE/cpumaximum=0-02:00:00.00~
  /wsdefault=0/wsquota=200/wsextent=1500 'YALPH2_start' ALPH2_batch
$ initialize/queue/batch/generic=(YALPH2_BATCH,YALPH1_batch)/START ~
  SYS$BATCH
$! end startq.com

```

EXPLANATION:

The procedure is common to all nodes.

The queue manager is started using a common queue file.

A generic cluster wide print queue is defined for the single VAX1 printer.

A local (fast) batch queue is defined for each node. Short jobs remain on the VAX where submitted.

A generic shared batch queue is defined across the cluster. Long jobs will be executed on the least loaded CPU.

Table I-3: NODE SPECIFIC DISK PROCEDURE

YALPH1DSK.COM

```
$ set device/served YALPH1$DRAO
$ set device/served YALPH1$DUAO
$ set volume/rebuild yalph1$dua0
$ mount/system/noassist YALPH1$DRAO: YALDATADSK

$ mount/system/noassist YALPH2$DUA1: YALPHY1
$!
$ Define/system/exec/tran=(terminal,concealed) User1$disk "YALPH1$DUAO:"
$ define/system/exec/tran=(terminal,concealed) data$disk "YALPH1$DRAO:"
$ Define/system/exec/tran=(terminal,concealed) User2$disk "YALPH2$DUA1:"
$!
$ exit
$!
$!end yalph1dsk.com
```

EXPLANATION:

For each node in the cluster a procedure named nodeDSK.COM is maintained in the common system area.

The disks local to this node are served to the other node(s).

Table I-4: NODE SPECIFIC PROCEDURE

YALPH1.COM

```
$ @sys$sysdevice:[ingres]ingresins.com
$ exit
$!
$! end yalph1.com
```

EXPLANATION:

For each node in the cluster a procedure named node.COM is maintained in the common system area. All node-specific software/hardware is installed/loaded through this procedure.

At our site, the only node-specific feature is the installation of the the above software system on VAX1. The complementary VAX2 procedure (YALPH2.COM) does nothing.

RUN-TIME LINKING OF SHAREABLE IMAGES USING
THE IMAGE ACTIVATE SYSTEM SERVICE

Clark Oliphint
2126 Wallace Ave
Aptos, CA 95003
(408)662-0751

Run-time linking of modules is used in the Second Generation Comprehensive Helicopter Analysis System (2GCHAS). Modules are Fortran 77 subroutines. One module calls another using the CALL statement. A command procedure substitutes a Module Caller for the called module. The Module Caller calls an Executive service which activates the called module.

INTRODUCTION

The Second Generation Comprehensive Helicopter Analysis System (2GCHAS) is being developed by the Army Aeroflightdynamics Directorate which is located at NASA Ames Research Center in Mountain View, California.

The purpose of 2GCHAS is to determine the flight characteristics of a helicopter from a physical description of the helicopter. Figure 1 shows the top-level inputs to and outputs from 2GCHAS. The various results of an analysis are selected by the input processing options and output options. The typical analysis does not produce all the results shown in Figure 1.

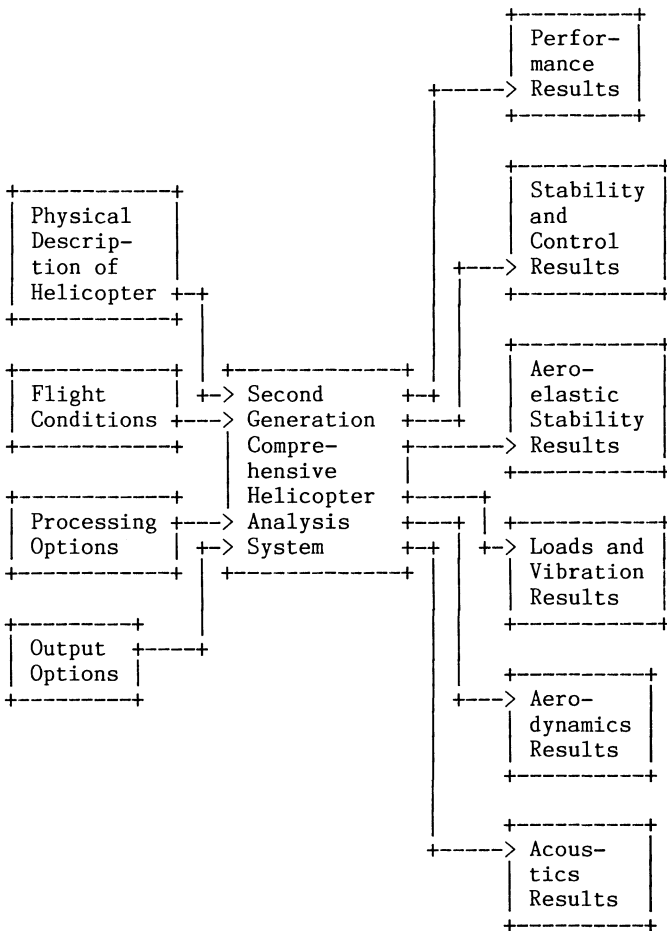


Figure 1. Purpose of 2GCHAS

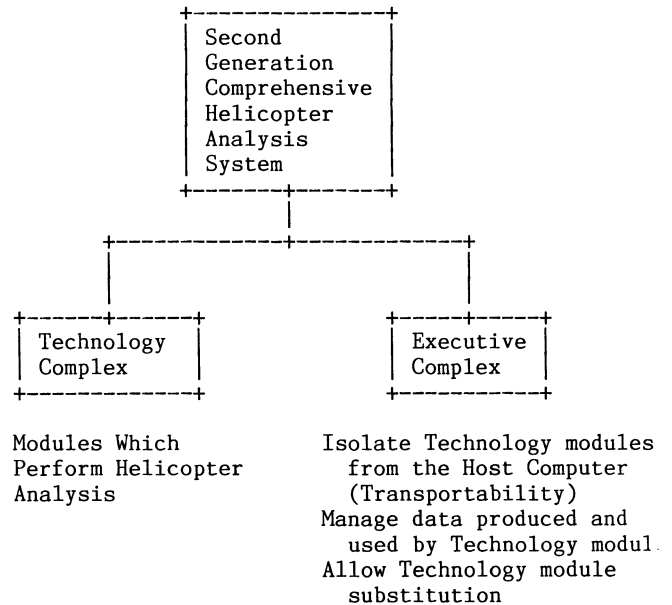


Figure 2. Top-Level Organization of 2GCHAS

Figure 2 shows the sub-division of 2GCHAS into two complexes. The Technology Complex consists of a large number of modules which perform the helicopter analysis. The Technology modules are written in standard Fortran 77. The Executive Complex provides services required by the Technology modules, and isolates the Technology modules from any particular host operating system. If 2GCHAS is transported to a different computer, only the Executive Complex is required to change.

The Executive Complex of 2GCHAS is being developed under contract by Computer Sciences Corporation (CSC). The CSC 2GCHAS project is located on-site at Ames Research Center.

STATUS OF 2GCHAS PROJECT

The 2GCHAS project is currently under development. The Executive Complex is being developed in a series of five builds. CSC delivered Build 2 of the Executive to the Army on December 19, 1985. Technology module development contracts will be awarded in early 1986. The Technology module development contractors will have remote access to Build 2 of the Executive. Build 3 of the Executive will be distributed to Technology module development contractors. Run-Time Linking was included in Build 1 of the Executive. It was tested by the Army as part of normal build testing. It was used by CSC in the development of Build 2 which has just been delivered to the Army. Run-Time Linking has not yet been used by Technology module development contractors.

THE PROBLEM

The 2GCHAS project required an alternative to the standard technique of linking all modules together for the following reasons:

1. There will be a large number of Technology modules in 2GCHAS. By design, new modules will be added to 2GCHAS throughout the life of the system as new helicopter analysis techniques are discovered.
2. A typical analysis uses relatively few of the available Technology modules. Available modules might supply alternative approximation methods or apply different flight simulation techniques or compute the various output results.
3. It is difficult to predict in advance which modules are required for an analysis. Modules are called as required by a combination of the processing options selected, the output results required, and the data describing the helicopter or the flight conditions.
4. Any module linked into an executable image uses resources even though it is never executed. For example, all modules in an executable image are assigned virtual memory when the image is run. Modules which are not executed thus reduce the amount of virtual memory quota remaining to contain data for the analysis.
5. Simultaneous development of new modules by multiple developers will continue through the life of the system. It is complicated to maintain and distribute a standard set of object modules to which the modules under development can be linked.

THE SOLUTION--RUN-TIME LINKING

In order that the Technology modules be transportable, a module call looks exactly like a subroutine call. Suppose, for example, Module A calls Module B. Figure 3 shows how the modules are linked using the standard linking procedure. Each

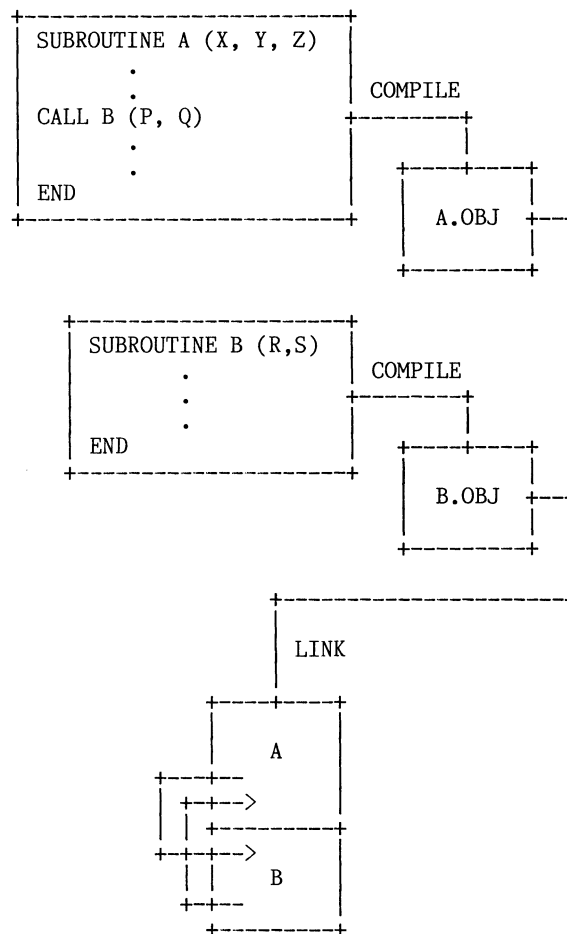


Figure 3. Standard Fortran Subroutine Linkage

module is compiled, producing an object file. The object files are then linked by the Linker into an executable image which can be executed using the RUN command. (There must be a main program linked with the subroutines, but it has been omitted here to show the module linkage process more clearly.)

Figure 4 gives an overview of the Run-Time Linking technique used in the 2GCHAS project. A command procedure, DEFMODULE (for Define Module), produces a source file for a module caller from the source file of the module. DEFMODULE then compiles both source files to produce object files of the module body and the module caller. In this example, Module B is defined first. DEFMODULE produces the source file for B Caller from the module B source file, then compiles both source files to produce object files for B Caller and the body of module B. The object file for B Caller is put in an object library containing the module callers for all defined modules, and the object file for the body of B is linked to produce a shareable image of B. When Module A is defined, DEFMODULE similarly produces object files for the body of A and for A Caller. The object file for A Caller is put in the module caller object library, and the object file for the body of A is linked with the library of module callers to produce a shareable image for Module A. Since Module A calls Module B, B Caller is linked into the shareable image for Module A.

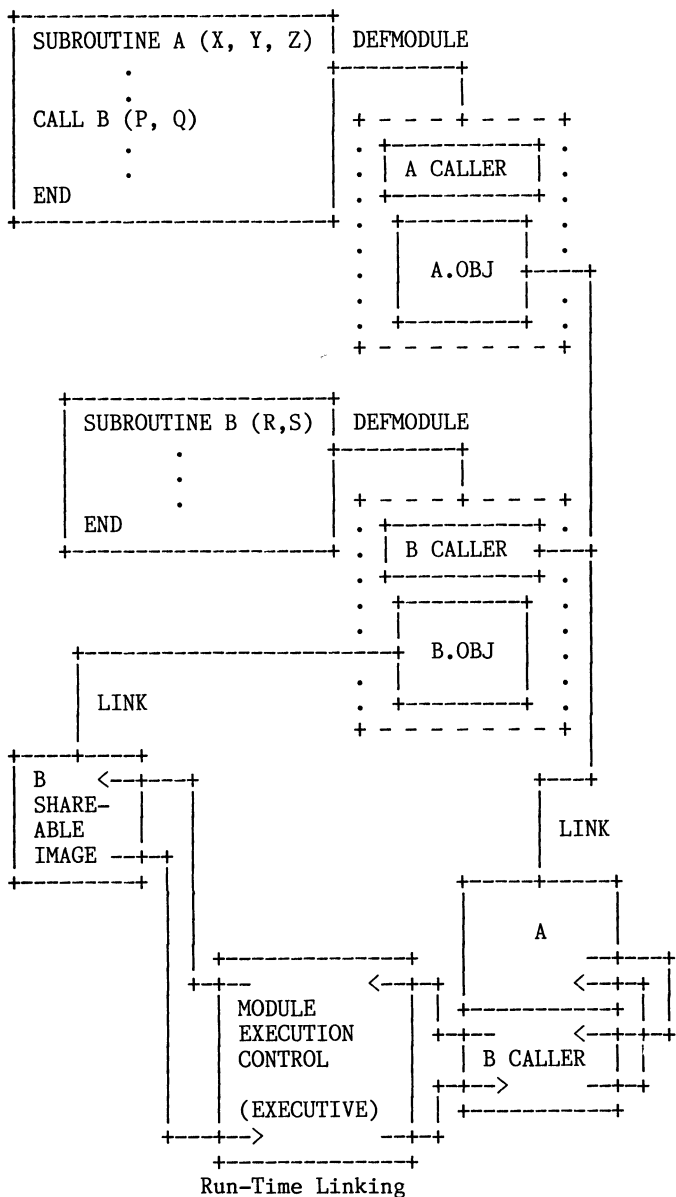


Figure 4. Overview of 2GCHAS Technology Module Run-Time Linking

When Module A executes the "CALL B" statement, it actually executes a subroutine call on B Caller, which has been linked into the image in place of Module B. The B Caller subroutine, which was created by DEFMODULE, calls an Executive service, Module Execution Control (XMEC) to activate the shareable image for Module B. When XMEC is called to activate a shareable image, it checks a list to see if the image is already activated. If the image is not activated, XMEC activates it by calling the Image Activate and Image Fix System Services. After XMEC activates the shareable image of Module B (or finds it already activated), it executes a subroutine call on Module B and passes the argument list from the subroutine call executed by Module A. When Module B completes its execution, it returns to XMEC, which returns to B Caller, which returns to Module A.

The activation of the shareable image of a module when the module is called as a subroutine is the Run-Time Linking technique used in 2GCHAS. From the point of view of the calling module, it has executed a standard subroutine call. From the point of view of the system, the module is assigned virtual memory and other resources only if it is executed.

XMEC uses the Image Activate and Image Fix System Services because at the time Build 1 was implemented, only Release 3 of VMS was available. Release 4 of VMS is now available. In Build 3 of the Executive, XMEC will use the Library Service LIB\$FIND_IMAGE_SYMBOL to activate shareable images.

TRADEOFF CONSIDERATIONS

The advantages of using Run-Time Linking are as follows:

1. Virtual memory and other resources are allocated to modules only if the modules are executed.
2. Enhancements to modules can be tested by module substitution at run time.
3. The option of linking a module directly using the standard Linker remains available with no change in the source module, since the source is standard Fortran 77.

The disadvantages of using Run-Time Linking are as follows:

1. Program execution time is increased. The first time a module is called, an elapsed time of about 0.2 seconds is required to activate the module. All subsequent calls on the module execute only slightly longer than a subroutine call because the LIB\$FIND_IMAGE_SYMBOL Library Service is executed instead of a subroutine call. If an analysis runs any significant amount of time, the overhead time required to activate the modules which perform the analysis can be ignored.
2. Fortran COMMON blocks can not be used to communicate between modules linked at Run-Time, although the subroutines which are contained within a single module may communicate with each other through COMMON blocks as usual. In 2GCHAS, the Executive contains data management services which provide a data structure intended to replace Fortran COMMON blocks in communicating between modules linked at run time.

SUMMARY

Run-Time Linking promises to be a useful technique for 2GCHAS because of the large number of modules which will eventually be in the system, and because, for a typical analysis, relatively few modules are executed. The additional execution time required for run-time linking will be insignificant compared to the execution time of an analysis.

The degree to which Run-Time Linking is successful will be determined when the technique is in general use by Technology module developers.

Robert C. Groman
 Woods Hole Oceanographic Institution
 Woods Hole, Massachusetts 02543

ABSTRACT

This paper reviews Woods Hole Oceanographic Institution's experiences in implementing a VAXcluster during the summer and fall of 1985. These experiences may help you prepare for, and avoid, the hurdles and problems that we encountered.

INTRODUCTION

The Woods Hole Oceanographic Institution (WHOI) is a private non-profit corporation (about 1000 employees) doing basic research and development in the field of oceanography. The Institution consists of five departments: biology, chemistry, geology and geophysics, ocean engineering, and physical oceanography. Researchers in these departments seek grants and contracts from government agencies and from private foundations. Institution personnel share common research tools such as the laboratory buildings, ocean-going ships and the central computing facility.

The central computing facility is operated by the Information Processing and Communications Laboratory (IPCL), a group of 22 Ocean Engineering Department employees. Researchers pay for the computing resources based on the not-for-profit, cost center, concept: the more computer time used during the year, the less costly the computer time. IPCL also provides computer expertise to the Institution via consulting services, applications programming, newsletters, documentation, seminars and classes.

The central facility consists of two VAX-11/780 systems. These systems provide time sharing and batch services for computer modeling, graphics, data management and data processing applications. IPCL also provides systems and operations support to a project-owned VAX-11/785 and two VAX-11/750's supporting the National Marine Fisheries Service, image processing and analysis, and the Institution's administrative and accounting needs, respectively. All of these VAX systems use the VMS operating system. Many researchers also use microVAX I or and microVAX II systems for their projects. The Institution currently has three microVAX I's, two microVAX II's and three VAXstation I's. There are a total of thirty-seven (37) disk drives (over 16 gigabytes) used by the larger systems, consisting of nineteen (19) RA81 456 megabyte drives, twelve (12) RP07 516 megabyte drives, two RM05 256 megabyte drives, two RP06 176 megabyte drives, one RM80 124 megabyte drive, and one RA60 205 megabyte drive.

We connected four of these VAX systems into a VAXcluster beginning in the summer of 1985. The four systems, one VAX-11/785, two VAX-11/780's and one VAX-11/750, each support a number of local disk drives and connect to an HSC50 disk controller with two RA81 disk drives (see Figure 1). Our experiences during this implementation may serve to aid others who are just beginning the clustering process.

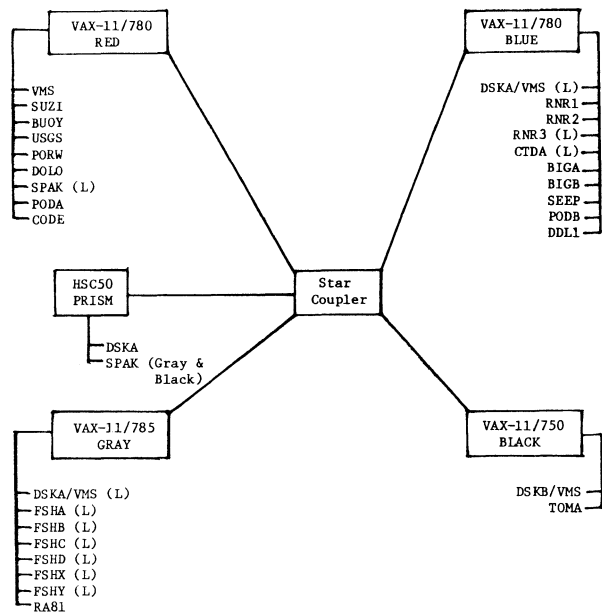
This paper begins with a brief review of VAXcluster concepts, including the differences between homogeneous and heterogeneous clusters. The next section discusses clustering problems which are likely to be seen by most new cluster sites. The last section discusses the unique problems we encountered due mainly to our heterogeneous cluster environment.

VAXCLUSTER CONCEPTS

A VAXcluster is an integrated organization of VAX/VMS computer systems that use a high speed 70 megabit per second communications channel for information transfer (1). The VAXcluster provides a single security and management domain which, under certain circumstances, takes less human resources to manage than individual VAX/VMS systems. A VAXcluster (also called a cluster) can be considered something more than a local area network of loosely coupled central processing units (cpu's). The machines in a VAXcluster can work closely together to share print and batch queue resources and share a common disk based file system. However, the cpu's can also exist independently of one another (unlike tightly coupled cpu's) if one or more of the cpu's fail. With appropriate hardware, end users are less susceptible to hardware component failures, including whole cpu's, within the cluster environment.

A homogeneous cluster is a VAXcluster where all software products are available to all cpu's on the cluster. (The VAXcluster manual prefers the statement that all known VAX images are the same on each cpu.) Also, logical names and all mass storage devices and queues (print and batch) are common to all cpu's on the cluster. You can log onto a homogeneous cluster and accomplish your

Figure 1
WHOI VAXcluster Configuration



(L) = local disk only

(node names) = disk accessed via cluster but access restricted to specific nodes

SPAK = lower cost, temporary storage space for large disk files.

work with equal ease no matter which cpu you are logged onto. In fact, there is only one user authorization file needed in a homogeneous cluster.

A heterogeneous cluster is a VAXcluster where one or more attributes of a cpu environment differentiate it from the other cpu's in the cluster. For example, not all cpu's may be able to run the Fortran compiler, you may not be authorized for every cpu in the cluster (perhaps one of the cpu's is dedicated to a single department), or certain mass storage devices (disk drives) may only be available to one cpu (perhaps for security reasons). The more differences there are, the more heterogeneous the cluster is. The more heterogeneous, the more human resources it will take to run.

COMMON CLUSTERING PROBLEMS

Our site upgraded from VMS version 3.7 to version 4.1. Although we were using the cluster interconnect wiring as a Decnet communications channel prior to VMS 4.1, we wanted to take advantage of the VAXcluster capabilities, especially the easier accessibility of disk files. However, we encountered a number of difficulties while implementing the VAXcluster. Many of these problems are likely to be encountered by other VAX sites since as one of our system programmers said (2), "Darn! All the stuff they say in the book is true." This

section reviews those problems, point by point, so that you may be better prepared for them yourself.

1. Disks that are mounted with the /SYSTEM qualifier that you will mount cluster wide must have unique volume labels. You can prepare for clustering your VAXes by taking care of relabeling your disk volumes in advance.
2. If you remove a cpu from the cluster and run it independently of the cluster, you must change the SCSYSTEMID and SCSNODE system generation (sysgen) parameters. The SCSYSTEMID is to the cluster what the Decnet node number is to Decnet and, in fact, SCSYSTEMID must match the Decnet node number if the cpu is clustered. SCSNODE is a text string similar to the DECnet node name. It is used in the construction of disk logical names that are mounted cluster wide. STARTNET attempts to force node names and IDs to be identical.
3. In addition to possible political problems in your organization, clustering forces you to consider the concept of quorum votes. You must assign votes to each cpu node in the cluster and possibly to one or more disk drives to improve cluster availability and to avoid cluster partitioning. Quorum votes are used by the cluster software to help determine how many systems are in the cluster, what should happen if one or more of the nodes should fail, and to ensure that each node in the cluster continues to work cooperatively with the other nodes. This is essential for such things as the file system integrity and the system manager's sanity. The reference manual (1) adequately discusses this topic but be sure to read it carefully.
4. System shutdowns are more complicated in a clustered environment. VMS provides for a smooth transition from an "n" cpu cluster to an "(n-1)" cpu cluster. DEC added code to the SHUTDOWN.COM procedure to accomplish this.

However there are other areas of concern. For example, in a cluster, taking down one cpu may not just effect the users on that cpu. Since the cpu can be acting as the MSCP server (i.e., allowing a non HSC50 disk to be cluster available) for local disks, "remote" users (i.e., users logged onto another cpu) can be drastically effected. You will be taking away access to their data. (Dual pathing disk drives takes on real significance in these situations.) VMS anticipated this situation by broadcasting REPLY/ALL messages to all users logged onto the cluster. However, we have found that many users are irritated by what they consider spurious shutdown messages from a "remote" cpu. We are working on a scheme to minimize these problems by careful use of the SET BROADCAST command. Using SET TERMINAL/NOBROADCAST is not recommended since this prevents all messages from

reaching your terminal.

There is another consequence of sudden removal of a cpu from the cluster besides the fact that a locally connected disk becomes unavailable to the other cpu's. If the cpu does not return to the cluster before the disk mount verification times out (a sysgen parameter), the disk must be manually dismounted and mounted. We decided to increase the timeout period to its largest value and handle lack of disk timeouts due to non-cluster related problems separately since these are rare. Be aware, however, that processes that are accessing these wayward disks wait until the disk returns before continuing.

5. AUTOGEN, the software that helps select sysgen parameters based on your system size, did not increase the intermediate and small request packet sysgen parameters (IRP's and SRP's) when we clustered. As a result, one of the cpu's was unable to boot in the cluster environment until these parameters were increased manually.
6. The format of the VMS PROCESSID changed. Any command procedures or other code that depends on the PROCESSID format may break.
7. At the present time, all cpu's in the cluster must be at the same VMS release level. This situation may change. We hope so since VMS upgrades will be more difficult in a cluster environment unless you take full advantage of the common system disk concept.
8. As with any enhanced software product, implementing clustering required more physical memory. Digital Equipment Corporation recommends that each cpu in a cluster have at least four megabytes of physical memory. If you are configuring a new VAX system do not short change yourself on this resource. Add a couple of extra megabytes for good measure.
9. Preparation and testing for the VAXcluster is more difficult since it involves at least two cpu's. Whenever the systems staff needed to test the new cluster, at least two groups of users were inconvenienced.

SITE SPECIFIC PROBLEMS

Our site also came up against a number of problems that are due to our heterogeneous environment. The two VAX-11/780's are general purpose machines that can be used by anyone in the Institution. Fees are charged according to the resources used, such as cpu time, connect time, input/output activity, disk storage, pages printed etc. The VAX-11/750 is run on behalf of the Ocean Engineering Department and can be used only by certain staff members. Beginning in 1986 these users will be charged only for cpu time, connect time, input/output activity and pages printed. The VAX-11/785 is run on behalf of the National Marine Fisheries Service. These

users pay a fixed fee (time and materials). Their costs are not based on computer resources used. On all systems, disk drives may be privately owned or may be part of the system. Also, each cpu has its own set of software products. The heterogeneous nature of the cpu's on the cluster create some interesting problems. This section reviews these problems, point by point.

1. Software installation is more complex than on a single cpu system. Whether you use a shared system disk or not, you will have to be concerned about which cpu has the right to access different licensed software. This applies to both DEC and non DEC products. It can be awkward to prevent access to a software package (especially a non DEC product) when the vendor assumes a homogeneous environment.
2. Each of our cpu's arrived at WHOI and came on-line at different times, spread over five years. Because we have so many users (over 1000), it was not possible to keep unique user identification codes (UIC's) among the systems prior to VMS release 4. Hence, to ensure the file system ownership integrity, the operations staff had to map the old UIC's into new, unique UIC's and then change all file ownerships accordingly (via the SET FILE command). The disk file owners did not experience significant inconveniences, but the operations staff spent quite a few (long) weekends accomplishing these changes on over thirty disk drives.
3. Our initial intention was to use a single, shared system disk. But due to performance considerations we decided to use two system disks, with the ability to fall back to one disk if necessary. We struggled with how to recover from a down, shared system disk since it will remove two cpu's from the cluster. It is also very difficult to maintain these two system disks with up to date software and procedures in the heterogeneous environment. Volume shadowing, the VMS feature that will automatically keep a multiple copy of a volume, will help resolve many of these problems.
4. We had hoped to easily export our VAX/VMS expertise to MicroVAX I and II system owners, using the clustered VMS system disk as the model. However, between the size of VMS and the command procedures that we wrote to simplify the operators' lives working in a clustered environment, the 31 megabyte disk on the microVAX is too small. Also, the default system root is SYS0 on microVAXes and it is not clear whether or not we can force the default somewhere else. (Even exporting our efforts to the 750 system had its difficulties. To boot a 750 from other than root 0 requires either a change to the ROM or a powerfail recover from the TU58 cassette.) While we still can export our expertise, we cannot define a "common system disk." Each microVAX is tailored for its own uses.

5. We had to delay implementation of the cluster scheme because the HSC50 lacked a critical microcode update. Field service was responsible for hardware microcode updates and they did not always receive complete information in a timely manner. In addition to the HSC50 update, our field service engineer alertly noted that the cpu's and RA81's also needed updates if they were to work with the newest version of the HSC50. Had he not spotted this potential problem we might have been in even worse shape. There has been a recent change in the microcode distribution scheme. Software Distribution will now handle microcode updates too.

Of course, an HSC50 down for repairs or updates affects more computer users, especially if it supports system disks as well as data disks. We either have to prepare the users for multiple system unavailability or implement a scheme to keep the systems up by moving the system disks to less used data disks. Neither the computer users nor the operations staff look forward to this situation.

6. Our site cannot take full advantage of the VAXcluster software feature due to accounting requirements. For example, we hoped to combine our so-called DSKA public disks (one exists on each cpu) into one, cluster available, DSKA disk. The DSKA disk is used by all computer users who do not own or have access to private disk space for online storage. Advantages of combining these disks include better disk space utilization and easier time for those computer users that use two or more cpu's. They do not have to maintain separate copies of their LOGIN.COM and other customizing files. However, since the public disk space on each system is not charged out in the same manner on each system, we cannot combine the disks. DSKA on the 11/750 and 11/785 VAXes are free while the 11/780 VAX users pay daily disk block charges. Using our existing accounting software would result in overcharging or undercharging for disk space, and the administration does not want any significant effort put into accounting software changes.

Another problem area is the batch and print queues. Since each cpu has its own rates for batch queues, making cluster wide batch queues becomes difficult. Similarly for cluster wide print queues, not only are the per page rates different, but each system uses its own preferred style of line printer paper. An unattended cluster wide print queue becomes quite unmanageable. This situation is further complicated by system ownership problems that prevent people from using the resources on a private cpu even if they were willing to pay.

7. We are attempting to work out a plan for adding additional public space (DSKA) to the cluster. The most direct solution is to use volume sets. However, volume sets have the

disadvantage that if one of the disks in the set fails, the volume set is unusable until the disk is repaired. If we create new logical names for the added disks, such as DSKB, DSKC, etc., many computer users will be inconvenienced since they routinely use DSKA:[...] to reference command files in their own as well as their associates' directories. Since computer users often have multiple login identities, it will be awkward for them to use more than one device name for the "public" disk. (Most users do not use SYS\$LOGIN when referring to their own login directory; they use DSKA:[...] explicitly.)

8. Finally, I was unable to clearly define the distinction between dual pathing/dual porting and cluster wide access to disk drives. Private disk owners did not see the value of redundant access to MSCP served disks until after the cluster was up. I should have done a better job explaining what happens when a cpu goes down making its local disks inaccessible to the other cpu's in the cluster.

CONCLUSION

We did receive positive comments from computer users once the cluster was implemented. People especially liked the easier, more direct access to the disk drives mounted cluster wide. Using the HSC50 or the MSCP software to access "remote" disk drives is faster and significantly less complicated than using DECnet software. The VAXcluster also offers growth potential both in cpu and disk resources. However, the VAXcluster can be complicated to implement and administer and certain cluster features, like shared batch and print queues, may not be available in a heterogeneous environment. We are learning more about the VAXcluster environment as we gain experience with it. With this new knowledge and with the help of new VMS features, we are addressing the remaining problems.

ACKNOWLEDGMENT

I would like to thank Debbie Parent for the care and speed with which she typed this paper.

The National Marine Fisheries Service (project number 17/71.00) and the Woods Hole Oceanographic Institution, Information Processing & Communications Laboratory (project numbers 5602 and 5610) provided support for this work.

Woods Hole Oceanographic Institution
Contribution Number 6241.

REFERENCES

1. Guide to VAXclusters, September 1984, Digital Equipment Corporation, Maynard, MA.
2. Sass, Warren J. Personal communication, November 4, 1985.

Forrest A. Kenney, Howard Clifford

Digital Equipment Corporation
8301 Professional Place
Landover, MD 20785

ABSTRACT

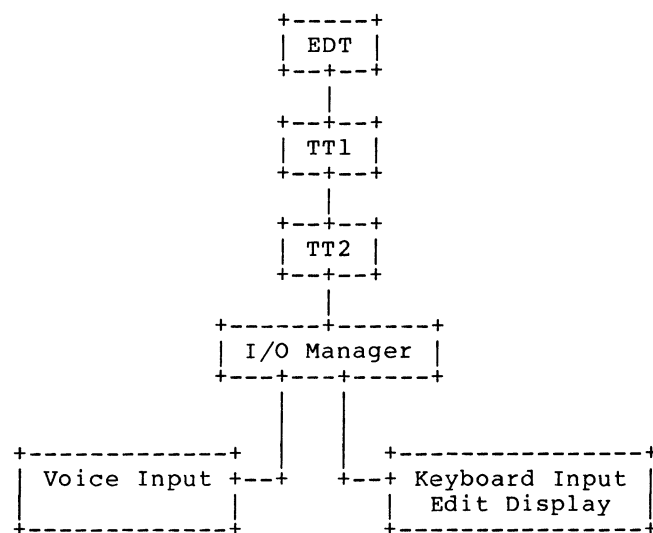
This paper describes a software device called a Pseudo Terminal. A Pseudo Terminal is a software device that looks like a terminal to the VMS operating system and its utilities. The Pseudo Terminal provides a mechanism which allows one process to intercept another process's terminal I/O requests. VMS mailboxes partially provide this capability but programs which use terminals as their input/output device cannot, without modification, use VMS mailboxes. Many VMS utilities, such as the EDT editor, will not run with a VMS mailbox as the input/output device. This paper discusses the possible approaches one can take when building a Pseudo Terminal. The paper discusses the merits and weaknesses of these approaches. Finally, the paper will outline some of the problems we encountered in building one Pseudo Terminal implementation.

Introduction

What are Pseudo Terminals and why do we need them? A Pseudo Terminal is a terminal that appears to the operating system as a physical terminal but is in actuality a collection of data structures and code which describe a terminal. A Virtual Terminal also appears to the operating system as a physical terminal, but its data structures are mapped onto a physical terminal. For many applications a Virtual Terminal is ideal. With a Virtual Terminal a user has control over the Virtual Terminal session, but he has no control over its input and output streams. Having answered what a Virtual Terminal and what a Pseudo Terminal are, let's address why we need a Pseudo Terminal. The best way to do this is to provide a couple of examples where a Pseudo Terminal is the only reasonable way to handle the problem.

In the first example assume we want to develop an application that will allow us to receive data from two input devices and direct this data into the EDT editor. This first example is one that was actually built by a colleague; at that time he did not have access to a Pseudo Terminal. Figure 1 shows the major hardware and software components used to implement this application. As Figure 1 illustrates, the application required four terminal ports. The basic scheme was to have an I/O manager allocate a port for the editor and start a process running the

FIGURE 1

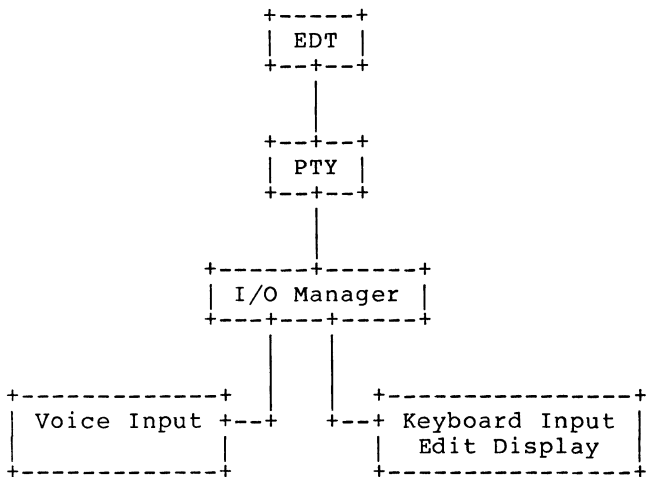


EDT editor. The EDT editor will read and write data using TT1 as shown in Figure 1. Ports TT1 and TT2 are connected in a manner that causes output from TT1 to be input for TT2, and output from TT2 to be input for TT1. The I/O manager then reads a single character at a time from TT2 and writes it to the users input terminal. The I/O manager also reads from both the keyboard input terminal and from the voice input terminal a single character at a time and writes it to TT2 which allows the EDT editor to read it as input data. This configuration worked but put a heavy I/O load on the system as every single character input to the editor required two

I/O operations to be performed by the I/O manager. In addition to the burden placed on the system from the extra I/O operations, it tied up four terminal ports which on most systems are a scarce resource.

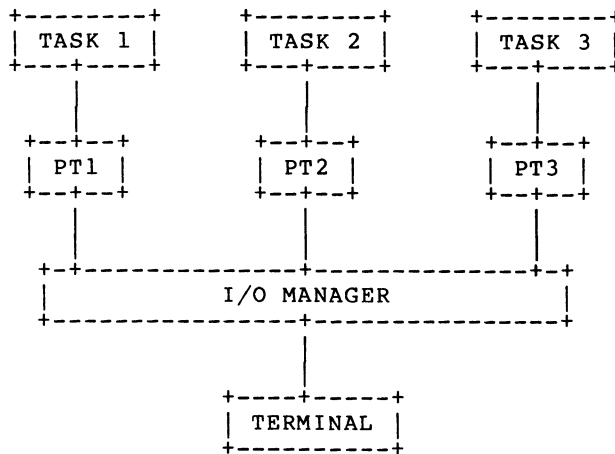
Having seen how the problem was solved without using a Pseudo Terminal, let's see how you could use a Pseudo Terminal to solve this same problem. The scheme is that one would build an I/O manager; the I/O manager would then create a Pseudo Terminal and start up a process running EDT. The EDT editor would read and write to the Pseudo Terminal as if it were a physical terminal. The I/O manager would assign I/O channels to the ports containing the terminal and the voice input device. The I/O manager would then be responsible for reading from both the input devices and writing data to the display device. The I/O manager would also be responsible for reading data from the Pseudo Terminal and writing data back to the Pseudo Terminal. Depending on how the Pseudo Terminal is implemented, this case could have substantially lower I/O operation overhead on the system. At the very least, this approach ties up substantially fewer terminal ports and also requires fewer hardware interrupts. Figure 2 shows the major hardware and software components required if you use a Pseudo Terminal (PTY).

FIGURE 2



In the second example, assume we wish to build what is essentially a poor man's VAXstation. That is, we want the ability to build a program that will allow us to display the output from multiple independent processes in windows on the screen of a single terminal. This second example can be partially accomplished without Pseudo Terminals through the use of VMS mailboxes. Figure 3 illustrates how you would accomplish this using Pseudo Terminals.

FIGURE 3



In this example the I/O manager creates as many Pseudo Terminals as are required and assigns an I/O channel to the output terminal. The I/O manager then determines which window on the terminal is currently active and directs the data from the appropriate Pseudo Terminal to the correct part of the physical terminal's display. The I/O manager also reads data from the physical terminal and returns this data to the appropriate Pseudo Terminal. There are many advantages when using Pseudo Terminals for this application instead of mailboxes; the list below summarizes some of them:

- o By using a Pseudo Terminal you do not have to worry about losing data because an I/O request was too small. This is not true if you use mailboxes.
- o Depending on how the Pseudo Terminal is implemented you may be able to use substantially fewer I/O requests than you would if you used a mailbox.
- o Pseudo Terminals support the use of out-of-band, CTRL/Y, or CTRL/C ASTs. This would be absolutely impossible if you used mailboxes.
- o The Pseudo Terminal supports all valid VMS terminal driver I/O functions codes and modifiers. Mailboxes support only a limited subset of these functions. Examples of functions not supported are IO\$_EXTEND, IO\$M_OUTBAND.
- o Finally, because a Pseudo Terminal looks exactly like a physical terminal, utilities that only work with a terminal still work as expected. A classic example is the EDT editor, which

will not work in screen mode unless it believes that it is hooked directly to a Digital supported terminal.

From these examples you can see that there are many cases where a Pseudo Terminal is a very desirable device to have. The examples cited above are by no means a comprehensive list. Some other possible uses could be a network terminal interface for a TCP/IP network, a tool that would allow you to log to a file or read from a file for a terminal application and many more.

Design Considerations

Why did we build a Pseudo Terminal and what were our primary design considerations? We were approached with a request to provide a device that would allow a window managing process to be built. The primary request was for a modification to the mailbox driver to make its device type terminal. The secondary request was that the I/O cost for the device be on the same order of magnitude as the mailbox driver. The system configuration had to support up to 20 users with 2 windows per user. The final restriction was that this device needed to be completed in less than three months. With these constraints in mind we began a study to determine what was the best approach.

The first alternative examined was to use the existing Pseudo Terminals that are available to users of DECNET. These terminals are RTTDRIVER and CTDRIVER. Both of these implement a device that has the functionality needed. Having determined that either of these devices met our needs, we studied the flow of information through DECNET to determine if the performance would be acceptable. A read request from the remote node goes through the following major steps:

1. A read request is issued to the remote terminal RTTx.
2. The request is sent to the NETACP through the NET device. The NETACP does processing on the request and queues it to the actual network communications device.
3. On the node performing the final terminal read, the read request is picked up from the network and fed into the NETACP.
4. The RTPAD reads the request from NETACP, disassembles the request, and issues the read to the local

terminal. When the read request is completed, the RTPAD process sends it to the NETx device.

5. The NETACP does necessary processing and queues the request to the network communication device.
6. The NETACP on the originating node receives the request and queues it to the Remote Terminal ACP (REMACP). REMACP completes the request and the results are returned to the user's remote process.

We determined that the amount of overhead involved would be unacceptable for our application, even if the request were being looped internally and not going out over the network. An additional disadvantage is that no published documentation exists for the protocol used between the RTTx device and the RTPAD process. For these reasons this approach was ruled out.

The second alternative studied was to make use of the existing VMS terminal class driver and to implement a Pseudo Port Device to talk to the class driver. By using this approach we would be freed from emulating the terminal driver's behavior.

We first studied the overhead associated with this approach. The I/O manager code would be required to perform single character reads (with echo disabled) from the terminal and pass the characters into the Pseudo Terminal. It would then need to read any characters to be output from the Pseudo Terminal and write them to the actual terminal. In the worst case this degrades into 4 I/O operations for every single character typed. If we assume that every I/O operation costs approximately 1 millisecond, that each person is typing at 20 characters a second, and that we have 10 typists, then roughly 80% of the available CPU cycles would be consumed by the I/O load.

Next we looked at the difficulty of building a port device. We could find no published documentation for the terminal driver port and class interface, this meant that there was a high risk of us not being able to build the code in the allocated time. For these reasons this approach was ruled out.

The final approach considered was to build a Pseudo Terminal device that met our need of ease of implementation and low system overhead. The device was originally conceived to be a cross between the VMS mailbox driver and the DECNET

RTTDIVER. That is, the device would pass messages that described the I/O to be performed, but would be much like the mailbox in terms of device access and ownership restrictions. By doing this, the number of I/O operations to the actual terminal would exactly match the number of I/O requests to the Pseudo Terminal. For record-oriented I/O operations this is a large savings over the previously discussed approach. For single character I/O requests this approach saves two I/O operations when compared to the previously discussed approach. Additionally, the device is simple enough that it could be implemented within the time frame allowed.

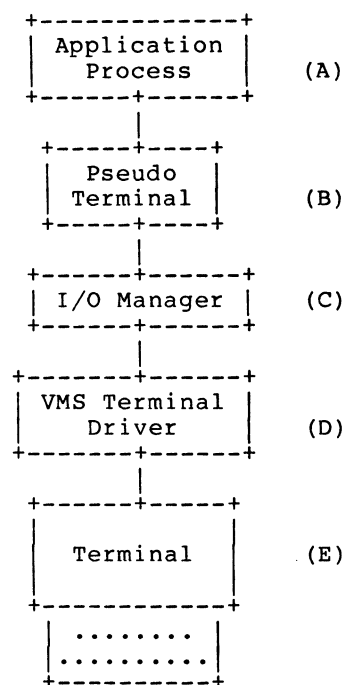
The Pseudo Terminal supports all the terminal driver function codes and modifiers documented in VAX/VMS I/O User's Guide version 4.0. The device specific function codes are listed below:

1. IO\$_GETIO passes to the I/O manager any pending terminal I/O requests.
2. IO\$_FINISHIO completes any terminal I/O request that requires completion.
3. IO\$_QUEUEAST causes the Pseudo Terminal to deliver any ASTs that may be associated with a specified character.
4. IO\$_SETATTN queues an attention AST that is delivered to the I/O manager whenever the application side performs a cancel operation.

Figure 4 shows a detailed layout of all software and hardware components involved in using our Pseudo Terminal.

To better understand the device let's look at a typical sequence of events that an I/O manager goes through to use our Pseudo Terminal. The example below assumes a relatively simple model. The model consists of an I/O manager, or in this case an I/O logging process that wants to capture, log to a file, and forward to a physical terminal, all the I/O requests issued by the EDT editor. The example correlates each step to the components in Figure 4.

FIGURE 4



1. The I/O logging process (C) creates a new Pseudo Terminal using the PT_CREATE routine. The PT_CREATE routine assigns a channel to the template Pseudo Terminal and receives a channel number for its Pseudo Terminal. The routine using the channel number then gets the device's name using SYS\$GETDVI. This step causes component (B) to be created.
2. The I/O logger (C) then uses the SYS\$CREPRC system service and starts up the EDT editor with its input and output assigned to the Pseudo Terminal whose name it received from PT_CREATE. This step causes component (A) to be created.
3. The I/O logger (C) reads the pending terminal I/O request from the Pseudo Terminal (B). This is done by using a SYS\$QIO(W) with a function code of IO\$_GETIO.
4. The I/O logger (C) then writes out the I/O request to a log file. It also performs the I/O request to the terminal driver (D).
5. If the request causes data to be output, it appears on the terminal (E).

6. If the request is one that can be completed without needing information returned to the application (A), we go back to step 3.
7. If the request is a read request or a sense operation for type-ahead count, then the I/O logger (C) has to wait until the actual terminal operation is completed. Once the terminal driver (D) completes the request then the I/O logger sends the results of the operation back to the EDT editor. This is accomplished by issuing a SYS\$QIO(W) with a function code of IO\$ FINISHIO to the Pseudo Terminal (B). The I/O logger (C) then goes back to step 3.
8. The above loop is be repeated until the editing session is completed. At this time the I/O logger deassigns its channel to the Pseudo Terminal and it ceases to exist.

Device Characteristics and Restrictions

Having decided what was the best approach for our problem, we derived a broad class of restrictions and assumptions about the device to keep the driver down to a manageable size while retaining acceptable functionality. This section outlines the device's major characteristics and restrictions as originally designed. The next section discusses some of the problems with these assumptions and our final solutions to the problems.

Primary Device Characteristics

<u>Characteristic</u>	<u>Explanation</u>
Device Independent	The device is set to have the following device independent characteristics DEV\$M_AVL, DEV\$M_CCL, DEV\$M_IDV, DEV\$M_ODV, DEV\$M_REC, DEV\$M_SHR, and DEV\$M_TRM.
Device Type	The terminal type is set to a VT100 with 80 columns and 24 lines.
Input Speed	The input speed is set to 9600 baud.
Output Speed	The output speed is set to 9600 baud.
LFfill & CRfill	The CRfill and LFfill values are set to 0.

The following limitations apply to the Pseudo Terminal:

- o The Pseudo Terminal when created has VT100 terminal characteristics. These characteristics may be altered by either the application process or the I/O manager process.
- o Pseudo Terminal does not provide for associated mailboxes.
- o Pseudo Terminal does not provide for command recall and command editing.
- o Pseudo Terminal does not provide type-ahead buffering. It does not accept input from the I/O manager unless there is a currently pending request from the corresponding application process.
- o Pseudo Terminal provides status returns on write requests which indicate the validity of the requests but do not indicate the actual completion of the write to the physical terminal.
- o The Pseudo Terminal device name is not the same as the device name for the standard VMS terminal driver. Processes must use the SYS\$INPUT and SYS\$OUTPUT devices created for them by the I/O manager. If other channels are used to communicate with the terminal, the Pseudo Terminal device name must be used when assigning those channels.
- o Pseudo Terminal does not provide timing of read requests. All timing requests are forwarded to the I/O manager.
- o The following function modifiers are masked out before the I/O request is forwarded to the I/O manager:
 - on read: IO\$M_DSABLMBX
 - on write: IO\$M_ENABLMBX
- o In an itemlist read, the item TRM\$M_TM_DSABLMBX is removed from the itemlist prior to forwarding the request to the I/O manager.
- o The Pseudo Terminal processes one request at a time for each Pseudo Terminal device. Subsequent requests are queued until the active request completes.

- o The Pseudo Terminal does not perform write breakthrough operations in the same manner as the terminal driver. Write breakthrough operations are queued immediately behind the currently active request for the Pseudo Terminal. They do not interrupt the active request but wait for it to complete.
- o The I/O manager, creator of the Pseudo Terminal, cannot request the IO\$_READxxx or IO\$_WRITExxx functions.
- o The I/O manager, creator of the Pseudo Terminal, cannot specify the following modifiers on a set request:
 1. IO\$_M_CTRLCAST
 2. IO\$_M_CTRLYAST
 3. IO\$_M_OUTBAND
 4. IO\$_M_OUTBAND ! IO\$_M_INCLUDE
 5. IO\$_M_OUTBAND ! IO\$_M_TT_ABORT
- o The I/O manager, creator of the Pseudo Terminal, cannot specify the IO\$_M_TYPEAHD CNT function modifier on a sense request.
- o The IO\$_GETIO, IO\$_FINISHIO, IO\$_SETATTN, and IO\$_QUEUEAST requests can only be issued by the I/O manager.

The last major implementation decision was to save system memory resources. We accomplished this by not using the VMS terminal driver Unit Control Block (UCB) extensions. We felt that this was a good decision as we needed only a relatively small number of additional fields beyond those used in the standard UCB.

Implementation Problems

In this section we will discuss some of the problems encountered in implementing our Pseudo Terminal and our final resolution of these problems. The first major problem we encountered was tied to our decision to use our own UCB extensions. We initially stubbed out the major portions of the driver so that we could test to make sure that the driver tables were correctly set up. At this point we also tested to make sure that the VMS utilities that query the driver tables would work. At this phase they all behaved as we anticipated. A few days

later when we had some code that filled in some of our device specific UCB fields these routines caused a system crash. After several crashes and much study of the micro-fiche, we discovered that several terminal driver UCB extension fields are not optional. UCB\$_TL_PHYUCB and UCB\$_TT_LOGUCB are used by the SYS\$GETDVI system service. These fields are pointers to the device's logical and physical UCBs. They must either contain valid pointers to the terminal devices logical and physical UCB or they must be 0. If either of these cases is not true the SYS\$GETDVI system service will fail when used on the Pseudo Terminal. Their purpose is to allow for the use of Virtual Terminals in version 4 of VMS. The other field that we discovered is necessary is the field UCB\$_TL_BRKTHRU. While failure to have and use this field will not cause the system to crash, it will cause the SYS\$BRKTHRU system service to allow or disallow the wrong types of broadcast messages to be sent to the Pseudo Terminal.

The second major problem that we encountered was that a process could not issue a SYS\$BRKTHRU to itself unless it had operator privilege. We traced these problems back to another of our design assumptions. The problem stems from how the SYS\$ASSIGN system service treats devices that are marked as shared. When a device is marked as being shared the SYS\$ASSIGN system does not fill in the device owner field (UCB\$_PID). It does this to allow multiple processes to access the device. The reason this prevents the SYS\$BRKTHRU system service from working is that this service checks to see if you are the owner of the device by matching your processes internal PID against the device owner's PID. If they don't match, it checks to see if the process attempting the breakthrough write has operator privilege.

The obvious solution to this problem is to make the device non-shareable. The problem with this direct approach is that the device I/O manager process then becomes the device owner. This would prevent connecting of another process to the Pseudo Terminal unless that process has SHARE privilege. Our solution was to mark the device as not being shared and have the driver modify the call frames so that the driver's CLONED UCB routine receives control when the SYS\$ASSIGN system service returns. The CLONED UCB routine is then able to make changes to the fields in the devices UCB that SYS\$ASSIGN fills incorrectly for our purposes. While this seems a rather crude solution to the problem it is a rather common approach throughout VMS.

The third major problem that we encountered was that the Pseudo Terminal would not allow spawned subprocesses. This was traced to an undocumented function modifier `IO$M_TT_PROCESS`. This modifier is used to `_set` which process currently has control of the device. The primary effect of this is to determine which process receives any ASTs to be delivered. The solution was to add support to the Pseudo Terminal set routines for this undocumented function modifier.

The last major problem that we encountered was one we had anticipated from the beginning. Some utilities take advantage of the fact that the terminal driver allows a write to proceed until a read starts. Since we perform all operations in the order they are received, there is a potential for conflict here. We have noticed problems with `SHOW CLUSTER/CONTINUOUS` and `PHONE` utilities. They work to varying degrees but not in an acceptable manner.

Other Pseudo Terminals

While working on our Pseudo Terminal and on this paper we have discovered the existence of a couple of other Pseudo Terminals for VMS. The below are brief discussions of these other Pseudo Terminals. The first one of these is supplied in the DEC/TEST Manager software version 2.0. It allows the DEC/TEST Manager to test interactive applications. If you can figure out the interface to it you should be able to use it for all your Pseudo Terminal needs. But be advised; it is only documented and supported for use with the DEC/TEST manager. The DEC/TEST Manager's Pseudo Terminal is implemented as two devices one being a Pseudo Port driver for the VMS terminal class driver. The other driver is used by the Pseudo Terminals I/O manager to send characters to and read characters from the Pseudo Port driver. For the intended application this is an optimal solution to the problem as it removes the I/O manager from having to emulate any of the VMS terminal driver functionality.

The second Pseudo Terminal that we are aware of is in the public domain. It is available over the ARPANET and is expected to be on the FALL 85 DECUS VAX SIG tape. The driver was originally written by Dale Moore at Carnegie Mellon University and modified to work on VMS V4.2 by Kevin Carosso of Hughes Aircraft Co. The Pseudo Terminal is implemented as two devices; one is a Pseudo Port driver for the VMS terminal class driver. The other driver is used by the Pseudo Terminal I/O manager to send characters to and read characters from the Pseudo Port

driver. The driver appears to have been written to provide a network terminal. For networks that are supporting machines whose terminal drivers do not or cannot provide similar functionality to the VMS terminal driver, this is a very good solution.

Both of these drivers take a similar approach to the problem of providing a Pseudo Terminal but vary in their implementation. Obvious questions to ask are why did we not use one of these devices and why did we take the approach we choose. The answer to the first part of the question is that at the time when we were looking for a solution to our specific problem we were unaware of the existence of either of these devices. The answer to the second question has already been discussed in the Design Considerations section. But to summarize for our application the overhead in VMS QIO mechanism for the most common case would have been unacceptable.

Future Plans

We feel that the device as it presently exists is an acceptable baseline. Some of the planned enhancements include modifying the device so that write requests are no longer blocked by read requests. The first level would be to have the device send a message to the VMS job controller if the device is presently not owned. At a later date we will investigate adding a more complete level of unsolicited input support. We also plan to add support for associated mailboxes for both the I/O manager and the applications side of the device. The mailboxes on the applications side would provide the same support as the present terminal driver does. For the I/O manager side they would be used to notify the I/O manager of the applications side hanging up and possibly other significant events. The final changes fall in the area of general cosmetics. We will attempt to use as many of the terminal driver UCB fields as possible and remove any unnecessary duplicate fields that we presently have in the driver.

Summary/Conclusion

This paper has illustrated that there are many uses for Pseudo Terminals and many acceptable approaches to building them for the VMS operating system. It is fortunate for people wishing to build a Pseudo Terminal that VMS provides many alternatives and mechanisms for reaching their goal. As is frequently the case there is no one correct or optimal solution and hopefully this paper has

illustrated this. Based on the number of requests we have received and the response to the papers on this topic at the FALL 85 DECUS Symposium, there is a large user base who feel they have a need for a Pseudo Terminal. This paper has provided some insight into the trade-offs and problems for people wishing to attempt similar sorts of devices.

Finally, the approach we chose for our Pseudo Terminal has more than met our original design objectives. We were able to build test and debug it in less than the budgeted time. The device has been in use for over six months without any problems and has placed less load on the system than originally was allowed. We believe that for applications that are eventually going to do output to the VMS terminal driver, our approach is the optimal one, particularly when performance is a primary concern.

Acknowledgment

The authors would like to express their appreciation to Ms. Joy Dorman of Digital Equipment Corporation and Mr. Fred Schroeder of Titan Systems. Without Joy's help and guidance our Pseudo Terminal would not exist today; she kept us from making many mistakes during all phases of the project. Fred was critical in convincing the final users that the device we proposed was both feasible and the only reasonable solution for their problem.

INSTALLING MULTIPLE VERSIONS OF VMS LAYERED PRODUCTS

Gary L. Bellon
Monsanto Company
St. Louis, Mo.

ABSTRACT

This session discusses a method of Installing Multiple Versions of a Layered Product, such as Fortran, and have all accessible simultaneously. Installation of a new version of a layered product usually introduces problems. So it is desirable to do the installation at a time which is not critical to the users of the product. In a large processing environment made up of diverse groups, it is difficult to find a time which is not critical to one of the groups.

Along with the existing area, referred to as CURRENT, three additional version areas, NEW, OLD, and TEST are created, each able to hold one version of the same software package. Accessing of the multiple versions is accomplished by setting up additional directory trees, similar to the COMMON tree, and modifying search list to include one of the Version Areas. Third party packages are placed in a separate set of trees.

The session discusses in detail the problems and limitations of installing and maintaining products in these directory trees.

GOALS

One of our primary goals was to *not* conflict with the standard VMS System Disk structure, or the logicals which define it. We believe that our implementation accomplished this by the fact that we were able to perform our VMS Version 4.2 Upgrade without any adverse effects. The only consideration was to move the new DCLTABLES, and HELPLIB down to the other version areas.

Another important goal was to allow the normal use of VMS not to be affected unless specifically requested by the user, so that the general user would not have to be concerned about other versions of products in the version areas.

We also wanted, for those users who did wish to use other versions, to be able to select which version area (TEST, NEW, CUR, or OLD) with a minimum of effort. When references are made to the Current area, this is actually the standard VMS storage structure.

The implementation should also supply uniform storage for multiple versions of purchased software packages from Digital and other vendors. While this goal is listed last, it was actually one of some importance in that it standardized the way that multiple versions of the same package, or even just a procedure, were stored. Our experience had been that each person had his or her own manner of renaming files or directories.

LIMITATIONS AND CONSIDERATIONS

It seems that with everything there is some price to be paid. In this case the price is an increase in the amount of work that is involved in maintaining your system. The increases are in the following areas:

- Installations will be required multiple times. When

you wish to move a package to another version area, it will normally require that you go through the installation procedure another time.

- Records must be carefully maintained in order for you to know which entries have been made in common areas such as the DCL Tables and Help Library.
- The Startup procedures of some packages will have to be modified to make references directly to the version area in which the package is currently stored. This will be discussed in more detail later.

It is important to realize that by the nature of some products, having multiple versions running concurrently is difficult, and in some cases, impossible. An example of difficult is when a package uses system wide names for mailboxes or writable global sections. An example of impossible is when the package uses a particular piece of hardware. Even in this case, though, just the ability to store more than one version and to quickly stop one version and start another has proven to be very useful.

USAGE OF VERSION AREAS

The TEST area is used to hold software for which access is limited and/or is under development. This area is normally used to do initial installations and tests by the package maintainer. For packages or procedures which are maintained by several people, this area provides an easy way to determine whether or not another person is already working on some item. Access to the TEST versions will be limited by normal UIC based and ACL protection.

The NEW area is used to hold new versions of existing software and allow testing of them by interested users while the current version is still used for primary production. This allows a new version of a package to be phased

SYS\$SYSDEVICE
Master File Directory

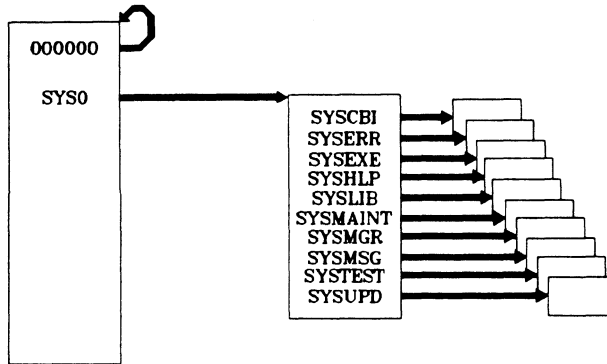


Figure 1: VMS Single System Volume

into use by each user or group of developers at a time which is not critical to their development cycle. Access to this area is World:RE, the same as current.

The OLD area is used to hold the previous version of a revised software package. This allows isolation of problems to a particular version by trying to duplicate the problems encountered when using the revised production version. In this case the package is removed from the OLD area after a sufficient period of time has elapsed to insure the stability of the revised version. A secondary use of the OLD area is to allow long term use of a replaced version of one package which is required by another package. Access to this area is also World:RE.

Interaction of different packages or images between version areas may exist. By that I mean, there will be instances when a user or system generated image in the current area may not be compatible with a library that is in, for example, the NEW area. This is to be expected, and in fact, is a major justification for the version areas, in that these types of incompatibilities can be identified and corrected before the package is moved to the CURRENT area and can effect all users.

The primary goal of the system manager should be to have all versions of packages within the CURRENT area compatible with one another. The other areas should be used to hold any images or libraries required for the correct operation of that area. For example, if a new version of TDMS is installed in NEW and your version of Datatrieve is linked with TDMS, switching to the NEW area and running Datatrieve will still be using the version in CURRENT, and may not be compatible. Therefore a version of Datatrieve linked with the TDMS in NEW,

should be created and placed in the NEW area.

A special case to consider, involves indefinitely retaining in OLD, a version of some package that is required by a second package. There may be some incompatibilities which you may wish to define and live with. This will be a judgement call for each particular instance, taking into consideration the severity and extent of the incompatibility and the importance of supporting the second package which requires the older version.

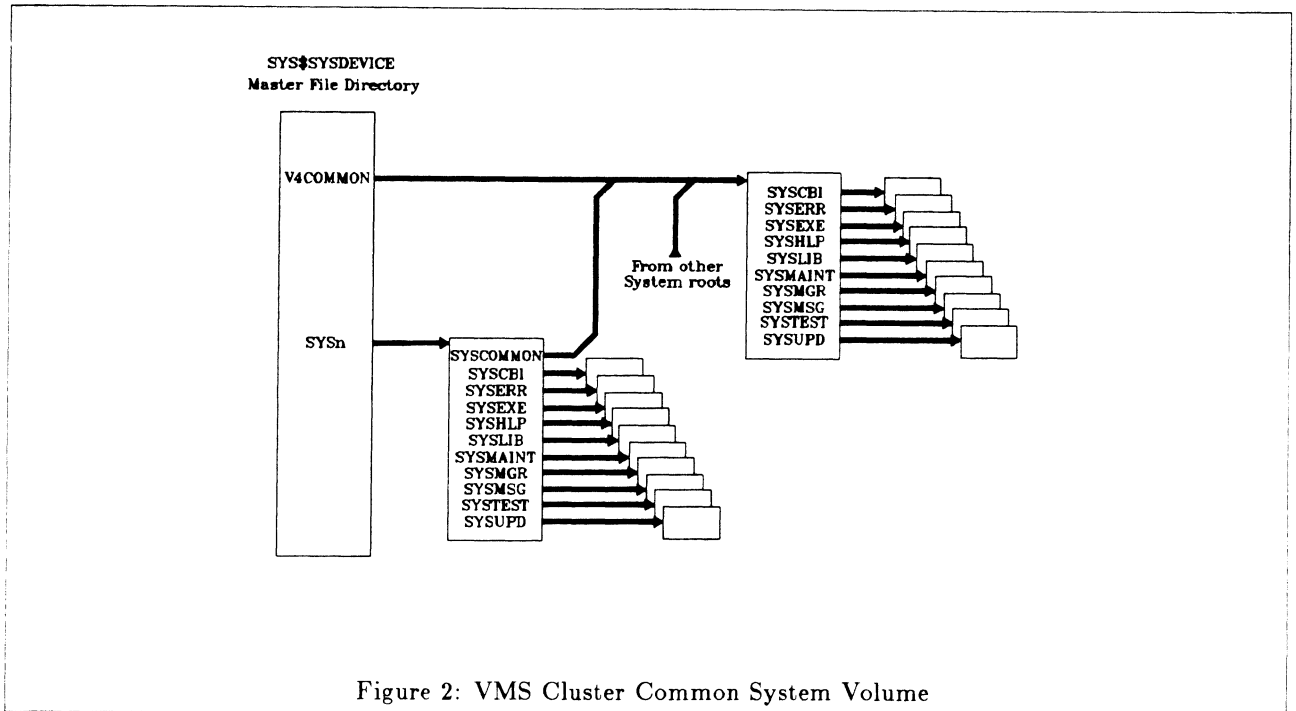
VMS SYSTEM DIRECTORY

The simplest form of the a VMS system volume is one which contains only one system. See figure 1.

I would like to make one point at this time to ease the understanding of these figures. The natural way that people think of a directory is as a group of files. Actually a directory is a file which contains a list of file names and pointers to those files. So each box in the figures represents a directory file and the names inside the box are the list of the files contained in that directory. As you can see in the Master File Directory, there is an entry for itself and the pointer points back to itself.

In the Master File Directory there is an entry for the directory SYS0. In the directory SYS0 is the list of directories that correspond to the system logicals, SYS\$SYSTEM, SYS\$LIBRARY, SYS\$MANAGER, etc. These entries, in turn point to their own individual directories which contain the list of the actual files. So these figures show only directory files, no data files.

There may be additional entries in the SYS0 directory depending upon which layered products have been installed. The VAX 11 RSX product creates the syn-



onym directories; 001001, 001002, and 001054. These entries actually point to the same directory files as SYSLIB, SYSMMSG, and SYSEXE respectively.

This directory tree structure is the basic unit required for a system and is the building block for the Cluster Common System Volume.

VMS CLUSTER COMMON

As distributed, the cluster common system volume has two basic sections, System Specific and Common. Refer to figure 2. The common area directory, V4COMMON, has the same format as the directory we just saw on the Single System volume. The system specific directory, which is normally referred to as a system's root, has only one addition entry and that is SYSCOMMON. This SYSCOMMON entry also points to the common area directory.

There can be up to 16 roots, 0 through F, with E being reserved for Standalone Backup and F being reserved for VMS Upgrades.

Each root has a pointer to the common directory. Not being a Digital employee, I can only surmise that this was done to allow for more than one common area, for example during an Upgrade, or possibly to allow a completely separate system to reside on the pack by removing the SYSCOMMON entry from the root directory. It would still have to be a member of the cluster to avoid corrupting the volume.

Most of the VMS files reside in the Common directories with only those files with specific data for one system residing in that system's root directory.

SYSTEM VOLUME WITH VERSIONS

A VMS system volume is modified to support multiple versions by creating three addition directory trees, each being an exact copy of the common directory. The names, as you might expect, are TEST, NEW, and OLD. Refer to figure 3

In creating these directories, it is important not to use rooted logicals, meaning those logicals which contain a device, a directory and end the directory specification with a ".". This is because they have the habit of also creating a lower level [000000] directories. It is best to use a full device and directory specification such as:

```

$ CREATE/DIR SYS$SYSDEVICE: [TEST.SYSEXE]

```

The files and directories within these trees are, as is the case in the other VMS directories, owned by the SYSTEM UIC, [1,4]. Access to the NEW and OLD directory trees will be the same as the access to the Common area. The world access to the TEST directory tree should be removed.

If you are performing this procedure on a Single System volume, the operation is exactly the same.

You will need to copy into each directory tree from the common directories the files; [SYSHELP]HELPLIB.HLB, [SYSLIB]DCLTABLES.EXE, and an image you will have to create, [SYSEXE]ACCESS_x_TEST.EXE. The x in the image name will be T, N, and O for the TEST, NEW, and OLD areas respectively. This image is nothing more than an exit statement with a successful return status. It is used when switching versions to insure that the user has access to this area.

A separate version of the DCLTABLES and HELPLIB library are required because these files are normally mod-

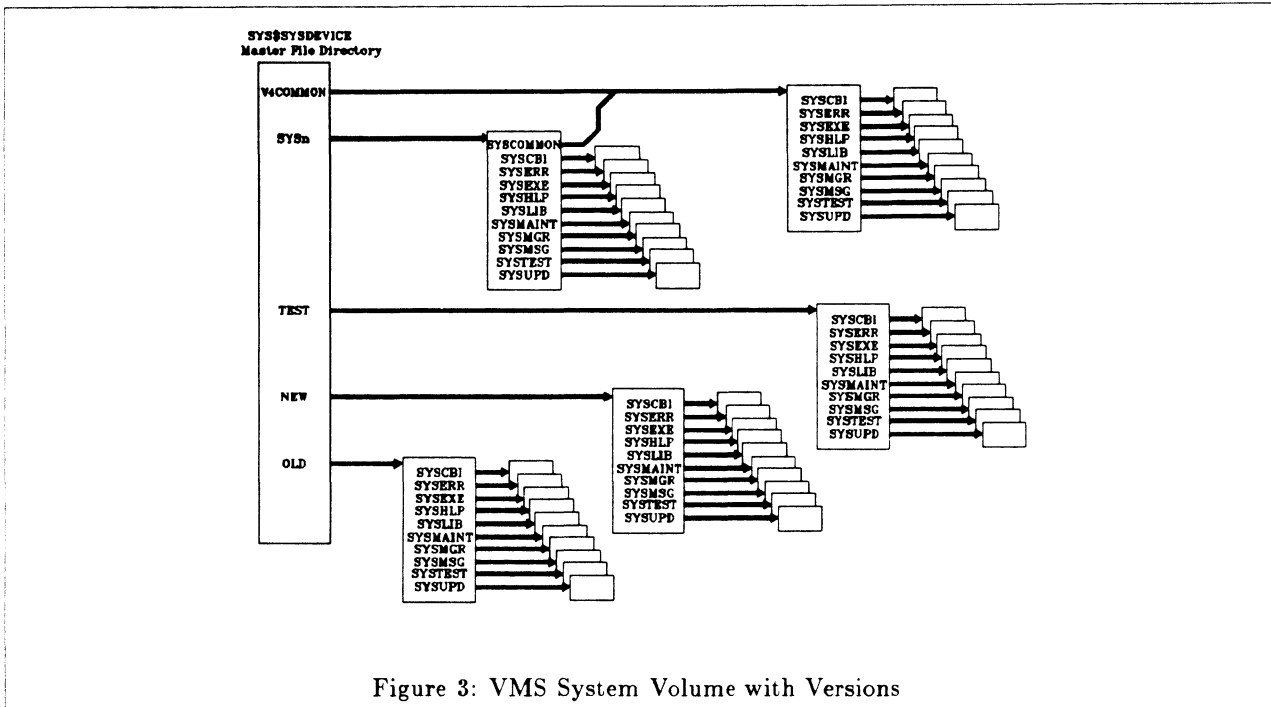


Figure 3: VMS System Volume with Versions

ified when a package is installed, and you will want these files to reflect the actual contents of the version area.

CUR:	"SYS\$SYSROOT"	=	"\$1\$DUAX: [SYSn.]"
		=	"SYS\$COMMON"
TEST:	"SYS\$SYSROOT"	=	"SYS_SYSTEST"
		=	"SYS\$SPECIFIC"
		=	"SYS\$COMMON"
NEW:	"SYS\$SYSROOT"	=	"SYS_SYSNEW"
		=	"SYS\$SPECIFIC"
		=	"SYS\$COMMON"
OLD:	"SYS\$SYSROOT"	=	"SYS_SYSOLD"
		=	"SYS\$SPECIFIC"
		=	"SYS\$COMMON"
<hr/>			
"SYS\$SPECIFIC"	=	"\$1\$DUAX: [SYSn.]"	
"SYS\$COMMON"	=	"\$1\$DUAX: [SYSn. SYSCOMMON]"	
"SYS_SYSTEST"	=	"\$1\$DUAX: [TEST.]"	
"SYS_SYSNEW"	=	"\$1\$DUAX: [NEW.]"	
"SYS_SYSOLD"	=	"\$1\$DUAX: [OLD.]"	

Table 1: System Search Lists

SYSTEM LOGICALS/SEARCH LISTS

The logical search lists required to accomplish access to the various System Version sections are shown in Table 1. The search list in the normal VMS environment is the logical SYS\$SYSROOT. This standard system logical

search list is modified for searching in the version areas by adding the logical that defines the version area as the first item in the search list.

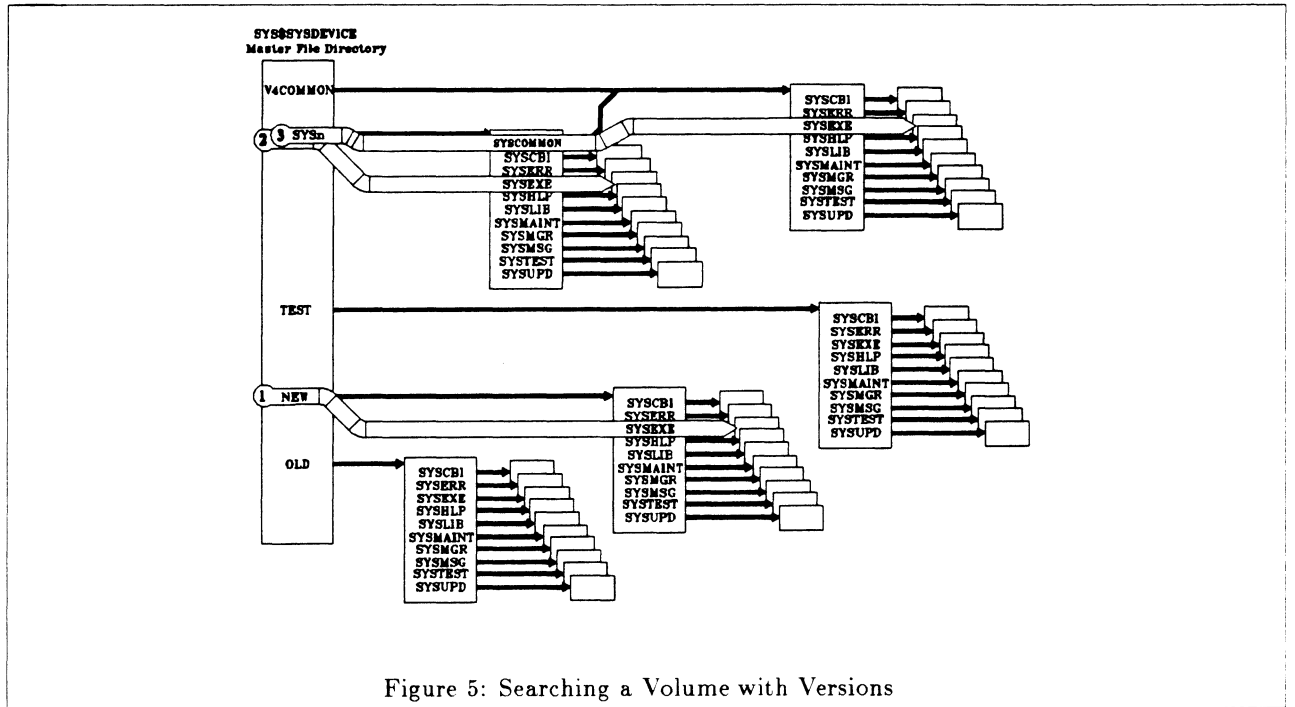
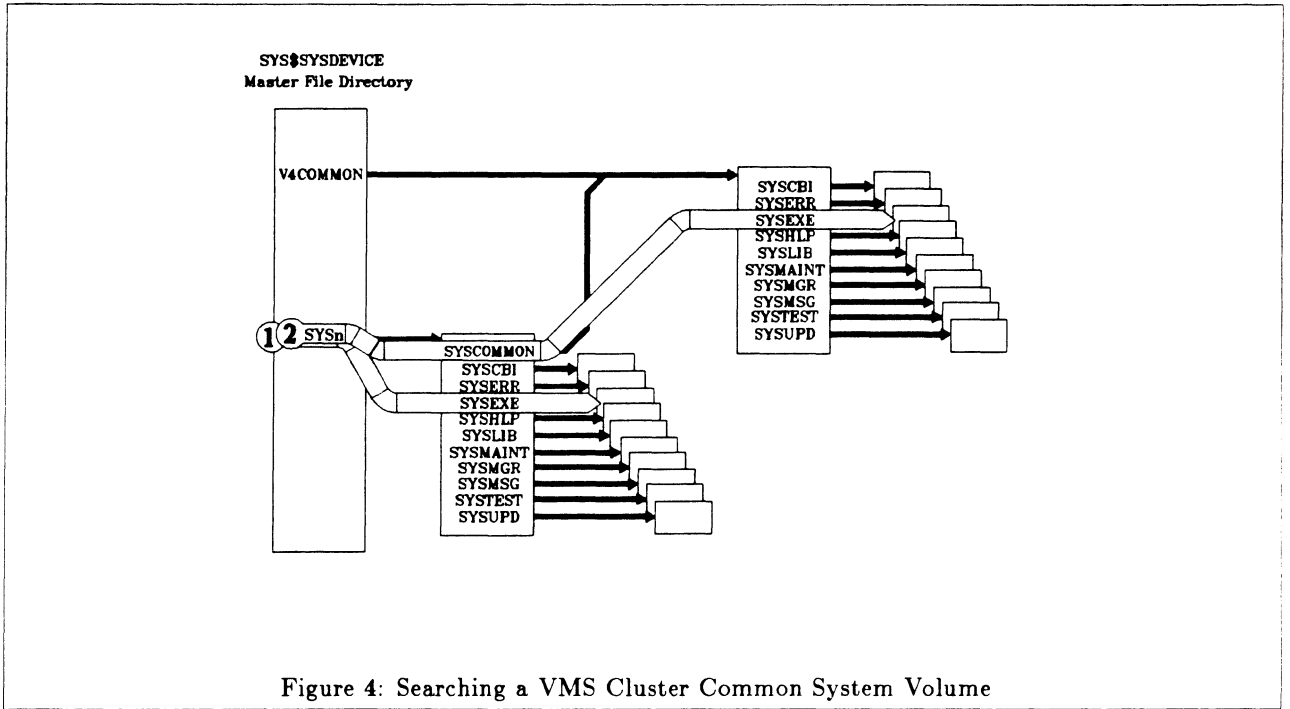
You may have noticed that the second item in the search list for the version areas was changed from a device and directory to be a logical which translates to the same device and directory. This was done because the search lists which include the version areas are defined in Supervisory Mode, so that they are not "trusted" logical names. A "trusted" logical name being one defined in Kernel or Executive Mode. This is important in gaining access to privileged images, sharable images referenced by privileged images, ... Just as a note, the subject of when "trusted" logical names are required is covered in detail in a News Bulletin in the November VMS Systems Dispatch, but one important detail was left out and that is only the last logical translated must be a "trusted" logical.

The logical translations at the bottom of the tables show the logicals used in the search list. The first two are defined in the STARTUP procedure. The last three, you will have to define in your SYSTARTUP as system wide Executive Mode logicals. We used an "_" instead of a \$ because of possible future conflicts.

On Single System Volumes, logicals SYS\$SYSROOT, SYS\$SPECIFIC, and SYS\$COMMON all translate to the value shown in the table for SYS\$SPECIFIC. In that case, all of the search list in the table would leave off the last item, SYS\$COMMON.

SEARCHING A STANDARD VOLUME

The search of a standard VMS volume is performed in two steps, as defined by the logical SYS\$SYSROOT. The



example shown in Figure 4, is a search for an image file that is referenced using the SYS\$SYSTEM logical. The first step of the search is to open the Master File Directory (MFD) and search for the system's root directory entry. Upon finding that, the system opens the root directory and searches for the entry of the image file that

is to be activated. If the entry is not found or access is blocked, the system then tries the second item in the search list. This time searching the root directory for the SYSCOMMON entry. Upon finding that, it opens the common directory and again searches for the SYSEXE entry. This directory is then searched for the entry of the specified image.

SEARCHING A VOLUME WITH VERSIONS

To use packages from one of the version areas, the standard VMS search list is modified, as we saw in the previous table, to include one of the three logicals that translate to a version directory. So the search now is made up of three steps. The first step is to search the MFD for the version directory, and upon finding it, opening that directory and searching for the SYSEXE directory entry. Then searching in that directory for the specified image. If, as before, the image is not found or access is blocked, the system then proceeds to the next items in the search list, which are the same as was just described in the standard VMS search.

SITE SPECIFIC DIRECTORY

For the utilities and procedures that we add to VMS for system management, operational support, and general user reference, we have created a separate set of directory trees shown in Figure 6. Without going into a lot of the specifics concerning our tree structure, all that is really important to the implementation of version areas is to take the entries that are in the MFD and copy them to a lower level directory named CUR. The other version areas are made by duplicating this tree in their directories

Also software packages purchased from third parties or developed by an in house staff, can be placed in the version directory trees. The subdirectory tree structures can remain as they were defined by the supplier of the package. The only difference is that they are in a lower level directory instead of the MFD, but this is made transparent by the use of rooted logicals.

The trees, as far as versions are concerned, are used in the same manner as the VMS system directories, and therefore do not need any further explanation.

An interesting point is that these directory trees are split across two physical volumes.

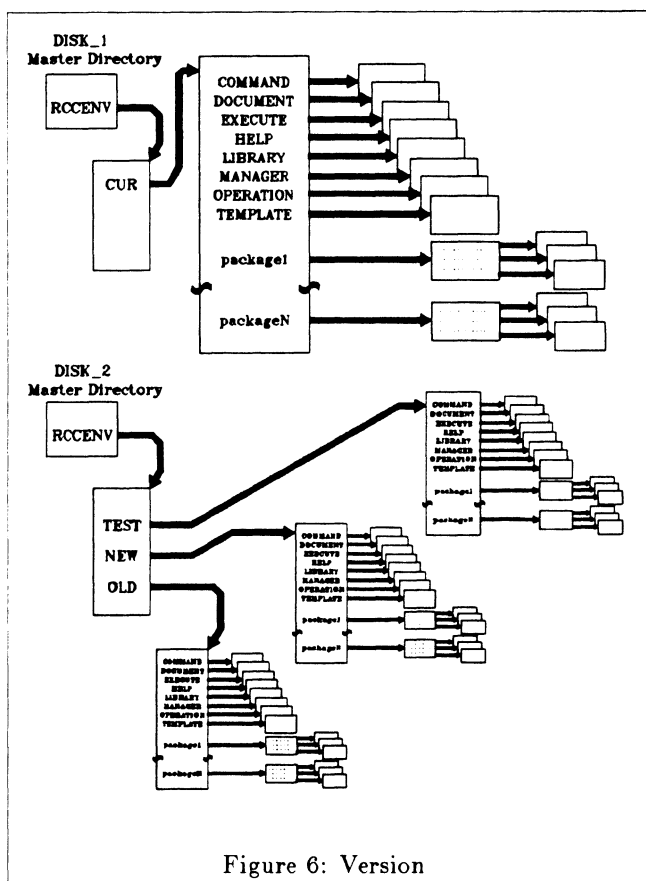


Figure 6: Version

SITE SPECIFIC LOGICALS/SEARCH LISTS

The logicals used to gain access to our site specific version areas are shown in Table 2. The manner in which the search lists are modified for the versions areas is the same as what is done with the VMS search list, i.e. an item which defines the version to include is added to the beginning of the search list. Again, remember to use only "trusted" logicals as items of the search list.

SPECIFYING DEFAULT VERSION

In order to use one of the versions other than Current, the user must request the version by using the command: VERSION parameter, where the parameter entered is either TEST, NEW, CUR, or OLD. This procedure performs the following steps:

- First, the parameter is verified to be one of the possible version areas, and the procedure branches based upon which version was requested.
- The search list logical for SYS\$SYSROOT is created in the user's process table which will include the specified System Version section, unless the version requested is CUR, in which case the logical in the process table is removed.
- The search list logical for RCC_ENV is created in the user's process table which will include the spec-

CUR:	"RCC_ENV"	=	"\$1\$DUaz:[RCCENV.CUR.]"
TEST:	"RCC_ENV"	=	"RCC_TEST"
		=	"RCC_CUR"
NEW:	"RCC_ENV"	=	"RCC_NEW"
		=	"RCC_CUR"
OLD:	"RCC_ENV"	=	"RCC_OLD"
		=	"RCC_CUR"
<hr/>			
	"RCC_CUR"	=	"\$1\$DUaz:[RCCENV.CUR.]"
	"RCC_TEST"	=	"\$1\$DUay:[RCCENV.TEST.]"
	"RCC_NEW"	=	"\$1\$DUay:[RCCENV.NEW.]"
	"RCC_OLD"	=	"\$1\$DUay:[RCCENV.OLD.]"

Table 2: Site Specific Search Lists.

ified RCC Version section, unless the version requested is CUR, in which case the logical in the process table is removed.

- The characters T|, N|, or O|, are appended to the beginning of the users prompt to be a constant reminder that the software being referenced may not be the production version. If the version specified is CUR, the version indicating characters is removed from the prompt.
- Since the access to the TEST directory will be limited and because the noncurrent versions of the site specific sections are on a separate volume, an image is activated in both the VMS and site specific sections of the requested version. If the requested version can not be accessed, the abort sequence will be activated.
- A list of the packages found in that Version is then printed. In each RCC_ENV:[DOCUMENT] directory there is a file called PACKAGE_LIST.DAT
- An error or a request to abort the procedure will cause the procedure to return the process to the Current version.
- The DCL tables within that version will be loaded into the process.

All packages within a version will be accessed when that version is requested. Individual selection of packages within a version will, for the most part, not be possible. Planned to be implemented in the future is a procedure, which will be named EXECUTE.COM, which will be activated with two parameters; the version to be activated from, and the symbol normally used to activate the package. This activation method will *not* be possible for packages which are defined in the DCLTABLES, meaning almost all of Digital's layered products, but is possible for many third party packages. The maintainer of the packages will allow this type of activation by defining a command procedure of the name *pkg_VERSION.COM* in the same directory as the normally activated image or procedure. In this way, each maintainer of a package can determine individually if that package is suited to be activated independently.

INSTALLATION PROCEDURE

In order to install a layered product into a version area all that is required is to switch your process to the desired version and to specify to VMSINSTAL the version directory as the the destination root. The example shows the product saveset coming from a disk volume. An explanation of the parameters for VMSINSTAL can be found in the *Guide to Software Installation*.

```
$ ! Install a Package in a Version Area.
$ VERSION NEW
```

```
.
```

```
.
```

```
.
```

```
$ @SYS$UPDATE:VMSINSTAL FORTO43 DIST:[FORTO43] -
  OPTIONS R $!$DUAX:[NEW]
```

```
$ DIR SYS$SYSDEVICE:[*...]/MOD/SINCE=start_time
$
```

For record keeping purposes, it is a good idea to perform a directory on the system device of all the files modified since the installation started and direct the output to a file. The listing should be checked to insure that only files in the root you specified have been changed. This list will also be needed when the the package is removed from that version area.

Another planned enhancement is to create a procedure which will call VMSINSTAL for you, but will accept as parameters, a more understandable name for the package, and a logical name to define the root. This procedure will update the PACKAGE_LIST file with a single entry, rather than the sometimes multiple, cryptic product saveset names. This procedure will convert the name to a proper saveset name(s), translate the logical, and pass the parameters to VMSINSTAL, multiple times if required.

SOFTWARE INSTALLED

As of this time we have successfully installed the following products into a version area:

- FORTRAN
- VAXC
- PASCAL
- LSE
- DECSPELL
- MMS
- BASIC
- CMS (Future versions will not be compatible)
- DATATRIEVE (Only after many corrections were made to the KITINSTAL)

More will be stated later about the specific problems encountered while installing packages.

VMS updates which only effect utility routines could also be put into the Version area, although this should generally be avoided. Modified versions of the standard System modules and Drivers can not be accessed from the versions areas during a system boot.

STARTUP CONSIDERATIONS FOR CONCURRENT VERSIONS

The image activator first checks for any possible installed images based upon the search list that the image is referenced by. So for example, if the image is specified as SYS\$SYSTEM:*image*, all possible variations of that search list are checked for an installed image of that

name. This means that if there are two versions of an image, one in a version area and one in the current area, but only the one in the current area is installed, the current version will be the only version ever activated because it is always in the search list. So, if the current version of an image is installed, then all versions of that image must be installed.

When an image is installed /SHARE, there are global sections created based upon the image's file name. So, if more than one version of the image is installed shared, there will be global sections of the same name. VMS does not maintain any link between an image installed /SHARE and the global sections associated with that file, it just does a top down search of the table to find a global section with a name that corresponds to the file name, so only the entry nearest the top of the table is used. For this reason, only one version of an image can be installed /SHARE, normally the one in current because it should be the one most used. The other versions will create private sections for each activation.

If there is a startup procedure supplied with the product, it will be necessary to merge the startup procedures from other versions into the one in the current area, since it is only one called at startup. If some images are installed, the procedure must be modified to insure that the version directory is specified explicitly, e.g. SYS_SYSNEW:[SYSEX].

STARTUP CONSIDERATIONS FOR NON-CURRENT PACKAGES

Starting of packages which can have only one active version may be done in one of two ways; the first being based upon the version which is the first item in the logical search list, and the second being based upon the directory tree which holds the procedure which starts the package.

Start Up Based Upon Current Default Area

A package's startup procedure can be made to start up the version of the package based upon the current values of the search list logicals. The example shows a loop to insure that the logical is translated to its final value. The final value is then appended on to the beginning of the file specification to be the value of that package's logical names.

```
$ ! Define the Package's Root logical name.
$ temp = "RCC_ENV"
$trans_loop:
$ dev_dir = temp - "]"
$ temp = f$strnlm(temp)
$ if temp .nes. "" then goto trans_loop
$
$ def/sys PKG_LOGICAL 'dev_dir'PKG]
$
```

Start Up Based Upon Procedure's Location

Another method is to start up the version of the pack-

age based upon the directory in which the procedures reside. Determine the procedure's name from the lexical, extract the device and top level directory, and then define the package's logical

```
$ ! Define the Package's Root logical name.
$ temp = f$environment("procedure")
$ dev_dir = temp - "]"
$
$ def/sys PKG_LOGICAL 'dev_dir'PKG]
$
```

SPECIFIC PROBLEMS

The problems encountered maintaining versions areas can be grouped under the three general headings of; limitations on concurrent versions, installation problems, and general precautions.

Limits on Concurrent Versions

If a package contains a protected sharable image, this will preclude it from ever having more than one active version. This is because of the the limit of no more than one version installed /SHARE. CMS, CDD, and SPM are layered products we know include this type of image.

If a package has a writable global section, this will preclude it from ever having more than one active version. This too is because of the limit on the number installed /SHARE.

The use of a system wide mailbox name will also preclude there being more than one active version. There is a way to specify the logical tables in which the mailbox names go into, but the complications involved with this were considerable. In any case, the developers of our major third party packages have the same problem of running multiple versions so they already change mailbox names with each new version.

Packages which are associated with a particular piece of hardware, can also only have one active version.

Installation Problems

During the installation of the layered product, various system files and utilities must be able to be referenced in the same root as the installation is being done. Unfortunately, the KITINSTAL.COMs do not precheck all of the requirements at the beginning of the installation as they do to generate the list of active processes, so you must run the installation until it fails on a file reference. In some cases, as with Datatrieve 3.2, the procedure was not reporting which files were not found. So with some packages you will have to read and interpret the KITINSTAL procedure. After determining all of the files required for read only reference, they are copied to the version area, and then deleted after the installation.

It would be nice if the first thing that any KITINSTAL performs is a check for required files and reports any deficiencies before starting the main installation. Even better would be for the KITINSTAL to use the SYS\$... logicals

for read only references, and the VMI\$... logicals created from the root specified by the input parameter, for the modify references.

On the other hand, there has been instances when modify references were made with the SYS\$... logicals, meaning that files in the common area have been modified. That is why I recommend running a DIR /MOD /SINCE=*start_time* after each installations. If files are found to have been modified they should be moved into the directory tree in which the installation was being performed.

As long as the products flow through the version areas in the expected sequence, entries made in common files in the new area such as the DCL Tables and the Help Library, will remain compatible with the entries for the product in the current area, even though the specific files for that product have been removed from the new area. Should a version of a product, though, not work out to your satisfaction and the decision is made to remove it from the system. The specific files can be removed in the same way they would be if the product had moved up, but you are left with entries in the DCL Tables and Help Library which are not compatible with version of the product in current. The easiest thing to do just may be to reinstall the version of this product that is in the current area into the new area and in that way replace the entries in DCL Tables and Help libraries so that they are then the same as the current area's entries.

General Precautions

Care should be taken to read the Release Notes of any product before installing it to try to avoid conflicts such as the new version changing the format of a file or data structure, making it incompatible the existing version still in production. Each situation will require not allowing concurrent versions, or instructing and trusting the users of the product that once the new version has been used, to use only that new version.

Changes in the manner in which an image is activated, from one version to another can be another source of problems. For example the previous version of CMS was activated by a symbol using the dollar sign, the new version has a DCL Table entry. So those users who privately created the CMS symbol, received an activation error when trying to run the new version.

There are still a few peculiarities with the use of search lists which seem to be limited to commands like SET FILE/OWN, when a wild card is included in the argument which makes it span volumes. These types of errors will have little if any effect on general users but may make the maintainer's life a little less predictable.

The most visible aspect of search list that can sometimes confuse a user is that the status resulting from a bad file access, is the result of the last item in the search list. This means that you may get a File Not Found Error, when actually there was a Protection Violation on the file found in a higher item of the search list. This seems like one of those things that has to just be kept in mind while

using the system.

Be aware that the different editors handle search list differently: some always put the file in the directory pointed to by the first item of the search list, others put the output into the same directory that the input was found, and still others will do both depending upon whether it recognizes the directory tree as a VMS structure or not.

One last little mention of creating directory trees with rooted logicals, they will sometimes create a middle level directory by the name of [000000], so be sure to fully specify the argument for the Create Directory command.

Daniel P.B. Smith and Robert B. Goldstein
 Eye Research Institute of Retina Foundation
 20 Staniford St
 Boston, MA 02114

ABSTRACT

Our facility is equipped with VAX and Macintosh computers, and we try to take advantage of the best features of each. Taking advantage of the graphics and word processing capabilities of the Mac, we off-load those applications to the Mac and thus free the VAX for large database applications that require good response time. We also use the VAX as a store-and-forward system for Macintosh objects, partly eliminating the need for Apple-talk. We run MacTerminal VT100 emulation software on our Macs and connect them to the VAX using existing terminal cabling (RS-232). A program, MACSnVAX, makes the VAX appear to the Macintosh as another Macintosh so that MacTerminal can perform "Mac-to-Mac" file transfers. We have loaded a large library of Mac software onto our VAX, which is accessible to all Macs in the building. We have also written a program, TIF, that interchanges MacWrite and Microsoft word documents with Word-11 documents on the VAX.

INTRODUCTION

This paper focuses on the operation of a program called MACSnVAX. We will present a sample session and several tasks for which MACSnVAX can be used.

At the Eye Research Institute (ERI) we have 5 VAX computers linked via Ethernet, with some 50 terminals, printers, and plotters attached to them through a MICOM switch. We also have about 20 Apple II, 10 Macintosh, and 5 IBM PC computers (Figure 1). Our central computer staff consists of three professional-level people.

DIVISION OF LABOR

On our VAX systems we perform word processing with Word-11, biostatistics and graphics using RS/1, modeling, database applications, reference library, and general accounting applications. We use one VAX for image processing. On the Macintosh computers we run Microsoft Word for word processing, NWA-Statpak for statistics, and the standard Macintosh programs MacDraw and MacPaint. Some users employ RUNOFF, Applewriter II, and MacWrite for word processing.

In the areas of word processing and statistics some overlap occurs for the following reasons: (1) Individualism - we have 50 independent investigators who like to exercise total control over their resources and

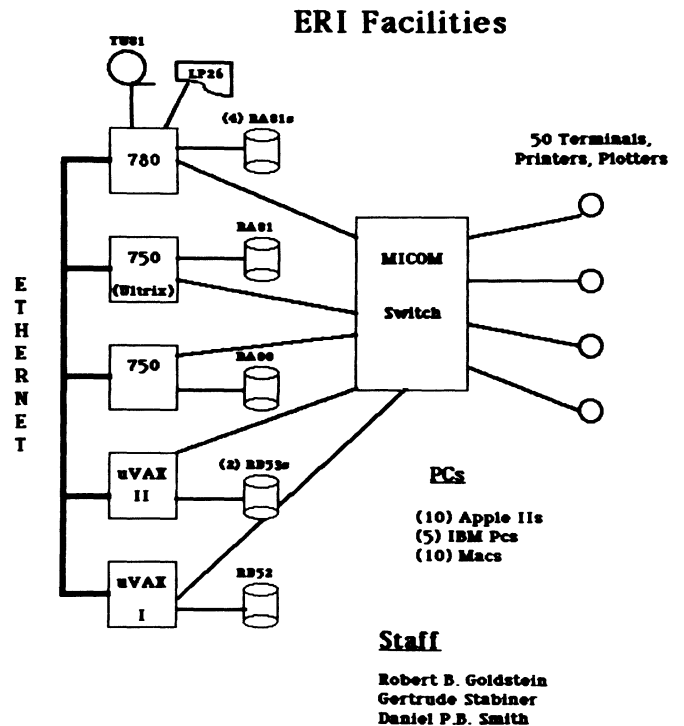


Figure 1. ERI Computer Facilities and Staff
 prefer to use their own microcomputers.
 (2) Offloading - we encourage people who want to do only word processing to use the

Macintosh. This allows us to maintain a better response time on the VAX system. (3) Convenience - it is convenient for database users to do word processing on the VAX rather than switch to other machines. This overlap creates problems in the areas of file transfer and file compatibility.

MACSnVAX

MACSnVAX, one of two programs we wrote to help solve the above problems, may be used in the following ways:

- As a "store-and-forward" system. One user can upload his files to the VAX, and at a later time another user can download the file to his Macintosh.
- As a library. The Boston Computer Society has released approximately 20 disks of public-domain software. We have loaded a large portion of this library onto the VAX and made it available to all the Macintoshes on our net.
- As a slow file server. Users can take advantage of the vast disk capability of the VAX to store all their files. When they wish to use one of them, they can download it to their Macintosh. Operating at 9600 baud, it takes only a few minutes to download a document of substantial size. Note, however, that cost/byte of storage is much higher on the VAX than on Macintosh disks.
- As an archiver. We have an extensive backup policy for our VAX systems. Therefore, Macintosh documents that are no longer needed but shouldn't be deleted, can be loaded onto the VAX where they will be backed up on tape and kept for several years.

- As a text file converter. MACSnVAX can convert Microsoft Word documents to VAX text format for editing with EDT.

MACSnVAX has the following capabilities:

- It works with MacTerminal, using the XMODEM protocol, to make the VAX look like a Macintosh for purposes of file transfer.
- It can store any type of Macintosh object (programs, documents created with MacDraw, MacPaint, Microsoft Word, etc.) on the VAX. Any valid 3-fork Macintosh file can be stored on the VAX.
- It produces a Macintosh-like catalog of the objects stored on the VAX, using the CATALOG command.
- It can store a descriptive paragraph with each object, using the ANNOTATE command.
- It can display text from Microsoft Word files, or extract it so it may be manipulated with EDT or other VAX editors.
- The MACSnVAX internal Command Line Interpreter is identical with the VMS CLI, so that a familiar environment is provided.
- Wildcards are allowed where appropriate.

MACSnVAX Session and Use

Figures 2 through 5 show a typical MACSnVAX session.

Figure 2 shows a Macintosh Desktop with the MacTerminal program and the document "VAX" that contains the correct settings for the user to log in to the VAX.

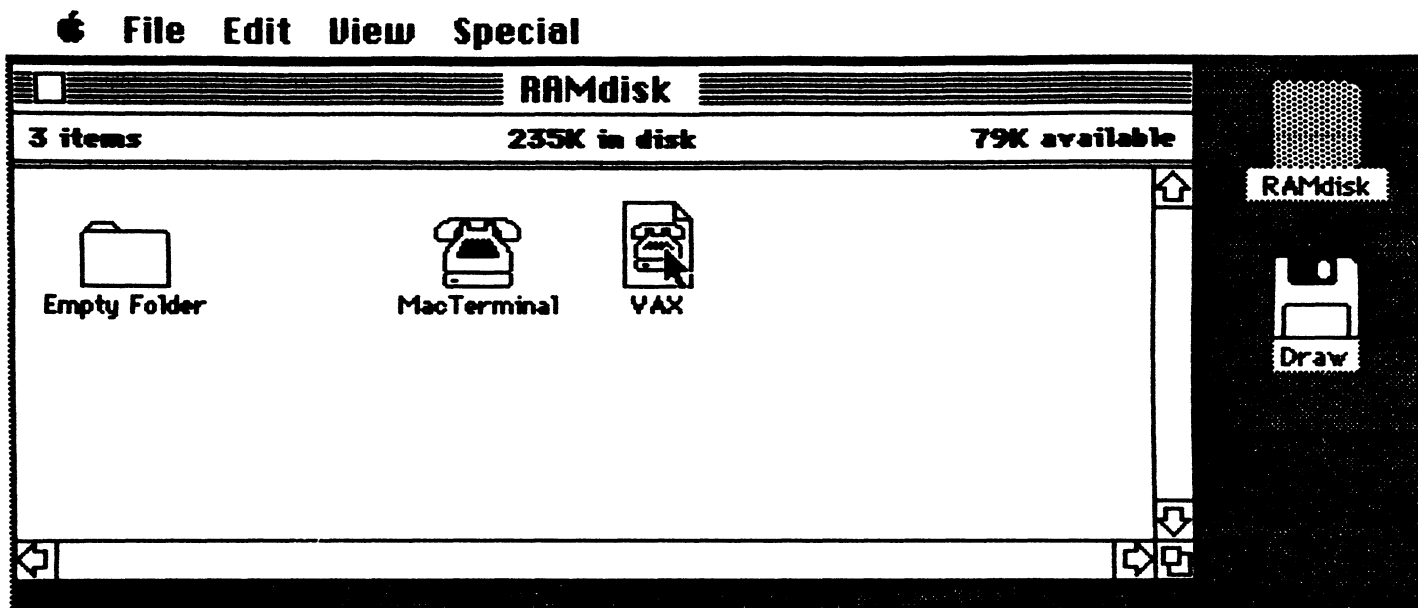
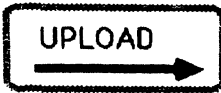


Figure 2. Initial Macintosh Desktop

```
$ macsnvax
MACSnVAX v4.1, copyright (c) 1985
by Eye Research Institute of Retina Foundation
All rights reserved
```

You're now in main section



H)elp, C)atalog, G)etinfo, U)pload, D)ownload, S)ection, Q)uit

MACSNVAX>u

Please send your file.
(pull down "File," select "Send File...")

Received Macintosh file ERI
as VMS file DU1: [COMPUTER.SMITH.MACDEMO]ERI.MAC;

To add a description of this file, use the a)nnotate command.



H)elp, C)atalog, G)etinfo, U)pload, D)ownload, S)ection, Q)uit

MACSNVAX>a/t

Annotation file: DU1: [COMPUTER.SMITH.MACDEMO]ERI.INF;

Annotation method:TYPE

Limit each line to about 60 characters.

Erase individual characters with RUBOUT, lines with CTRL-U

Type CTRL-Z after last line.

You can begin typing your description when a clear line appears.

Bob--this is the MacPaint document we use as the ERI startup
screen. Maybe you can use it in the DECUS presentation. --Dan
^Z

Figure 3. Examples of the UPLOAD and ANNOTATE Commands

```
$ macsnvax
MACSnVAX v4.1, copyright (c) 1985
by Eye Research Institute of Retina Foundation
All rights reserved
```

You're now in main section

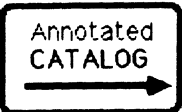


H)elp, C)atalog, G)etinfo, U)pload, D)ownload, S)ection, Q)uit

MACSNVAX>c

Catalog of main section

VMS name	Macintosh file	Type & creator	Size	Last modified
AUSTIN	Austin Econ	(New) Font doc	34K	14-SEP-1985 12:21:35
AUSTIN_DOC	Austin Econ Doc	MacWrite	17K	2-NOV-1985 10:48:29
BIGCURSOR	BigCursor	Application	2K	14-OCT-1985 02:37:52
ERI	ERI	MacPaint	9K	21-OCT-1984 19:38:58



MACSNVAX>g eri

Annotated Catalog of main section

VMS name	Macintosh file	Type & creator	Size	Last modified
ERI	ERI	MacPaint	9K	21-OCT-1984 19:38:58

Bob--this is the MacPaint document we use as the ERI startup
screen. Maybe you can use it in the DECUS presentation. --Dan



MACSNVAX>d eri

Sending: ERI
9216 bytes (data) 0 bytes (resource) 9216 bytes (total)

Figure 4. Examples of the CATALOG, GETINFO, and DOWNLOAD Commands

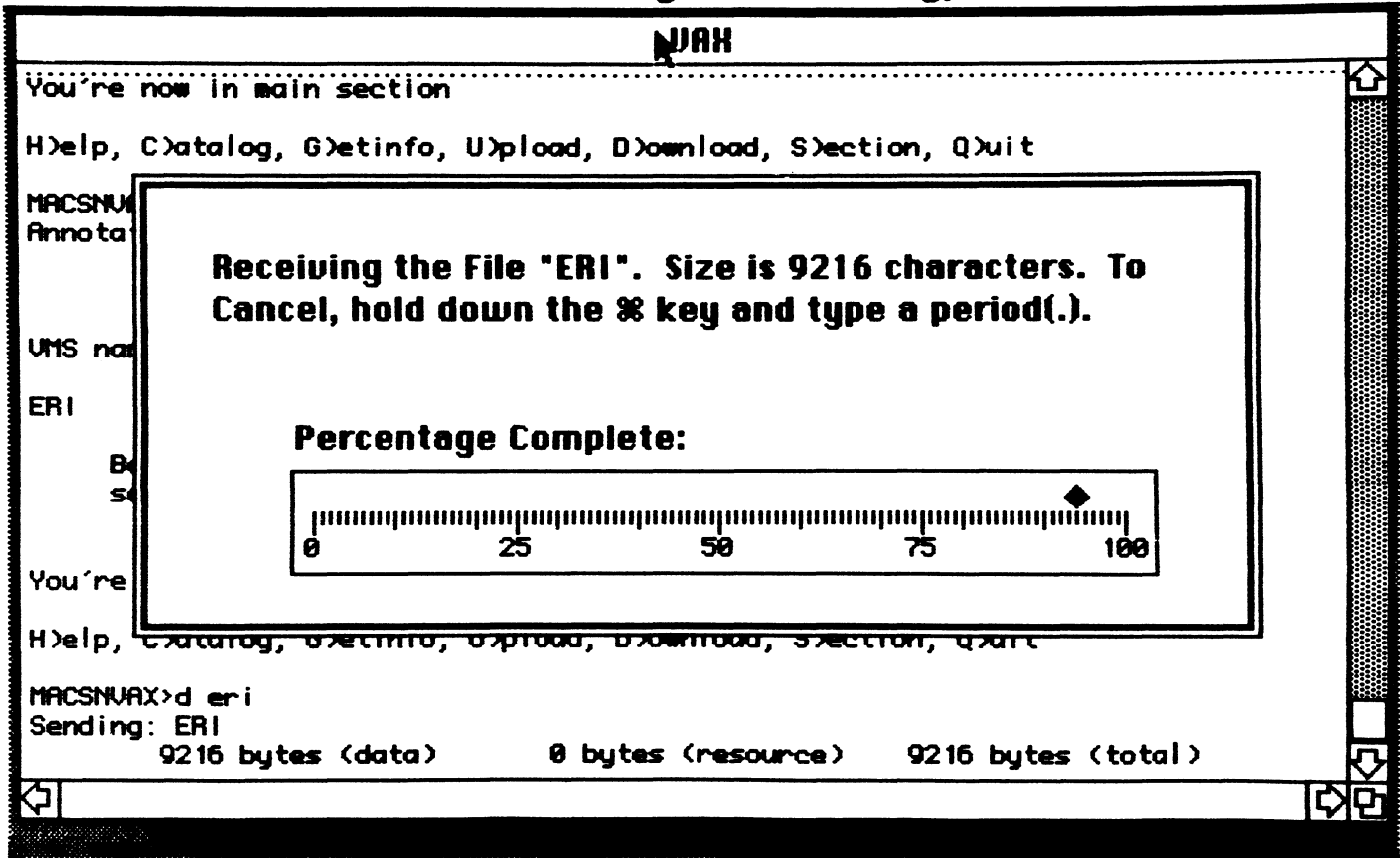


Figure 5. MacTerminal File Transfer Screen

Figure 3 shows a user uploading an object, ERI, to the VAX. MACSNVAX creates a VMS filename for the object. After the object is uploaded, the ANNOTATE command is used to create and store a descriptive paragraph with the object.

Figure 4 shows the use of the CATALOG and GETINFO commands to display information about the object, and the start of the DOWNLOAD command.

Figure 5 shows the Macintosh screen during the download operation. After the download the ERI object appears on the Macintosh Desktop.

A Macintosh object that resides on the VAX can be used only in limited ways. Currently, it can be used in conjunction with our Image Analysis facility, or, if it is a text object, it can be converted into a format that can be read by one of the VAX word processing programs.

The Image Analysis system, an Adage 3000, has no alphanumeric capability, and therefore we must use the Macintosh text to annotate images. Although MACSNVAX is not yet in active use for this purpose, it allows us to use the powerful drawing capability of MacDraw for Image Analysis applications.

We have written a program, TIF, that transfers documents from one word processing program to another (Figure 6). TIF uses a RUNOFF-like format that contains a least common denominator set of word processing capabilities. TIF assumes that if a document is transferred, it is also going to be re-worked. Although TIF does not preserve complex formatting instructions, such as

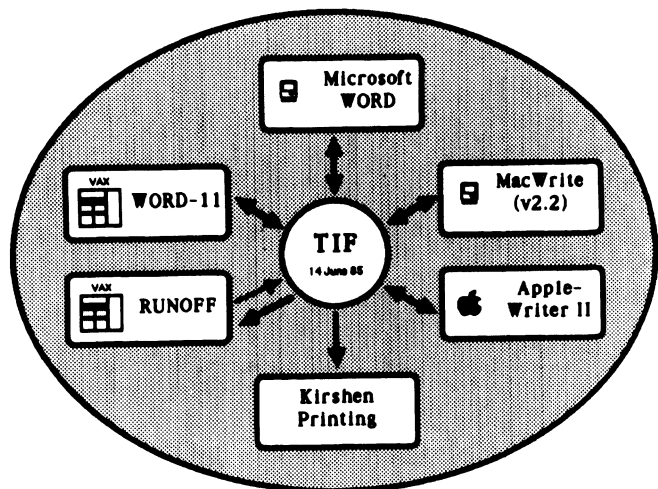


Figure 6. TIF and its Supported Word Processing Systems

columnar output, it does preserve centering, paragraphing, and character highlighting (bold and underline). TIF performs a 2-step transfer: first the document is converted to TIF, and then re-converted from TIF to its new format. TIF currently works with RUNOFF, Word-11, Microsoft Word, MacWrite, and Applewriter II. We also use TIF to send documents electronically to the printer of our quarterly newsletter.

FUTURE UPGRADES

We plan the following enhancements to MACSnVAX:

- o Better handling of VMS filenames. Currently filenames are truncated to 9 characters.
- o MACSnVAX options will be handled with a SET command internal to MACSnVAX.
- o Improvement of the primitive VIEW command that allows us to view MacPaint documents on a VT240, and its integration into our image processing system.
- o Provisions for a rename and/or delete function from within MACSnVAX.
- o Additional TIF modules, probably to support IBM PC word processors.
- o Addition of a MacMail facility, allowing Macintosh objects to be mailed from one user to another.

CONCLUSION

By implementing a program such as MACSnVAX, we hope to approach the ideal of using VAX systems and Macintosh computers for what each does best.

POSTER PAPERS

A FORTH-83 Standard FORTH

M. P. Hanson
Department of Chemistry

and

R. J. Wilson
Computing Services
Humboldt State University
Arcata, California 95521

James' original version of FORTH has been substantially revised to bring it into compliance with the 1983 standard of the language. The source program is written in MACRO-11 and can be compiled to run under RT-11, RSX-11 or RSTS/E.

In addition to the standard word set the split file structure of V. Vinge (SDSU, San Diego) has been accommodated. This file structure is extremely useful in a teaching environment using a time sharing system. With such a system a student requires a minimum disk allocation, can use the full language, and can be given messages or software from the instructor.

The users of FORTH have attempted to standardize the language. Over the life of the language there have been three standards adopted. The first FORTH-77 followed relatively quickly by the standard FORTH-79 published by the FORTH Standards Team in 1980. The most recent standard FORTH-83 was published by the FORTH Standards Team in 1984.

The program described here is thought to meet the 1983 standard but no guarantee is made or implied to that fact. It is however, our intent to maintain and improve this program for at least the next five years. We would appreciate any errors being brought to our attention; especially useful would be appropriate means of correcting our error. We would also be interested in suggestions for extensions and improvements.

In the program described here James' original MACRO-11 version of FORTH² has been substantially revised to bring it into compliance with the 1983 standard of the language. Many of the changes are code implementations of a higher level program written by Vernor Vinge. The source

program is written in MACRO-11 and contains a number of options. The program can be compiled to run under RT-11, RSX-11M, RSTS/E. A command file to implement FORTH under each of these operating systems is given together with the source file on the RT-11, RSTS/E, and RSX-11 tapes.

WHAT IS FORTH ?

FORTH is an interpretive language generating threaded code and having compiler and operating system attributes. The system the user sees is interpretive. Thus commands, requests, etc. (WORDS in FORTH) are entered and executed. Numbers entered are placed on a stack for temporary storage until they are needed. Additionally the machine code for the action specified by a number of FORTH WORDS can be threaded together in a new WORD's definition in such a fashion that the interpretation of this new WORD causes immediate execution of this new code

FORTH is composed basically of a dictionary, a text string interpreter, and a compiler.


```

.
.
.
-----
| Dictionary Entry For |
| ENCLOSE              |
-----
LINK2 = .                ; START OF DICTIONARY
                        ; ENTRY FOR EMIT
.BYTE 204                ; LENGTH BYTE - SIGN
                        ; BIT SET
.ASCII ^EMIT^           ; THE NAME OF THE
                        ; FORTH WORD
.EVEN
.=.-1
BYTE 240                 ; LAST CHARACTER OF
                        ; NAME (OR BLANK
                        ; FILL) WITH HIGH
                        ; BIT SET
.WORD LINK              ; POINTER TO
                        ; BEGINNING OF
                        ; PREVIOUS
                        ; DICTIONARY HEADER
                        ; ( ENCLOSE HERE )
.LINK = LINK2           ; POINT LINK TO
                        ; BEGINNING OF THIS
                        ; WORD
PEMIT                   ; POINTER TO MACHINE
                        ; CODE FOR EMIT

```

```

-----
| Dictionary Entry For |
| KEY                  |
-----

```

```

-----
| Dictionary Entry For |
| KEY?                 |
-----

```

```

.
.
.
^ v ^ v ^ v ^ v ^ v ^ v ^ v ^ v ^ v ^ v ^
.
.
.
PEMIT:  CMP      (S),#40
        BLT      1$
        INC      42(U)
1$:     TST      @#177564
        BEQ      1$
        MOV      (S)+,@#177566
        EXIT    ; THIS IS A TWO INSTRUCTION
                ; MACRO WHICH TRANSFERS
                ; CONTROL FROM THE MACHINE
                ; CODE OF ONE DEFINITION TO
                ; THAT OF THE NEXT.
.
.
.

```

The operation of the text string interpreter is most easily shown with a trivial example. A one line terminal entry and its result is

```

42 EMIT <CR>
* OK

```

The text string interpreter breaks the string 42 (delineated by a following blank) out of the input stream and looks in the dictionary for its definition. Failure to find this string leads to an attempt (successful) to convert it to a number and place it on the stack (R5).

The text string interpreter next breaks the string EMIT (delineated this time by the end of the input stream) out of the input stream and looks in the dictionary for its definition. Success here leads to the execution of the word i.e., transfer to the code beginning at Pemit. In this case the result is the display of ASCII character 42 (*) on the terminal.

Successful interpretation of the entire input string is acknowledged by the printing of the string OK.

A search of the dictionary for a WORD (ASCII string) involves following a linked list of defined WORD strings until a match is found or the list terminates. Note in the dictionary example that LINK contains the address of the beginning of the last WORD defined. Note also that this address contains a length byte followed with that number of ASCII characters and a terminating last character or blank with the high bit set. The next memory location contains the address of the next to the last WORD defined and so on to the start of the dictionary. The beginning of the dictionary is flagged by LINK containing zero. The text string interpreter compares the ASCII string with the first string it encounters in the dictionary (that marked by LINK). A comparison failure leads to a comparison with the string pointed to by the link field of the last word defined and so on until a match is found or the list terminates.

THE COMPILER

The operation of the compiler is also most easily demonstrated with an example. A terminal entry with the resulting acknowledgment is shown as

```

: ASTERISK 42 EMIT ;<CR>
OK

```

The appearance of the string : in the input string puts FORTH in compiler mode and creates a dictionary header for the word which appears next in the input string

(ASTERISK here). The code created is that displayed below between the first appearance of LINK2 and DOCOL. Successful conversion of the next string as a number causes the address of a machine code routine LITER (from literal) to be placed next in the dictionary memory. The effect of LITER is to move the contents of the next memory location to the stack (R5). The string 42 is then converted to binary and stored in the following memory location. The string EMIT is found in the dictionary and the address of the start of the machine code for it is stored in the next memory location. The final string ; causes the entry for the control transfer routine EXIT to be stored in the next memory location and returns the mode to interpret.

The effect of the above entry on memory is shown as:

```
-----
| Dictionary Entry For |
| TASK                 |
|-----|
```

```
LINK2 = .
.BYTE    210
.ASCII  ^ASTERISK^
.EVEN
.=.-1
.BYTE    240
.WORD    LINK
LINK = LINK2
DOCOL    ; A POINTER TO FOUR MACHINE
          ; INSTRUCTIONS WHICH START THE
          ; EXECUTION OF THE WORD POINTED TO
          ; BY THE NEXT MEMORY LOCATION
          ; I.E., LITER
LITER    ; A POINTER TO CODE WHICH
          ; MOVES THE CONTENTS OF THE NEXT
          ; MEMORY LOCATION TO THE STACK
          ; (R5).
42.
PEMIT    ; A POINTER TO THE MACHINE
          ; CODE FOR EMIT
EXIT     ; THE USUAL EXIT ROUTINE
```

OPTIONS

Single Character Input

Single character input can place a significant load on the processor in a time sharing system and can degrade performance. A non FORTH-83 standard optional assembly can be made so that instead of one character, one line of characters is processed at a time. The nonstandard assembly is made by setting the variable LOADED to unity. If LOADED = 1 is commented out in the file FORTH.MAC before assembly the standard approach to character input is taken.

RSTS Emulation of RSX (KEY?)

The RSTS operating system does not support unsolicited I/O and as a result KEY? does not behave as required by the standard. The FORTH word KEY? looks to the input buffer and sets a flag if any character appears there. Under RSTS the input buffer is not

made available to the program until a carriage return is struck so two keys are needed for KEY?. The proper functioning of KEY? was thought so important that it was implemented in a modified form. A control C (^C) interrupt is available under RSTS. As implemented KEY? tests for ^C and sets a flag if ^C has been struck and otherwise clears the flag.

The Split File Structure (RSTS, RSX)

FORTH divides disk storage into 1024 byte BLOCKS. These blocks are subdivided for display into 16 lines of 64 characters each. Each BLOCK can be edited to define new words definitions for user applications as for example:

```
-----
SCREEN #1
0 ( LOAD SCREEN )   DECIMAL
1
2 1 WARNING ! ( GET ERROR MESSAGES NOT 3
3 NUMBERS )
4
5 : THRU ( <LO, HI> -- <>  LOADS FROM
6   SCREEN LO THRU SCREEN HI)
7   1+ SWAP DO 1 LOAD LOOP ;
8
9 ( .( LOADING ASSEMBLER ) 10 15 THRU )
10 CR
11 .( LOADING EDITOR ) 2 LOAD 40 57 THRU
12 ( THIS EDITOR IS THE WORK OF S. DANIEL.
13   SEE "The FORTH Inc., Line Editor" in
14   FORTH Dimensions, vol 3, #3. pg 80 )
15 CR 7 EMIT ( RING THE BELL)
-----
```

The BLOCK may also be used for the storage of binary data.

The split file structure described here is the creation of Vernor Vinge⁵. In a typical teaching situation the computer will be running under RSX or the RSTS emulation of RSX. In this case the compiled nucleus of FORTH (FORTH.TSK) together with the first 90 BLOCKS of disk storage (TEACHER.DAT) would reside under the instructor's project and programmer number. These files would have read/run or read only privileges granted to other users. BLOCKS 91 to 180 (STUDENT.DAT) would reside under the student's project and programmer number with full freedom for the user. The words servicing disk I/O check the project and programmer ID of the user. If this ID is that of the instructor, full read/write privileges are granted to the first 90 BLOCKS. If this ID is that of a student, only read access is granted. In either case full read/write access is granted to the BLOCKS from 91 through 181.

This limited access to the first 90 BLOCKS has great pedagogical value. It allows the instructor to edit this region so as to pass information and/or software to the entire student population without fear that it will be corrupted by anyone but himself.

Additionally, a student requires a minimum disk allocation, can use the full language and be given messages or software by the instructor.

GETTING STARTED

First get a copy of "Starting FORTH" by Leo Brodie⁴. This book is without a doubt the best introduction to the language in existence. Second, get a copy of this program from the RT-11, RSTS/E, or RSX-11 symposium tape, compile, load, and run the program on your system. Start at the beginning of Brodie's book working the examples and then solve the end of chapter problems. It is really just as easy as it sounds.

The more serious student would augment Brodie's book with membership in the FORTH Interest Group⁵ and study of its periodical "FORTH Dimensions."

FUTURE PROSPECTS

With the experience gained here we should be able to port the public domain version of F83 by Laxan and Perry^{6,7} to RT-11, RSTS/E, and RSX-11.

REFERENCES

- 1) FORTH Standards Team
PO Box 4545
Mountain View CA 94040
USA
- 2) J. James, "PDP-11 FORTH User's Guide",
January 1980.
- 3) Vernor Vinge
Department of Math Sciences
San Diego State University
San Diego, CA
and
"Teaching FORTH on a VAX", FORTH
Dimensions, IV, #6, 1983
- 4) Leo Brodie, "Starting FORTH",
Prentice-Hall Englewood Cliffs,
NJ 07632
- 5) FORTH Interest Group
P.O. Box 8231
San Jose, CA 95155
- 6) Henry Laxan and Michael Perry
No Visible Support Software
Box 1344, 2000 Center Street
Berkeley, CA 94074
- 7) C. H. Ting, "Inside F83", Offete
Enterprises, Inc.,
San Mateo, CA

SUPPLEMENTAL CONTROL ALGORITHMS TO AUGMENT PROPORTIONAL-INTEGRAL-DERIVATIVE CONTROL
OF THERMOCHEMICAL REACTION RATES

Wendel J. Shuely
Paul E. Field
Virginia Polytechnic Institute
Blacksburg, VA 24061

The thermal disposal of hazardous compounds is being studied using a new approach for measuring the rates of thermal weight-loss reactions. A dynamic thermogravimetric method has been developed for scanning the temperature applied to a pesticide sample under investigation. The computer-controlled, closed-loop experiment maintains the reaction rate at a prespecified value to obtain numerous advantages in the kinetic measurement process. The software can be generalized to any closed-loop experiment in which the dependent and independent variables have been reversed to obtain improvements in accuracy, precision, analysis time, or other measurement advantages. Adequate control for this experiment type can be obtained with a software proportional-integral-derivative (PID) algorithm. Further advantages can be derived from a more complex, dynamic experiment in which all variables are changing continuously, in addition to the reversal of the dependent and independent variables. Dynamic, closed-loop control experiments require supplemental control modes to augment the PID algorithm. Parameter scan control, coast rate limit, and other modes were programmed. SYSTEM: PDP11, RT11, FORTRAN and MACRO.

INTRODUCTION

The supplemental control modes developed for closed-loop thermal reaction rate control should be useful for many related control processes. In addition, the use of a single-user, real time operating system (RT-11) and 16 bit CPU (PDP-11) is in line with the trend toward use of direct digital control (DDC) implemented on a dedicated computer.

The application of supplemental control modes in addition to proportional-integral-derivative (PID) control can be described by considering the following topics:

1. A brief review of the general control problem and a specific thermochemical reaction control example.
2. The relationship between reaction rate and heater control temperature, that is, between the controlled variable and the controlling parameter.
3. The flow diagram for the PID and supplemental control code.

4. The process control functions implemented in the code with definition of terms.

5. Examples of program module code.

6. Examples of experimental results showing successive improvement of control with augmentation of PID and supplemental control modes.

The scope of the research considered here is limited to the augmentation of PID control with additional interacting control modes. Descriptions of the reactions, computational kinetics, data acquisition, signal processing, and applications to hazardous compounds are not included; some of this research has been described elsewhere. (1, 2, 3, 4)

MEASUREMENT THEORY

The limitations of the conventional thermogravimetric (TGA) technique and advantages of the computer-controlled closed-loop method can be outlined by a comparison of their respective weight-loss vs time thermograms. Figures

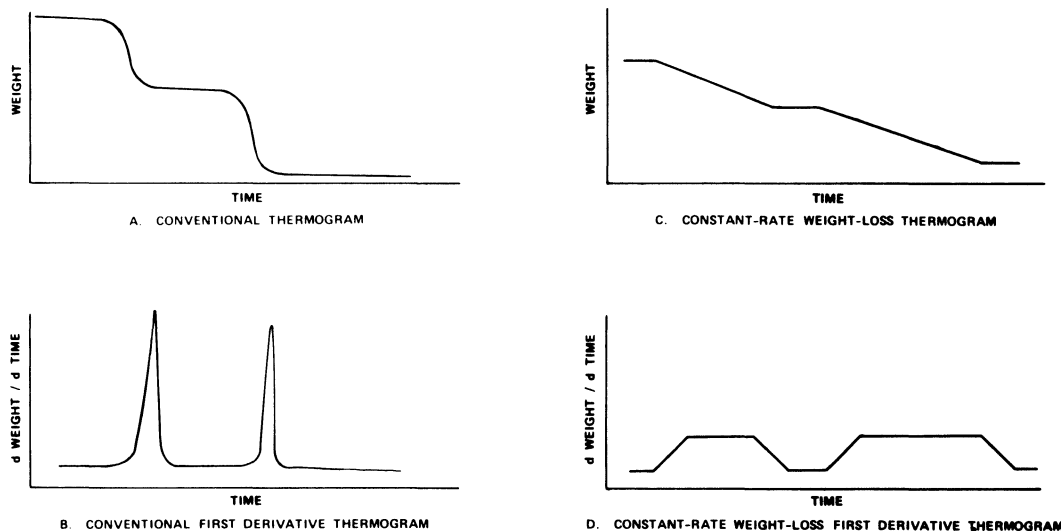
1A and 1B present idealized thermograms from a conventional TGA experiment in which the temperature is programmed at a linear rate. Two consecutive weight-loss reactions are shown in Figure 1A and it can be seen that weight-loss is initially slow, is followed by a rapid weight-loss indicated by the steep almost vertical slope, and is again slow at the end of the reaction. This wide range of rates can also be seen by inspecting Figure 1B where the change in weight-loss with time (derivative) is plotted against time. These peak-shaped curves show that the weight-loss rate increases rapidly, passes through a maximum, and decreases rapidly. The measurement of temperature and weight is attempted under these highly nonequilibrium conditions and under these variable thermal and mass fluxes. This results in a decrease in discrimination between consecutive reactions and a decrease in data accuracy. An equally important limitation results because for differential computational methods, kinetic parameters must be calculated by taking the derivative along the steep weight-loss slopes that are produced when

the sample is submitted to a conventional linear temperature ramp.

These limitations were overcome by the development of a system in which the computer directly interacted with the thermal chemical process. In concept, the weight-loss rate is converted to the independent variable and the computer controls the temperature scan to provide the requisite, complex, non-linear profile that maintains the set rate. In essence, the independent and dependent variables have been interchanged and this produces improved experimental conditions for measurement of reaction rate. Figure 1C shows that the idealized computer-controlled set rate is a straight diagonal line. This rate is at a favorable velocity, neither too slow nor too rapid, for accurate calculation of the derivative. On a microscopic level, the thermal flux into, and gaseous mass flux out of the sample take place at a slow, even rate. Figure 1D presents the derivative of the constant weight-loss rate which has a square wave shape, rather than the peak in Figure 1B.

FIGURE 1

IDEALIZED THERMOGRAMS AND THEIR FIRST DERIVATIVES



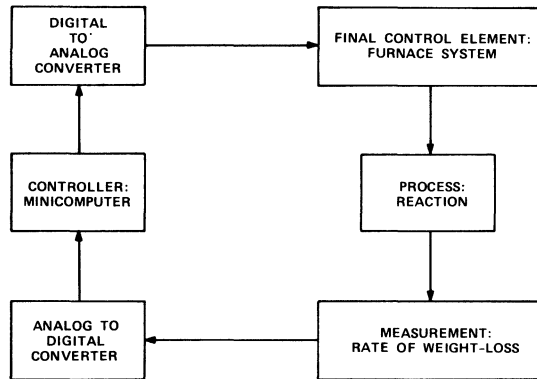
CONTROL THEORY

A block diagram of the closed-loop control system is shown in Figure 2. The process is a reaction taking place in a sample cell of the thermogravimetric system (mid-right block in Figure 2). Temperature and weight transducers convert the physical processes to electronic

signals (lower right block). The signals are converted to digital data and employed by the minicomputer controller to compute an appropriate digital control word. This word is converted to an analog signal interfaced to the control element, which adjusts the heater and temperature of the process reaction.

FIGURE 2

BLOCK DIAGRAM OF A DIRECT DIGITAL PROCESS CONTROL LOOP REPRESENTING THE MINICOMPUTER-CONTROLLED THERMOGRAVIMETRIC SYSTEM



The idealized relationship between reaction rate (controlled variable), heater control wattage (controlling parameter), and temperature (secondary controlling parameter) are shown on a common axis in Appendix A. The lower figure displays a typical weight-loss reaction rate shown as the plateau (compressed on the time scale). Typical relative values of the onset of the proportional derivative, integral, and coast bands are also shown. These times are propagated vertically on the x-axis to aid in tracking the interaction between the process controlling parameter (temperature/heat) and the controlled variable (reaction rate).

The rather subtle changes shown in the middle and upper curves of heater output and temperature are necessary to guide the very sensitive reaction rate to a plateau without overshoot. These criteria points are discussed in more detail below.

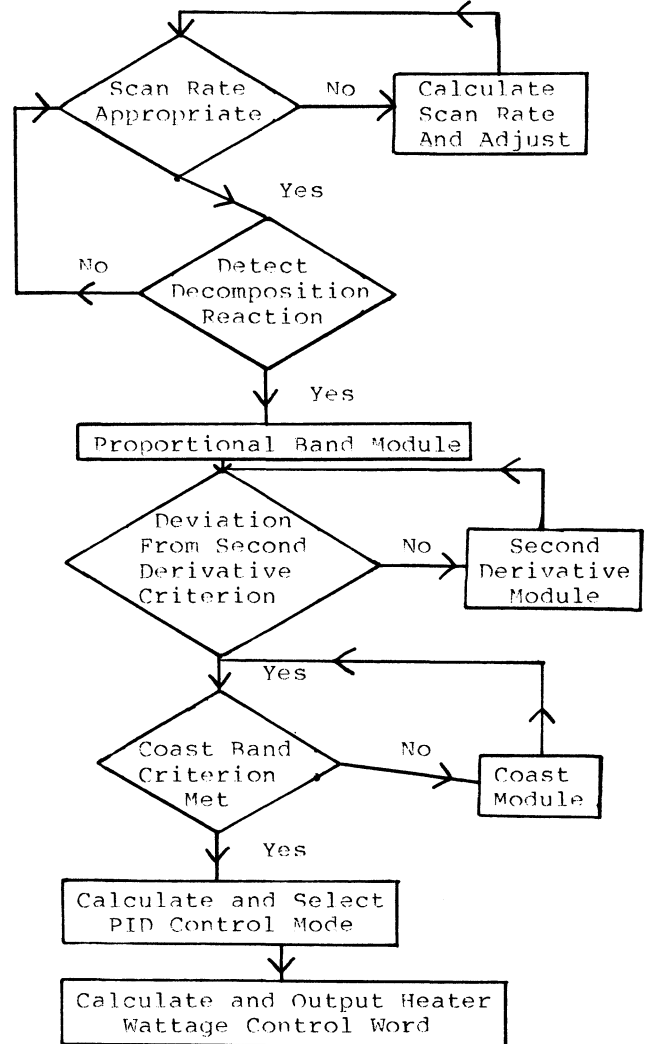
Note that in the later stages, the hyperbolic temperature profile is required to drive the reaction to completion for this hypothetical, first order reaction curve.

CONTROL ALGORITHM

A simple flow chart for the PID and supplemental code is shown in Figure 3. Ten control modules are displayed and these can be referenced to events on the previous figure displaying reaction-heater-temperature interactions.

Note that the system is unusual in that there is no pre-knowledge required (or often available) regarding the temperature at which a measurable rate occurs or at which the constant-rate set point criterion will be met.

Figure 3. General Flow Chart of Proportional-Integral-Derivative and Supplemental Control Modules in Algorithm.



The sequence of events leading to a smooth approach to the desired set point and reaction rate can be understood by viewing the display of the controlled variable (reaction rate and the controlling parameters (temperature and heater wattage) as a function of time (Appendix A and Figure 3). After a reaction is detected during the rapid temperature scan, classical proportional-derivative (PD) control is asserted with a reduced full scale heater wattage. Too rapid an approach toward the set point is indicated (Appendix A) by an excursion of the reaction rate signal out of the second derivative band. This band is a conical shaped area radiating upward toward the desired reaction rate from the point of assertion of the proportional band.

The heater wattage is reduced until the rate of approach to the specified reaction rate reaches the lower limit of the second derivative band.

The integral band criterion is met as the signal approaches set-point. As the signal is driven toward the set-point asymptote, the coast criterion will be encountered and the heater wattage is reduced for an interval in order to equilibrate the sample at the set point or specified reaction rate.

Appendix B provides a listing of control algorithm parameters that have been grouped according to computational logic and process control function. The first column describes the general process control function while the second column gives the specific function in the reaction rate control example. The third and fourth columns list the FORTRAN code and data type.

The control parameters are initially divided into variables and constants. The variables consist of four types: (1) the computed process control parameters that are updated each program cycle, (2) time counters that are converted into elapsed times for comparison with criteria times for coasting (with zero value of the controlling parameter), (3) event counters, that monitor consecutive entries into a specific control module for tracking performance of the algorithm, and, (4) flags, which are used to route program flow through appropriate modules, depending on events in previous program cycles.

The constants are control parameters, gain multipliers, or criteria values. They are "constant" for a single experiment or batch process in the sense that adaptive programming is not yet used to vary their values with changes in process load or set point. However, they may be changed during the course of the experiment under operator control for performance tuning or response-time measurements.

There are three full scale values of the control parameters representing different ranges of the full scale value of the controlling parameter (heater wattage or temperature). These are (1) a zero value, (2) an intermediate value for use near the set point, and (3) a high value used in the initial rapid temperature scan in search of a reaction. This program selection of the full scale value might be considered a primitive adaptive function.

The three gain factors are the classical proportionality constants in the PID equations. Note that the proportional band gain is derived from the onset value of proportional control in the usual manner.

Several criteria values are listed and these are employed in IF statement decisions governing program control. Most of these can be considered as initiating or terminating the action of specific control program modules.

The onset of proportional and integral control are initiated at BANDP and BANDI. The first program cycle encountering the reaction set point coast band is triggered at BANDC. The slope band of the allowable values of the second derivative are defined in TSTMAX and TSTMIN. The maximum time interval for maintaining zero values of the controlling parameter are defined in ISOTST for the initial coast module and IDITST for the second derivative band module.

RESULTS

The influence of the supplemental control modes on the performance of the control algorithm can be observed by inspection of an extensive sequence of time vs weight-loss derivative plots. Two examples are provided in Figures 4 and 5.

The examples from early versions of the control algorithm, without supplemental control, are omitted. While these provide a dramatic contrast to the well behaved control examples, the plots can be easily described as showing severe overshoot of the derivative with slow recovery to the set-point, recovery occurring either before or near the end of the reaction.

The examples show aspects of tuning the criterion constants and band levels; the effects of a range of values for tuning the algorithm can show, at the same time, the residual overshoot behavior or other performance problems when the supplemental modes are not available. Figure 4 shows the results of a relatively well tuned set of control constants. The time vs derivative curve almost matches the shape of the idealized

curve in Figure 1D. The temperature vs time trace shows an extremely rapid temperature scan (20 to 120 degrees C in 5 minutes) that is very quickly converted to closed-loop control of reaction rate. This example represents close to the limit of temperature scan speed without derivative overshoot. Note that the effect of the second derivative limits on rate-of-approach to the set point has resulted in a temperature decrease during the second 5 minutes of the closed-loop experiment. Likewise, the onset of the coast module causes a barely perceptible hesitation of the derivative approach to the 0.2 mg/min set-point. This hesitation can be seen at about 60% of set-point or at coordinates of 10 minutes and 1.2 mg/min. This example is especially interesting in that overshoot of the reaction rate derivative is avoided but temperature overshoot occurs with respect

to transition into a smoothly increasing time vs temperature profile. The kinetic data obtained during the period of declining temperature will be invalid, however, the later 90% of the reaction data is measured at the computer-controlled pre-set rate of 0.2 mg/min. Therefore, an operator selected set of software control constants can optimize an experiment for extremely rapid temperature scanned search for a reaction and obtain kinetic data at over 90% of conversion. Figure 5 show another extreme of control behavior. Another set of software constants can provide a slower computer-controlled search for a weight-loss reaction. A benefit of this option is a gradual approach to set point and the acquisition of valid initial rate data at close to only 1% of conversion. Figure 5 shows this slow, controlled rate of approach to set point.

FIGURE 4

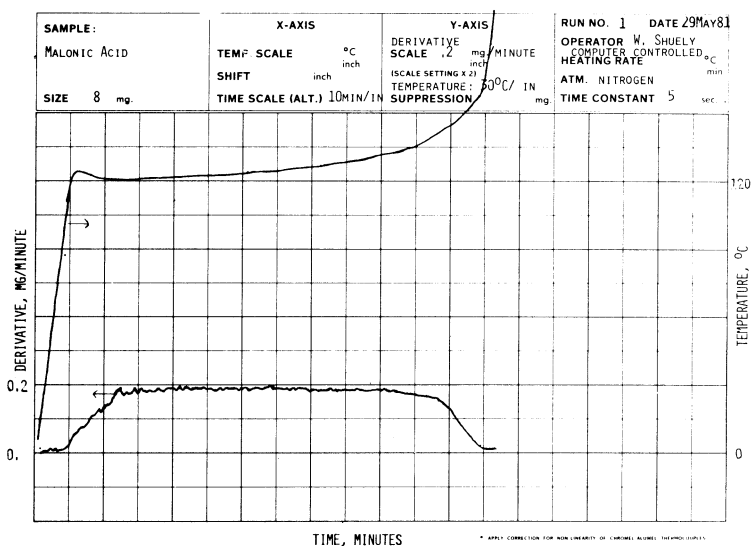
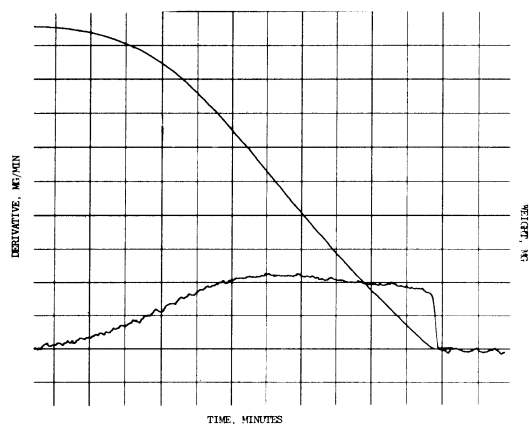


FIGURE 5



DISCUSSION OF SYSTEM PERFORMANCE

System performance here includes adequacy of the hardware/operating system/language and not the application code.

Provisions were made in control program design to monitor and compensate for excessive control lag due to the time required for control computations and decisions. The timing of the control computations could not be limited or defined explicitly since decisions and routing through modules were influenced by program response to a chemical process in a closed-loop system.

The array of design considerations employed need be mentioned only briefly since the results showed the 16 bit CPU (PDP-11/10 or 24), real-time operating system (RT11SJ), and FORTRAN/MACRO combination was adequately fast. The control computations were completed before the end of a data acquisition cycle for all sets of control parameters at all degrees-of-conversion of reactant.

Other applications or higher set points (reaction rates) may require more rapid cycle times; program design features to accommodate these cases are noted below. The data acquisition is interrupt driven and the control computations will be suspended as needed. After resuming, the control word will be computed based on obsolete data. However, the next control computation always employs the most recently acquired filtered datum.

Filtering of acquired data can be performed at three different points in the program. Two are within the MACRO subroutine and one within the FORTRAN control code. Immediate consecutive filtering can be provided by summing into a register or memory location from each A-to-D channel (MACRO). Fixed interval, channel sequencing, sampling, and summation are also available (MACRO). The FORTRAN code allows averaging over a selected number of the latest data before using the point to calculate a deviation from set point. All filters can be used simultaneously with all available options in terms of sample number and period.

The degree of control and options available for filtering allow minimization of sampling when process-to-control time is unfavorable. Editing modules combine and resolve all filter timing offsets and the real time computations of integral and derivative functions are performed with exact rate-time data pairs.

The control program was designed to trace separate module usage and timing requirements. The system of integer counters and flags described in Appendix B were employed to monitor program flow and debug the program. These flags and counters were useful in tuning the control

constants: they were not needed to optimize timing considerations but could be used to do so for alternate applications.

CONCLUSIONS

The reaction rate control process presents several unique and difficult problems. Software based control procedures have the flexibility and computational/logical power to provide solutions to these problems.

The control process demonstrates a relatively high process load during certain parts of the process and abruptly lower process load at the reaction temperature. Primitive adaptive programming of appropriate full scale output level solves this abrupt load change problem.

A critical transition occurs from the detection of a reaction to attainment of a controlled, constant rate process. This transition is successfully achieved by the second derivative band and coast-to-asymptote modes.

A smooth temperature-weight synchronization can be lost by exertion of frequent and abrupt control parameter changes; however, this early rate data is important because of the frequent use of initial rates in kinetic analysis. The control programs capability to provide for selection of delayed onset criteria exertion of control modes provides initial rate data without allowing overshoot.

The speed of the RT-11 FORTRAN system with successive approximation ADC (AR-11) is adequate for complex control problems and research on modification and augmentation of classical PID control modes. The output control can be implemented with FORTRAN IPEEK/IPOKE functions. It now appears that the input or data acquisition may also be (but was not) implemented with IPEEK/IPOKE; this is partially due to the relatively favorable process lag to control lag time ratio that was an inherent benefit of the initial objective of attaining a slow constant rate process.

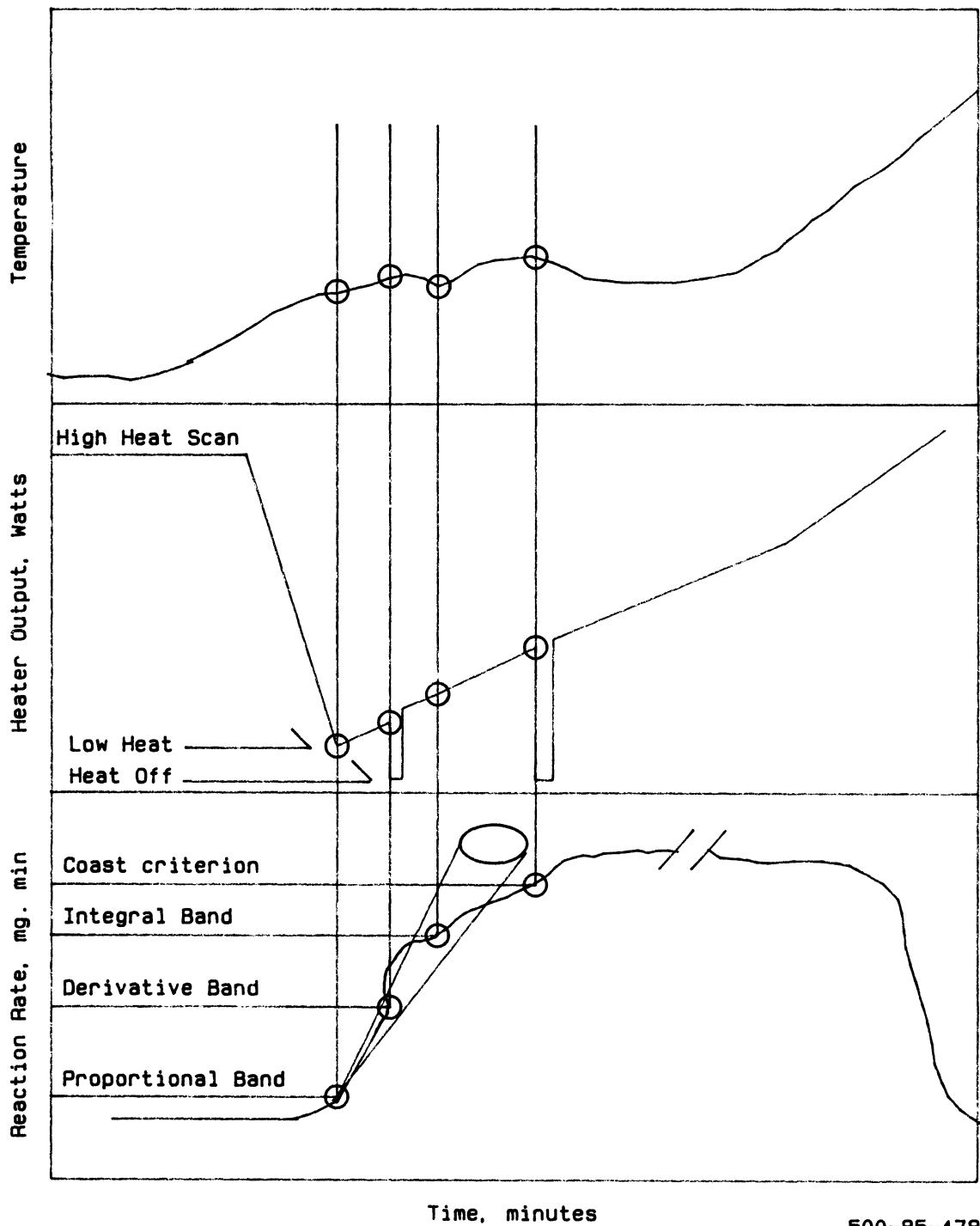
LITERATURE CITED

1. Shuely, Wendel J. ARCSL-TM-79003. Novel Thermogravimetric Instrumentation and the Advantages of Interactive Computer Control. December 1978.
2. Shuely, Wendel J. Computerized Thermogravimetric System for the Study of the Decomposition Kinetics of Liquids. ABS Pap ACS 1979 (Apr): COMP 18, 1979.
3. Shuely, Wendel J. and Field, Paul F. Computer-Control of Thermal Weight-Loss Reactions to Improve the Kinetic

Measurement Process. International Union of Pure and Applied Chemistry. Sixth International Conference on Computers in Chemical Research and Education. Georgetown University, Washington, DC. July 1982.

4. Shuely, Wendel J. and Field, Paul E. Computer-Controlled Instrumentation for Investigating Thermal Reactions for Disposal of Surplus Hazardous Chemicals. American Chemical Society Extended Abstract. 1982 (Sep): ENVIR: 9, 1981.

Appendix A. Idealized Relationships between Reaction Rate, Heater Control Wattage, and Temperature as a Function of Time for Typical Parameters of the Computer Control Algorithm



500-85-178

APPENDIX B. Control Algorithm Parameters Classified in Terms of
Process Control Function and Computational Logic

VARIABLES:

FUNDAMENTAL PROCESS VARIABLES:

Controlled Variable	Reaction Rate	DIFNEW	F
Controlled Variable	Second Derivative	DIFDIF	F
Controlling Parameter	Heat/Temperature	NHEAT	I

TIME COUNTERS:

Controlling Parameter Zero	Initial Coast	KOTIME	I
	Outside Second Derivative Band	IDTIME	I

EVENT COUNTERS:

Algorithm performance monitoring; counts number of times a control module is entered.	Onset Coast	KNTISO	I
	Overshoot	KNTMAX	I
	Second deriva- tive coast	KNTMIN	I

FLAGS:

Route control compu- tations through modules	Onset coast	KOSTFG	I
	Second deriva- tive	IDIFFG	I

CONSTANTS:

Control Words	Heater Off	NOHEAT	I
	Wattage Low	LOHEAT	I
	Ranges High	HIHEAT	I
Multipliers; gain factors	Integral	GAINJ	F
	Derivative	GAIND	F
	Proportional (100/PBAND)	GAINP	F

CRITERIA:

Onset of control mode: (percent of full scale)	Proportional	BANDP	F
	Integral	BANDI	F
	Coast	BANDC	F
	Derivative (maximum)	TSTMAX	F
	Derivative (minimum)	TSTMIN	F
	Onset Coast	ISOTST	I
Derivative Coast	IDITST	I	

