

→ COMP ROOM
UPSTAIRS



INTEREX
HP TECHNICAL
CONFERENCE
SAN JOSE
1 9 8 7

PROCEEDINGS

SAN JOSE
OCTOBER 18-22, 1987

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.



INTEREX

the International Association of
Hewlett-Packard Computer Users

Proceedings

of the

1987 Conference of
HP Technical Computer Users

at

San Jose, California
October 18-22, 1987

F. Stephen Gauss, Editor

The Paper Review Committee

Wayne Asp.....Hewlett-Packard Co
Tony Brown.....Bobier Tool Supply
John Campbell.....Campbell Computer Consulting
Dean Clamons.....Naval Research Laboratory
Stephen Gauss, Chairman...US Naval Observatory
Art Gentry.....AT&T Communications
Paul Gerwitz.....Eastman Kodak Co
Hugh Hanks Jr.....DOD
Mark Katz.....Graphicus
Jock McFarlane.....RCA Laboratories
Chris Pappagianis.....RCA Corp
Terie Robinson.....Hewlett-Packard Co
Larry Rosenblum.....Naval Research Laboratory
Nick Seidenman.....McDonnell-Douglas PSC
William Steele.....Tobacco Institute Testing Lab
Dan Steiger.....Naval Research Laboratory
Steven Telford.....Lawrence Livermore Laboratory
Alan Tibbetts.....Telos Computing
Don Wright.....Interactive Computer Technology

Introduction

This volume of the Proceedings of the INTEREX 1987 Technical Computer Conference was printed from camera-ready copy supplied by the authors. Due to the proliferation of word processors for HP and other computers, it was deemed appropriate to request that the authors format and print their own papers, rather than submitting them in machine-readable form, as in the past. It was gratifying to the editor to find that all of the authors were able to meet this requirement, thus saving him several months of work. Papers have been numbered sequentially in order of presentation at the conference with HP1000 papers numbered 10xx and HP9000 papers numbered 90xx. Papers based on the tutorials are numbered with a T designator and appear at the beginning of this volume. Several papers will be of interest to both communities, especially as the 1000 and 9000 lines merge at the high end. Because the tutorials often include a considerable amount of last minute information, as well as requiring much more work than a paper, it is not practical to insist that the authors prepare papers in time for the proceedings. Nevertheless, some of the tutorials are represented in this volume and their authors are to be commended for their efforts.

Thanks go to the authors who met the submission requirements and had their papers in by the deadline. The quality of the papers seems to be higher this year than ever before. Thanks also to the paper review committee for their timely responses and helpful comments.

Finally, thanks to my wife, Vivian, and to my employers, for their continuing support of my activities on behalf of the INTEREX conferences.

*F. Stephen Gauss
U. S. Naval Observatory
Washington, D.C.
1 August 1987*

Index By Author

Araujo, Argemiro Rodriguez, Occidental de Columbia.	9003
HP9000-IBM4381 Communications	
Ashby, T.G., R.J. Reynolds Tobacco Co.1018
A Real-Time Mini-Computer System For Automating Radiolabeled Xenobiotic Disposition Studies	
Asp, Wayne, Hewlett-Packard.1029
Effective Usage of EMA/VMA in FORTRAN Programs	
Bergman, Carol Hubecka, Hewlett-Packard Co.	1026
Disc Performance	
Bomgardner, Mark, Hewlett-Packard Co.	T1
RTE Performance Tuning	
Burch, Carl, Hewlett-Packard Co.T3
FORTRAN Standards	
Burchette, K.S., R.J. Reynolds Tobacco Co.1018
A Real-Time Mini-Computer System For Automating Radiolabeled Xenobiotic Disposition Studies	
Carson, Bill, Graphics.1006
Graphical Techniques In Data Analysis	
Carter, Stephen, Eyring Research Institute.	1003
Relational View of Image With Real Zip	
Chase, Tim, Corporate Computer Systems.	1030
EMA and the C Programming Language	
Christ, Phil, Hewlett-Packard Co.	1017
Manufacturing National Account Products Improve Manufacturers' Productivity	
Clucas, GERALYN, Graphics.	1005
Elements Of Good Graphing Techniques	
Davis, Charlie, Knight-Ridder Trade Center.	9005
Data Collection And Management System For On-line Real Time Financial Market Information	
deBethizy, J.D., R.J. Reynolds Tobacco Co.1018
A Real-Time Mini-Computer System For Automating Radiolabeled Xenobiotic Disposition Studies	
Drotning, William, Sandia National Laboratories.1014
A Data Analysis Environment For HP 1000 Computers	
Felman, Dan, Hewlett-Packard Co.1028
An Implementation Of LISP	
Fullerton, Steve, Statware.	T2b
PORT/HP-UX vs. Native Mode Migration	
Fullerton, Steve, Statware.	9004
Portable Tape Handling For HP-UX	
Gauss, F. Stephen, U.S.Naval Observatory.	1019
A Word Processing System For the HP1000	
Gerwitz, Paul, Eastman Kodak Co.1013
A Database Manager Subsystem For Image/1000	
Glover, David, Hewlett-Packard Co.T1
RTE Performance Tuning	

Goff, Thomas, NASA Goddard SFC.	1031
GIMMS Interactive Mapping/Image Processing	
Gregg, A. D., City Computing Ltd.	1024
A Modular Multiprocessor Simulation Philosophy	
Hanks, Jr., Hugh C., DOD.	1033
Data Acquisition At High Clock Rates	
Hellard, T.J., R.J. Reynolds Tobacco Co.	1018
A Real-Time Mini-Computer System For Automating Radiolabeled Xenobiotic Disposition Studies	
Johnson, John, Hewlett-Packard Co.	T5
The Ada/1000 Compiler and RTE-A Ada Run Time Support	
Johnson, M.B., R.J. Reynolds Tobacco Co.	1018
A Real-Time Mini-Computer System For Automating Radiolabeled Xenobiotic Disposition Studies	
Jones, Tony, Hewlett-Packard Co.	1015
HP1000/MEF to HP9000/500 Interface	
Josal, Todd, Naval Undersea Warfare Eng. Station.	1023
Monitor and Control Of Test Software Executing On A Remote Dissimilar System	
Kalb, Virginia, NASA Goddard SFC.	1031
GIMMS Interactive Mapping/Image Processing	
Klier, Pat, Eastman Kodak Co.	1013
A Database Manager Subsystem For Image/1000	
Klimpke, C.M.H., City Computing Ltd.	1024
A Modular Multiprocessor Simulation Philosophy	
Kozuka, Masataka, Nippon System Gijutu Co.	1027
Random Vibration Control Program	
Lawson, Greg, Advanced Microsolutions.	9001
Statistical Process Control - A Practical Approach To Increased Manufacturing Productivity	
Leslie, Donald, Raytheon.	1001
A Table Driven Plot Program For Radar Data	
McCormick, Ann D., Intelsat.	1016
Developing A Complex Engineering Test Database	
McDorman, Dave, Hewlett-Packard Co.	9007
Computer-Aided Test Software: Do You Want To Do A Better Job?	
McFarlane, Jock, David Sarnoff Research Center.	1021
GOSPEL: Generating Organic Structures for Printing Entirely On A LaserJet	
McInnes, Marvin, Consultant.	1012
IMAGE/1000: Secrets HP Never Told You	
McNabb, G., Naval Undersea Warfare Eng. Station.	1023
Monitor and Control Of Test Software Executing On A Remote Dissimilar System	
Miranian, Mihran, US Naval Observatory.	1032
Remote Monitoring and Control of Timing Equipment For Navigation Systems	
Mooney, Lori, Advanced Microsolutions.	9002
Distributed Database Management For Manufacturing Automation	
Nelson, Christopher, General Foods Corp.	1002
A Suite Of System Generation Tools	
Pavlinik, Ed, Hewlett-Packard Co.	1022
Optical Solutions To Mass Storage User Needs	
Pfeiffer, Richard, General Electric.	T2a
Porting A Real-Time Package To HP-UX	
Portessi, Peter, Cameo Systems.	9006
Computer Aided Manufacturing Using Relational Database Technology	

Reeves, Alistair, Telesat Canada.	1004
Implementing A Centralized Plotting And Slide Generation Facility For Hewlett-Packard Computers	
Robinson, Teric, Hewlett-Packard Co.9007
Computer-Aided Test Software: Do You Want To Do A Better Job?	
Ryan, Robert, Hewlett-Packard.T2a
Porting A Real-Time Package To HP-UX	
Sayed, Husni, IEM Inc.1034
An Introduction To Optical Disk Technology	
Scadina, Russ, Hewlett-Packard Co.1008
HP1000 A-series Crash Dump Analyzer	
Schober, Dan, Graphicus.1005
Elements Of Good Graphing Techniques	
Schumann, Paul, E-Systems.1011
Some Advanced Software Techniques	
Seidenman, Nick, McDonnell-Douglas PSC.	1009
Implementing High-Level Control Structures Using MACRO/1000 Macros	
Stass, Nancy, Telesat Canada.	1004
Implementing A Centralized Plotting And Slide Generation Facility For Hewlett-Packard Computers	
Suzuki, Minoru, Nippon System Gijutu Co.1027
Random Vibration Control Program	
Tibbetts, Alan, Hewlett-Packard Co.	T4
Unsupported Utilities For RTE	
Tibbetts, Alan, Telos Consulting.	1010
Using the HPCRT Library	
Vannicola, Francine, US Naval Observatory.1032
Remote Monitoring and Control of Timing Equipment For Navigation Systems	
Vogelsberg, Gary, Hewlett-Packard Co.	1025
Disc Interfaces For HP Systems	
Webb, P. J., Admiralty Research Establishment.1024
A Modular Multiprocessor Simulation Philosophy	
Whitney, Alan, MIT Haystack Observatory.1020
T _E X On the HP1000 and Laserjet+	
Wood, R., Telesat Canada.	1004
Implementing A Centralized Plotting And Slide Generation Facility For Hewlett-Packard Computers	



Index By Title

A Data Analysis Environment For HP 1000 Computers.1014
Drotning, William, Sandia National Laboratories	
A Database Manager Subsystem For Image/1000.1013
Gerwitz, Paul, Eastman Kodak Co.	
Klier, Pat, Eastman Kodak Co.	
A Modular Multiprocessor Simulation Philosophy.	1024
Gregg, A. D., City Computing Ltd.	
Klimpke, C.M.H., City Computing Ltd.	
Webb, P. J., Admiralty Research Establishment	
A Real-Time Mini-Computer System For Automating Radiolabeled Xenobiotic Disposition Studies.	1018
Ashby, T.G., R.J. Reynolds Tobacco Co.	
Burchette, K.S., R.J. Reynolds Tobacco Co.	
deBethizy, J.D., R.J. Reynolds Tobacco Co.	
Hellard, T.J., R.J. Reynolds Tobacco Co.	
Johnson, M.B., R.J. Reynolds Tobacco Co.	
A Suite Of System Generation Tools.	1002
Nelson, Christopher, General Foods Corp.	
A Table Driven Plot Program For Radar Data.	1001
Leslie, Donald, Raytheon	
A Word Processing System For the HP1000.1019
Gauss, F. Stephen, U. S. Naval Observatory	
An Implementation Of LISP.1028
Felman, Dan, Hewlett-Packard Co.	
An Introduction To Optical Disk Technology.	1034
Sayed, Husni, IEM Inc	
Computer Aided Manufacturing Using Relational Database Technology.	9006
Portessi, Peter, Cameo Systems	
Computer-Aided Test Software: Do You Want To Do A Better Job?.9007
McDorman, Dave, Hewlett-Packard Co.	
Robinson, Terie, Hewlett-Packard Co.	
Data Acquisition At High Clock Rates.	1033
Hanks, Jr., Hugh C., DOD	
Data Collection And Management System For On-line Real Time Financial Market Information.9005
Davis, Charlie, Knight-Ridder Trade Center	
Developing A Complex Engineering Test Database.	1016
McCormick, Ann D., Intelsat	
Disc Interfaces For HP Systems.	1025
Vogelsberg, Gary, Hewlett-Packard Co.	
Disc Performance.	1026
Bergman, Carol Hubecka, Hewlett-Packard Co.	
Distributed Database Management For Manufacturing Automation.	9002
Mooney, Lori, Advanced Microsolutions	
EMA and the C Programming Language.	1030
Chase, Tim, Corporate Computer Systems	

Effective Usage of EMA/VMA in FORTRAN Programs.	1029
Asp, Wayne, Hewlett-Packard	
Elements Of Good Graphing Techniques.	1005
Clucas, GERALYN, Graphicus	
Schober, Dan, Graphicus	
FORTRAN Standards.	T3
Burch, Carl, Hewlett-Packard Co.	
GIMMS Interactive Mapping/Image Processing.	1031
Goff, Thomas, NASA Goddard SFC	
Kalb, Virginia, NASA Goddard SFC	
GOSPEL: Generating Organic Structures for Printing Entirely On A LaserJet.	1021
McFarlane, Jock, David Sarnoff Research Center	
Graphical Techniques In Data Analysis.	1006
Carson, Bill, Graphicus	
HP1000 A-series Crash Dump Analyzer.	1008
Scadina, Russ, Hewlett-Packard Co.	
HP1000/MEF to HP9000/500 Interface.	1015
Jones, Tony, Hewlett-Packard Co.	
HP9000-IBM4381 Communications.	9003
Araujo, Argemiro Rodriguez, Occidental de Columbia	
IMAGE/1000: Secrets HP Never Told You.	1012
McInnes, Marvin, Consultant	
Implementing A Centralized Plotting And Slide Generation Facility For Hewlett-Packard Computers.	1004
Reeves, Alistair, Telesat Canada	
Stass, Nancy, Telesat Canada	
Wood, R., Telesat Canada	
Implementing High-Level Control Structures Using MACRO/1000 Macros.	1009
Seidenman, Nick, McDonnell-Douglas PSC	
Manufacturing National Account Products Improve Manufacturers' Productivity.	1017
Christ, Phil, Hewlett-Packard Co.	
Monitor and Control Of Test Software Executing On A Remote Dissimilar System.	1023
Josal, Todd, Naval Undersea Warfare Engineering Station	
McNabb, G., Naval Undersea Warfare Engineering Station	
Optical Solutions To Mass Storage User Needs.	1022
Pavlinik, Ed, Hewlett-Packard Co.	
PORT/HP-UX vs. Native Mode Migration.	T2b
Fullerton, Steve, Statware	
Portable Tape Handling For HP-UX.	9004
Fullerton, Steve, Statware	
Porting A Real-Time Package To HP-UX.	T2a
Pfeiffer, Richard, General Electric	
Ryan, Robert, Hewlett-Packard	
RTE Performance Tuning.	T1
Bomgardner, Mark, Hewlett-Packard Co.	
Glover, David, Hewlett-Packard Co.	
Random Vibration Control Program.	1027
Kozuka, Masataka, Nippon System Gijutu Co.	
Suzuki, Minoru, Nippon System Gijutu Co.	
Relational View of Image With Real Zip.	1003
Carter, Stephen, Eyring Research Institute	

Remote Monitoring and Control of Timing Equipment For Navigation Systems.	1032
Miranian, Mihran, US Naval Observatory	
Vannicola, Francine, US Naval Observatory	
Some Advanced Software Techniques.1011
Schumann, Paul, E-Systems	
Statistical Process Control - A Practical Approach	
To Increased Manufacturing Productivity.	9001
Lawson, Greg, Advanced Microsolutions	
T _E X On the HP1000 and Laserjet+.1020
Whitney, Alan, MIT Haystack Observatory	
The Ada/1000 Compiler and RTE-A Ada Run Time Support.	T5
Johnson, John, Hewlett-Packard Co.	
Unsupported Utilities For RTE.T4
Tibbetts, Alan, Hewlett-Packard Co.	
Using the HPCRT Library.1010
Tibbetts, Alan, Telos Consulting	

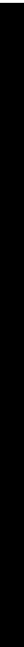


RTE Performance Tuning

**Dave Glover
Mark Bomgardner
Hewlett-Packard Co.
1266 Kifer Road
Sunnyvale, California 94086**

With the introduction of the SNAPSHOT/1000 Performance Analysis Service we have had the opportunity to investigate performance problems with several of our customers' systems. by using the data collection tools created by DSD for RTE-A and using the data reduction tools created by ITG, we have been able to identify problems and use those tools to help recommend a solution.

This presentation will briefly cover the process of collecting and reducing performance data and will identify the issues concerned with how to collect that data. The next step is trying to identify if a performance problem even exists with the system under analysis. A problem can mask itself in many different ways, where one area may be fixed by increasing a resource, another problem area may intensify its impact on the system. Several examples of systems where problems were easily identified and also those where the problems were not so obvious will be discussed.



FORTRAN Standards

Carl Burch
Hewlett-Packard Co.
Sunnyvale, California 94086

The Draft Standard for the next revision of the ANSI Fortran standard is about to become available for public review. Hewlett-Packard representatives have played an important role in its development. The new revision, generally referred to as Fortran 8x, makes significant additions to FORTRAN 77 in five areas:

- define array-level operations to efficiently utilize vector processing hardware and provide more natural syntax as used in mathematics and the physical sciences
- improve facilities for portable numerical computation
- allow user-defined data types to support user-built abstract data types
- provide more structures means of data and procedure encapsulation
- introduce the concept of language evolution, giving notice of the possible deletion of obsolete features in future standards

Many other additions have been made, as well, such as a new free-format source form, more general control constructs, recursion, and dynamically allocatable arrays. No FORTRAN 77 features have been deleted - 100% backwards compatibility.



Unsupported Utilities For RTE

Alan Tibbetts
Hewlett-Packard Co.
1266 Kifer Road
Sunnyvale, California 94086

Data Systems Division of Hewlett-Packard has been creating software for the HP 21xx family of computers since 1968. In the 19 years since then over 10,000 software products, such as programs, subroutines, libraries, diagnostics, drivers, operating systems, and subsystems, have been produced. In the course of development of this software many other programs have been developed as tools for the use of HP's factory and field personnel. Some of the best of these programs have graduated to the status of supported products, while others have 'leaked out' to the user community at large. These latter programs, while very useful, have not been given product status because of the commitment to supporting the programs that product status implies.

In the summer of 1987 DSD personnel were asked to share their 'Goodie Bags' with the users by contributing the software to the G-Job project. The software was loosely categorized and then given to Interex.

This session will explain what software was donated to Interex, plus a short description of each program, including 'Why would I use it?'.

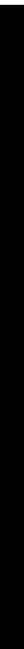


THE ADA*/1000 COMPILER AND RTE-A ADA RUN TIME SUPPORT

John Johnson
Hewlett-Packard Co.
Sunnyvale, California

This tutorial covers the technical issues concerning the Ada/1000 compiler and the RTE-A's support for an Ada run time environment, as viewed by one of the implementation team members. The goal is to allow you to better use the functionality that Ada/1000 provides. The tutorial features a brief history of the Ada/1000 development effort, a discussion of the requirements that Ada and its run time support place upon the operating system and the RTE-A enhancements implemented in order to meet these requirements. Enhancements to Link, initialized VMA, and software signals are covered. This tutorial concludes with a discussion of the performance characteristics and general positioning of the Ada/1000 product.

* Ada is a registered trademark of the U.S. Government (AJPO)



PORTING A REAL-TIME PACKAGE TO HP-UX

Richard M. Pfeiffer

**GE Corporate Research and Development
Schenectady, New York 12301**

J. Robert Ryan

**Hewlett-Packard Company
Woodbury, New York 11797**

Introduction

The experience presented here is the result of a project at GE for porting a real-time Fortran software package from an HP1000 system running RTE* to an HP9000 Model 840 running HP-UX,* the Hewlett-Packard superset of AT&T SVID UNIX.† The project was particularly challenging because the principal programmer was highly specialized in HP1000 machines and relatively new to a UNIX environment, and because of the complexity and size of the software package. The software, which is used for real-time test management and monitoring at GE, consists of 20 or more interrelated programs (depending on the configuration), with a total of approximately 1500 files and 80,000 lines of code. The following sections discuss the project background, describe the software package (called TMGR™) in more detail, outline the differences between the two HP systems, explain the actual porting process used, and summarize what was learned, with some practical guidelines for programmers.

Project Background

At the GE Research and Development Center, the group undertaking this project works extensively with computers in online data acquisition and process monitoring. They also work closely with two related groups that are involved with computer simulation problems and finite element numerical problems. Because of the different re-

* HP, RTE, and HP-UX are trademarks of Hewlett-Packard Company.

† UNIX is a registered trademark of AT&T in the USA and other countries.

TMGR™ is a trademark of GE.

quirements, each group specializes in different computer systems.

The project group has focused on HP1000s since 1969, when an HP2116B computer was first purchased to monitor a particularly large and complex mechanical experiment. Following the success of that experiment, the needs have grown, and computer systems have expanded to include most of the HP1000 models that have become available. The software written has grown from user-written programs, to user-written control programs with canned routines, to a complete package that can have routines added to perform nonstandard tasks with results incorporated into the standard data base maintained by the package. The present software package for HP1000s is called TMGR, which stands for Test ManaGeR. It is used for real-time data acquisition and data management. The scope and usefulness of this package is demonstrated in its use by over 20 sites within GE for their day-to-day work.

Another system heavily used by groups at the R&D Center and around GE consists of DEC VAX computers. They are used for many functions, including program development, word processing, number crunching, and graphic display of data. Many canned software packages are employed, as well as locally developed software.

Because of the different systems used at the company, a common question asked is why TMGR cannot run on large systems, like the 32-bit DEC VAX* line of computers, or on small desktop machines, like IBM PCs.† The answer has been that the slow response of the VMS* operating system to interrupts and the nondeterministic scheduling of processes are major obstacles with the former system, whereas the lack of multitasking in MS-DOS‡ creates problems with the PC. However, the evolving computer environment of supported businesses has led to a serious look at ways of adapting to these systems. In addition, a recent application of TMGR software, monitoring an unusually large set of experimental data, made it obvious that the limits of a 16-bit machine were being pushed on such projects. It had become clear that success at solving problems could be much greater with the capability of using either 32-bit computers with their large address space, small inexpensive machines for more modest applications, or machines that run a more popular operating system.

As a result, an investigation was begun into moving to different machines. The first machine considered was an IBM PC/AT running MS-DOS. Several Fortran compilers were found that were almost as good as that available on the HP1000, but the lack of a multitasking operating system was still a serious problem. Next, a DEC

* DEC, VAX, and VMS are trademarks of Digital Equipment Corporation.

† IBM and IBM PC are registered trademarks of International Business Machines Corporation.

‡ MS-DOS is a trademark of Microsoft Corporation.

VAX running the VMS operating system was considered. It included a Fortran compiler that seemed as rich and comprehensive as that of the HP1000, and VMS was multitasking. Unfortunately, the software tools available were found to be lacking when it came to debugging the small-program/message-passing style of system. Also, it only ran on a VAX, and therefore is limited to DEC hardware. Finally, a system running UNIX was considered. UNIX runs on many manufacturers' machines, including the IBM PC/AT and the DEC VAX. It includes the necessary multitasking, many useful tools, and also runs many of the same canned software packages available under VMS. Unfortunately, however, a good Fortran compiler is not included or generally available.

About this time, HP introduced the HP9000 Model 840 computer, hereafter referred to as the Model 840. This seemed to be a logical path to investigate, as the Model 840 offers a 32-bit machine that claims to address some of the real-time problems expected to be encountered. Also, HP claimed to have an emulation environment that would assist in the migration, and an HP standard Fortran compiler. These features seemed to resolve the major obstacles that had been identified in adapting the software to a new system.

Description of TMGR

In order to illustrate the scope of the project undertaken, it is necessary to give a general description of the TMGR package that was to be ported to HP-UX. TMGR has been developed during the past 12 years at the GE Research and Development Center. It is currently hosted on either an HP disc-based F Series or an A Series computer running RTE6/VM or RTE-A with VC+ software. The user is provided with a configurable package including 20 possible static signal front-end options and 10 dynamic signal front-end A/D hardware choices. Basically, static signals are defined as signals that can be characterized by a single reading valid for a certain time period: i.e., temperature, pressure, or voltage readings. Dynamic signals, on the other hand, comprise many values per reading: i.e., time series readings from vibration or microphone sensors. Typically, signal processing, such as Fourier transformations or correlations, is done on the raw dynamic data before the information is copied to the data base.

A single-user interface program is provided with TMGR that may be customized to allow the monitoring of data collection and control of its progress. Typically, the data collection is scan oriented. This means that every so many time units (usually seconds) a certain sequence of operations is performed. First, data are collected from the specified hardware. The data are then converted to engineering units, and, if desired, out-of-range data are flagged. Next, an optional user-supplied calculation program may be run which has full access to the current data and which may integrate

its results in the data base upon completion. Limits may be checked against supplied values, and data may be archived either to disc, magnetic tape, or both. Finally, the data may be displayed on multiple alphanumeric or graphic display terminals.

All these options may be controlled from the user interface as well as adding and deleting channels from the scan lists. The scan rate may be changed and the limit information modified, as well as changing the display page information, while the test is proceeding smoothly.

The package has found widespread use inside GE, with more than 20 installations currently in operation. An important feature of the package is that it looks the same to the end user, regardless of the choice of computer system and/or front-end hardware. In considering options for porting the package to other systems, this standardized user interface was a high priority.

Description of Model 840 from an HP1000 User's Point of View

A useful perspective for approaching the migration of programs to HP-UX is to look at a Model 840 from an HP1000 user's point of view. In this discussion, it is assumed that the user has much more experience with RTE systems than with UNIX or other similar systems. From this perspective, the following general areas stand out, and will be discussed in more detail below:

- Filename conventions, especially case-dependency, filename length, and path-names
- Processing differences, such as in EXEC call scheduling
- Documentation and the examples used
- Fortran version differences, particularly the consequences of nonstandard HP1000 extensions

Probably the first thing an HP1000 user notices when changing to HP-UX is lowercase letters. Although most users have experience with lowercase, this system does not up-shift automatically, and proper case is extremely important. In other words, if there are a lot of INCLUDE files used in Fortran routines and they happen to be in uppercase because that was correct in the RTE file system, when the INCLUDE files are copied over to HP-UX, they must be converted to lowercase. As can easily be seen, case-dependance can cause major problems.

Along the same line, no doubt many remember when the cry to HP was to allow for filenames longer than six characters. "All the other systems have longer filenames, and extensions!" was the complaint. Finally, the designers at HP heard the noise, so that now 16-character names with 4-character extensions are allowed on the HP1000s.

Although this seemed to be great at the time, there is now HP-UX to consider, which, like UNIX V, only allows 14 total characters, including the “dot”: i.e., 13 actual characters. Just imagine trying to come up with 1000 new filenames as was required here, and then think of the confusion caused if these filenames previously matched the routine names in the code. It’s as if all your old friends suddenly called themselves something different, and then refused to respond to their proper names. (The only consolation in this matter is that 13 characters are better than 6.)

From an HP1000 programmer’s point of view, the Model 840 is a mixed bag. It is different from the safe, familiar, and admittedly peculiar RTE system. Programs do not schedule son and daughter programs with EXEC calls. Instead, they split into two copies of the program itself, and then each examines itself and decides what its job is: either to run the new process or wait for its sibling process to finish.

Devices will now have full-path filenames such as “/dev/null” versus the familiar logical unit 0 for the bit-bucket. In Fortran, the keyboard is unit 5 and the screen is unit 6, not the familiar unit 1. This feature alone can cause a big headache.

Next, all the new documentation is presented in a C-language format, including all examples. This can be a major source of culture shock after 15 years of Fortran-oriented documentation.

The last major difference encountered is that HP1000 Fortran 77 isn’t “standard,” it’s better. A number of systems have been used over the years, and never have as many dialect problems been experienced as with the HP-UX Fortran compiler, mainly stemming from the nonstandard features that are taken for granted on the HP1000. It seems that when useful features are available, people take advantage of them. The big problem is that the HP1000 compiler does not let you know these extensions are being used. Thus, a programmer can become lulled into making use of them. This alone is not necessarily a problem, but the HP-UX “HP standard Fortran compiler” does not address many of these extensions. To be fair, since this project was completed, several other systems have been tried, including standard UNIX “f77,” and these were found to be even worse in this respect.

Most of the differences mentioned here are really significant only on large software packages. In the present case, the problems were more acute because the software consists of many interrelated programs that depend heavily on interaction with the operating system and that must function in real time. For many applications, the benefits of the UNIX-like operating system and much larger memory may far outweigh the difficulties. In addition, the PORT/HP-UX emulation environment allows many programs from RTE systems to run on the Model 840 with very little conversion effort. The porting process itself, as well as the choice of full conversion to HP-UX or to PORT/HP-UX, is discussed in more detail in the next section.

Explanation of the Porting Process

Figure 1 shows the steps which were followed in the process of porting Fortran programs from an HP1000 to the Model 840. As shown, the initial steps are similar for programs destined for either full HP-UX or PORT/HP-UX on the Model 840. First, the MAU utility is run on the HP1000 source code. It generates many pages of output detailing the system calls made by the code, and divides them into three categories: routines emulated by PORT/HP-UX, routines partially emulated by PORT/HP-UX, and routines not emulated by PORT/HP-UX. This very useful information can be used to determine the amount of work required to move the code to either PORT/HP-UX or full HP-UX. If many of the system calls are fully emulated by PORT/HP-UX, it should be an easy task to make the code run in the emulation environment. If many of the system calls are not emulated or are only partially emulated, then consideration should be given to moving to full HP-UX. This gains the full advantage offered by the Model 840. An annoying problem with the information is that it only checks for system calls, and only those that it is familiar with. It does not determine how much work must be done to the code for the new Fortran compiler to accept it.

Let us assume that you decide to migrate your package to the PORT/HP-UX environment based on this information. Now you must decide what to do about filenames. If you make use of INCLUDE files, they probably need to be made lower-case on the HP-UX system. This requires that each \$INCLUDE statement in the source code be modified to reflect the change. Also, the files should be renamed so that they are no longer than 14 characters. These conversions can all be performed on the HP1000 because of its insensitivity to case and the availability of longer filenames.

Next, make sure all the files are type 4. Then the TF program must be used to write a "tar" format tape. This step should be done with a relative path rather than an absolute one, as the HP-UX "tar" program cannot modify the directory information when reading the tape back in. Finally, use "tar" to read the tape into the Model 840.

Now, with all the files on the Model 840 in their proper directory position, the next thing that must be done is pass them through the "sed" filter HP has supplied. This filter addresses some of the Fortran differences between the two machines. For instance, it will comment out most compiler directives and write the appropriate Model 840 compiler directives. It will also place INCLUDE filenames in quotation marks. All directives that perform memory management on the HP1000, such as \$CDS, \$EMA, and \$MSEG are removed, as they are superfluous on a 32-bit machine. Also, it removes any comments from subroutine and function statements if they do not occur on a continuation line.

Once the above work has been completed, the difficult part begins. Now the source

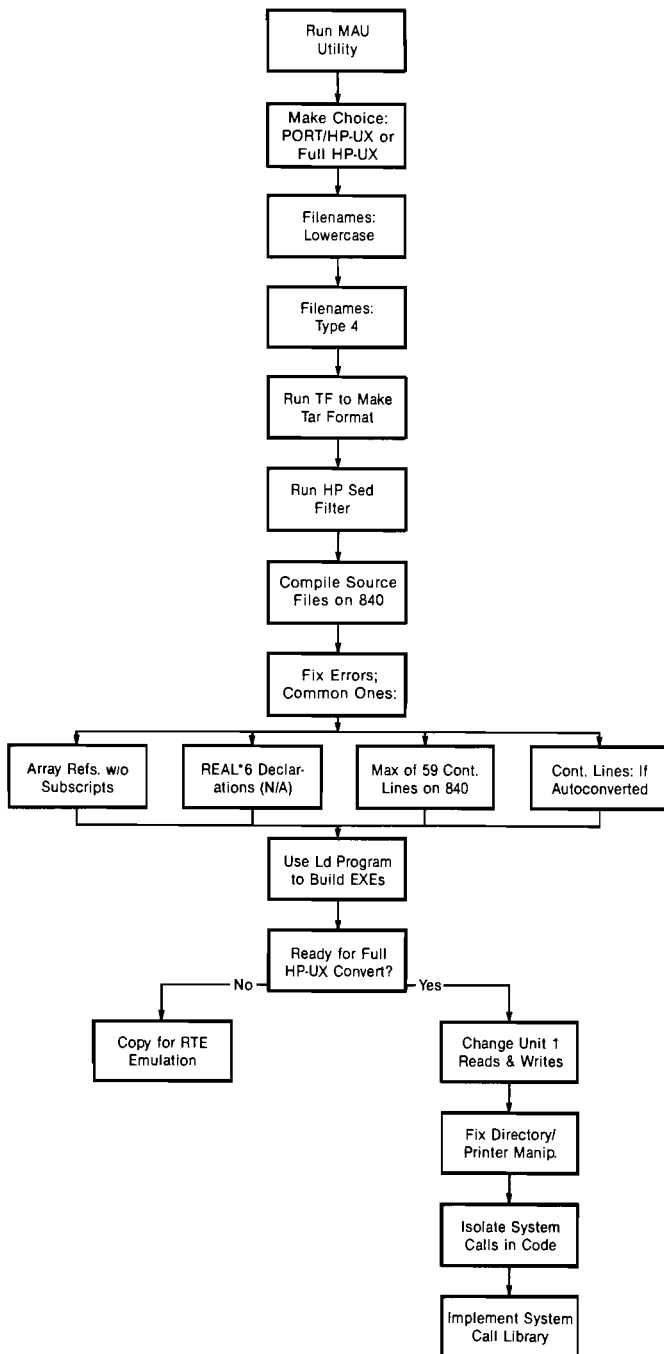


Figure 1. Steps for porting Fortran from HP1000 to HP9000 Model 840

files must be compiled using the Model 840's Fortran compiler. Errors will occur unless extreme care was taken on the HP1000. These are some common problem areas:

- Array references without subscripts in other than call statements
- Real*6 declarations, which are not supported on the Model 840 (change them to Real*8 or Double Precision)
- Continuation lines, because the Model 840 compiler has a maximum of 59 continuation lines, and it may be difficult to fix code exceeding the maximum
- Other errors that may arise if some of the automatically converted features occur in continuation lines

The major problem among these is related to variable subscripts of arrays passed into subroutines. This problem occurs when array subscript names are not defaulted to integer variables and are defined in the wrong order. For instance:

```
SUBROUTINE PEAR(ARRAY,SUB1,SUB2)
.
.
REAL*4 ARRAY(SUB1,SUB2)
INTEGER*2 SUB1
INTEGER*2 SUB2
.
etc.
```

This example will produce compiler errors on the Model 840 but not on the HP1000. To correct this either change SUB1 and SUB2 to ISUB1 and ISUB2 or simply reverse their order:

```
SUBROUTINE PEAR(ARRAY,SUB1,SUB2)
.
.
INTEGER*2 SUB1
INTEGER*2 SUB2
REAL*4 ARRAY(SUB1,SUB2)
.
etc.
```

After all the compiler errors are resolved, the next step is to attempt to build the executable files. Depending on the number of routines included, either a library constructed with the "ar" utility, or a list of files, may be supplied directly to the "ld" program. If this is expected to be done repeatedly, HP-UX has a "make" utility which monitors changed files and updates the appropriate ones when any changes have been made. The only problem is that the user must supply all the information in a text file

about which files depend on which other files for consistency. This means that if a change is made in the source of an INCLUDE file, and several routines use that file, "make" will schedule the compiler to update the relocatable files, the libraries they are included in, and any executable files that use any of these files. It is very tedious to build these "make" files and keep them up to date, although overall it tends to be worth the effort. There are also utilities in development that will scan source files and write the "make" files. Most of these are not perfect yet.

Once all the executable files are ready, they must be copied along with any required data files into the file space set up for the emulation environment. Provided utilities convert these files so that the RTE emulation thinks that they are RTE files.

These applications can only be run from within "rtesh," the emulation environment user interface program. Many RTE commands are available to the user, such as the ability to "RP" programs, and the ability to flag programs as unsharable, i.e., non-cloneable. The emulation seemed to be very complete but slightly slower than an A900 if the code uses many system calls.

Assuming you now desire to finish the migration to full HP-UX, what must be done? First, perform all the previous steps up to copying the files into the emulation file space. This is obvious as the programs must still be compiled and linked. The next thing to do is to go through the code and change all reads from unit 1 to reads from unit 5 and all writes to unit 1 to writes to unit 6. Then, if necessary, explicitly open any printer under a different unit number. Any code that manipulates the directory structure may have to be changed because of the single-root-directory structure of HP-UX. Finally, the case and size of program data files may have to be changed to match the conventions of HP-UX. The most difficult part is going through the code and isolating the system calls. Small interface routines can be written to perform generic system functions. This is simpler in a large package than changing all the similar calls to be HP-UX specific. The code is also more portable to other systems in the future. Some examples of this type of routine are retrieving the system time, scheduling another program, sending some data to/from another program, or outputting to a specific device. While none of these routines is impossible to write, depending on the specific requirements, a particular routine may be rather difficult to arrive at.

Some RTE features that must be simulated are *suspend saving resources* and *running on the clock*. The ability to suspend saving resources in core is one of RTE's greatest assets. It allows a process to start up and get ready to perform some function. The next time it is called for, it is all set to respond. It does not have to be searched for on the disc, and files can already be open. In HP-UX, this feature can be simulated by running a program that performs its setup and then suspends awaiting some signal or message. It is more flexible than RTE's technique, but requires more cooperation among the programs. Running a program on the clock can be handled similarly. The process itself must be set up and then suspended until a certain time. A parent pro-

cess may also be written to perform the same thing, but that adds additional overhead.

What Was Learned: Programming Guidelines and System Evaluations

One of the most significant lessons learned from the migration process was about the programming techniques which have been in use. It must be admitted that this process has forced a hard look at ways to improve programming practices, and thus avoid many of the problems that were encountered. The guidelines below should be useful for writing any software that might find itself on another machine someday (which could cover almost all of it), and of course will be of particular interest for software that is definitely earmarked for future porting.

1. *Learn the "language standard."* After all these years writing Fortran, it was assumed that what ran on the HP1000 was standard Fortran. It was also assumed that everyone supported all of MIL-STD-1753. Only after starting this project was it discovered how wrong these assumptions were.
2. *Always use named constants.* Many of the problems encountered were related to what the default integer size was. It is usually easy to override the default, but hard to enforce the presence of the option.
3. *Try never to build structures with equivalences.* Unfortunately, Fortran does not have structures. This makes it very easy to abuse the equivalence feature. If it is absolutely necessary to use it, do so at a low level, and only access the structure through function calls.
4. *Never port code that is changing.* While one group was porting the code, other people were adding new enhancements. This makes it very difficult to compare results because you do not know which machine has the error.
5. *Things always take longer than planned.* How much longer depends on the programmer's experience and on the complexity of the program. In this case, about half the project effort went into learning and getting used to the new system. With the large, complex set of programs that make up the software package being ported, it is estimated that the porting process took almost half the time that would be needed to rewrite the programs completely. Porting less complex software, on the other hand, may take only a small percentage of the effort that would be required to rewrite the code. Keep in mind that if there is a choice between porting software and buying more hardware, the hardware approach could be *much* less expensive in some cases.
6. *Learn to use as many of the tools available on the new system as possible.* There may be compensating features and productivity improvements possible on the new system that will eventually overcome any drawbacks of the transition. Unless the tools are learned, however, the full advantage of the new system will

never be exploited.

7. *Use the source code control system (SCCS).* One of the HP-UX tools deserving special mention for porting projects is SCCS. This tool keeps track of all the modifications made to each file in a compact format, i.e., by recording the differences rather than each complete version. Any previous version can then be restored as needed.

From this experience, a healthy respect for both HP-UX and RTE has been gained. Quite honestly, both have good and bad features. Comparing file systems gives RTE's global directories on each volume the edge over HP-UX's single root directory per file system. Variable-length extensions and hidden files in HP-UX are better than RTE's maximum of 4-character extension with no hidden files. However, the two worst features of the HP-UX file system are the short filenames and the amount of work it takes to walk the directory tree in the shell.

In a comparison of utility tools available, HP-UX wins hands down. But RTE also has several excellent utilities. In particular, EDIT/1000 wins handily over the combination of "sed" and "vi." This is not because one is better than the other, but because one is definitely better than two: i.e., it is easier to learn the full power of EDIT/1000 than to learn both utilities on HP-UX. Also deserving honors is RTE's WH utility. HP-UX's system status display utility, "ps," does not present the user with nearly as much useful information. On the other hand, "make" under HP-UX is almost the answer to every programmer's prayers (if it could only write its own dependencies file).

Comparing Fortran compilers is easy. Except for one bug that was discovered in the compiler on the HP1000, the only feature lacking is an option to flag nonstandard usage. The Fortran compiler on the Model 840 has more VAX extensions than HP1000 extensions. In fact, the compiler on the Model 840 was found to differ from that found on the current HP9000 Model 320 system. This is not a good omen for future compatibility.

The major Fortran deficiency of the HP1000 compiler is the Hollerith versus character data storage problem. This is a hard problem to resolve, as the underlying architecture has byte-addressing problems. Other systems seem to have solved it, however. Maybe an option could be added to RTE to optionally accept a byte address in a system call; all the old user code could be converted to use character variables.

The primary two difficulties that cannot be efficiently addressed by the compiler, or the system, are the size of numerical constants: i.e., whether a numeric 27 is a 16- or 32-bit constant. More insidious are *alignment problems*, particularly in common blocks, and worse, in equivalence statements. The first problem may be addressed by appending either an I or a J to all integer constants in the source code. This is an ugly solution, but the only other answer is to always use parameter constants which are de-



clared like all other variables. This solution usually makes the code more self documenting, assuming the symbolic name chosen is sensible and appropriate.

On the other hand, the equivalence problem is more difficult to resolve. First, all equivalences that are not required should be removed. Although this seems obvious, some Fortran code is old, and equivalences may have been used to speed up array access to fixed locations. Most, if not all, modern compilers now perform this function for the programmer automatically. The second reason for equivalences is variant use of data space: i.e., using a variable as an integer area in some cases and as a floating point area in others. Also falling in this area is the standard practice of using equivalences to build "structures" in Fortran. Unfortunately, byte, word, and double-word alignment problems are not addressed by any standard, and therefore if these techniques are used the code may be hard to port to other systems in the future.

Conclusions

After completing this project, the authors' conclusion is that porting a major software package to a new operating environment can be a practical option if approached cautiously and for the right reasons. Good reasons would include the necessity of moving to a larger system or the requirement of using a more popular operating system, especially for those who are already comfortable in the UNIX environment. On the other hand, if the goal is to change the software to save on new hardware, that may turn out to be a very costly "saving" in the long run, especially for fairly specialized software packages with a limited number of applications. Not only is the porting process itself a potentially large and unknown cost factor, versus the known cost of the hardware, but the hidden costs such as learning a new system and supporting two separate versions of the software may carry on well into the future. For software that truly deserves the available size and flexibility, however, the Model 840 has what appears to be a very reliable combination of hardware and a full range of software tools, especially for porting Fortran from the HP1000s.

The overall impression of the HP9000 Model 840 with the HP-UX operating system is that it is great. During the entire 7-month project using a very early machine, it had no hardware problems. Very few software problems in the HP-supplied code were found during that time, and they were all quickly resolved by HP.

References

1. *American National Standard Programming Language Fortran*, ANSI X3.9-1978, American National Standards Institute, Inc., 1978.

2. *Fortran 77/1000 to HP Fortran 77 or HP-UX Migration Guide*, PN 92430-90003, Hewlett-Packard Company, 1986.
3. *Fortran 77 Reference Manual*, PN 92836-90001, Hewlett-Packard Company, 1985.



PORT/HP-UX vs. Native Mode Migration

Stephen C. Fullerton
Statware, Inc.
P.O. Box 510881
Salt Lake City, UT 84151

July 20, 1987

1 Introduction

In April of 1986 Statware was invited by Hewlett Packard to participate in the *Fast Start* migration program. In particular, we were invited to port our statistics package, STAT80, to the HP 9000 Series 840. Since all STAT80 development is done on a HP 9000 Series 550, the port would not have been very difficult; however, rather than port the HP-UX code, I decided to port the HP 1000 code as an experiment to test out the capabilities of the PORT/HP-UX system.

I thought that this port would represent a worst case scenario for conversion to the HP 9000 Series 840 whereas taking the code from the HP 9000 Series 550 would be a best case scenario. This paper describes the experience using PORT/HP-UX, then porting native mode, and finally, a comparison of the two with some benchmarks.

Throughout this paper references to HP-UX and UNIX should be interchangeable except with the PORT/HP-UX facilities.

2 Migration Problems

STAT80 is closely tied to the host operating system. For example, there is no FORTRAN I/O; all I/O is done directly through the kernel. In addition, there are many other operating system calls. When I ported to the HP 1000, all system dependent HP-UX (UNIX) external references had to be converted to the analogous HP 1000 system call.

The following sections give an overview of the changes and effort required to port STAT80 from HP-UX to the HP 1000. These sections are important because I was trying to achieve UNIX functionality on an HP 1000 and had to use many HP 1000 system calls and much of the FMP library. Once the port was complete, a package that was tightly coupled to HP-UX was now even more firmly bolted to the HP 1000. When this converted code is moved back to HP-UX using PORT/HP-UX a complete, albeit circular, migration along Hewlett-Packard technical computers has been done.

2.1 System Calls

Strings passed to the kernel under UNIX must be ASCIZ strings; ASCII strings followed by a trailing zero byte (null). The HP 1000 operating system wants none of this, strings are passed "as is", or else

blank padded. Also, since FORTRAN passed all arguments by reference and character variables by a descriptor, there are incompatibilities with the HP-UX kernel where arguments are passed strictly by value or reference. The HP-UX FORTRAN 77 compiler provides the `$ALIAS` directive to help alleviate this problem. For example, the system routine, `write(2)`, has three arguments, the first and third are passed by value and the second is passed by reference. The FORTRAN code to call this routine would be something like:

```
$ALIAS WRITE = 'write' (%val,%ref,%val)
      INTEGER WRITE
      :
      ISTAT = WRITE(FD,BUF,NBYTES)
```

When passing a character string, the `$ALIAS` must declare the corresponding to be passed by reference, and a trailing null must be attached. For example,

```
$ALIAS SYSTEM = 'system' (%ref)
      INTEGER SYSTEM
      :
      ISTAT = SYSTEM("date"//char(0))
```

2.2 Input and Output

Input and output conversion from HP-UX to the HP 1000 was especially difficult as the HP-UX operating system views files as a stream of bytes, regardless of the content of the file. The HP 1000 has different types of record-oriented files. Also, I/O to and from a terminal is handled transparently to the program and doesn't require special coding. All I/O in STAT80 is done under HP-UX by using the `creat(2)`, `open(2)`, `read(2)`, `write(2)`, `close(2)`, `lseek(2)`, and `unlink(2)` system calls. This is as low level as you can get under UNIX and is roughly equivalent to HP 1000 system calls (EXEC, XREIO). One problem with using the HP 1000 system calls is that STAT80 does all of its buffering in EMA and assumes 4 byte integers. All HP 1000 system calls explicitly require 2 byte integers. Therefore, before writing all data was copied to a local integer array dimensioned `INTEGER*4`, that was equivalenced with a local integer array dimensioned `INTEGER*2`. All other information had to be copied to and from 2 byte integers.

The terminal I/O was converted to use XREIO in place of `read(2)` and `write(2)`. Under HP-UX, STAT80 reads from the "standard input", writes to the "standard output", and displays all error messages to "standard error." Terminal input is buffered with one-line buffers. Terminal output is buffered like any other file under UNIX, with multiple lines in a buffer. The terminal output buffer is flushed either when full, or else when a prompt is issued. Error output is unbuffered and is written one byte at a time.

In order to simulate this for the HP 1000, I used an XREIO call for the terminal read. This mapping was almost one-to-one as XREIO returns a single line. The terminal output was more difficult as I couldn't buffer more than one line and under UNIX, the newline character is used as a line delimiter and it is produced by STAT80 at the end of every line. The output routines had to be modified to detect a newline character to signify the end of the line and then output the line using an XREIO call. If the output line was flushed without a newline; e.g., a prompt, then the control words had to be set to the appropriate octal mask for generating this type of output. Standard error handling was even more difficult as I wanted it to be unbuffered; e.g., a 50 byte error message would result in 50 XREIO calls, each with a single byte.

File I/O was converted to use the FMP library. The `open(2)` was replaced by `FMPOPEN`, `close(2)` was replaced by `FMPCLOSE`, `read(2)` and `write(2)` were replaced by `FMPREAD` and `FMPWRITE`,

respectively. The binary I/O had to do its own record spanning, something not necessary under UNIX. All of these routines had to handle the newlines; stripping them on output and adding them on input.

File positioning is done by record than by byte; therefore, changes had to be made to accommodate this.

2.2.1 Sample Routines

The following routine, *sysred*, is the STAT80 routine to read from any file or device into a buffer. The HP-UX version of this routine is:

```
integer function sysred (fildes,buffer,nbytes)
* (read -- system dependent)
*-----
*
* STAT80: An Interactive Statistical Package
*
* Copyright (C) 1985 Statware -- All Rights Reserved
*
* Proprietary Software: The contents of this routine shall
* not be disclosed or made available, or any portion thereof
* in any form whatsoever to any person other than the author
* without prior written approval of the author.
*
*-----
*
* Read nbytes into buffer(*) from file descriptor: fildes (may be
* a unit number on some machines).
*
* @(#) sysred.r 1.3 11/4/86 09:34:39
*
*-----
$ALIAS read = 'read' (%val,%ref,%val)
*
* external references (function,subroutine,common)
*
* external refs      read
*
*-----
*
* external functions and subroutines
*
* integer            read
*-----
*
* non-common variables
*
* integer            fildes,      buffer(1),  nbytes
*
* return(read(fildes,buffer,nbytes));
* end
```

For UNIX systems, all that *sysred* does is provide a wrapper around the system routine, *read(2)*¹. The HP 1000 version of *sysred* is as follows:

```
FTW7Y,J
$CDS ON
```

¹Actually it isn't quite as simple as this because additional code is required to process a read that is interrupted.

```

$ALIAS krputo, EMA
integer function sysred (fildes,buffer,nbytes,nrecsz)
$ (read -- system dependent)
-----
$
$
$ STAT80: An Interactive Statistical Package
$
$ Copyright (C) 1985 Statware -- All Rights Reserved
$
$ Proprietary Software: The contents of this routine shall
$ not be disclosed or made available, or any portion thereof
$ in any form whatsoever to any person other than the author
$ without prior written approval of the author.
$
-----
$
$ Read nbytes into buffer(*) from file descriptor: fildes (may be
$ a unit number on some machines).
$
$ @(#) sysred.r 1.2 5/15/86 16:47:24
$
-----
$
$ external references (function,subroutine,common)
$
$ external refs      fmpread,      xreio
$
-----
$
$ external functions and subroutines
$
$ integer*2          fmpread
$
-----
$
$ non-common variables
$
integer      fildes,      buffer(1),  nbytes,      nrecsz
ema          fildes,      buffer,      nbytes,      nrecsz
integer*2    klen,        kerror,      kctrl(2),    nr
integer*2    ia,          kbuff(256)
integer*4    lbuff(128)
logical      btest

$
equivalence      (kbuff(1),lbuff(1));
$
include files.h
include iounits.h
include memglob.h
include ascii.h
$
kfp = fildes;
nr = min0(nbytes,nrecsz);
nr = min0(nr,512);
if (kfp == stdin) {
kctrl(1) = termiu;
kctrl(2) = 400B;
nr = -nr;
call xreio(1i,kctrl,kbuff,nr);
call abreg(ia,klen);
if (klen <= 0) {
if (btest(ia,7))          # eof
klen = -1;
}
if (klen > 0) {
nw = (klen + 3) / 4;
}
}

```



```

        for (j = 1; j <= nw; j = j + 1)
            buffer(j) = lbuff(j);
    }
} else {
    klen = fread(fildcb(1,kfp),kerror,kbuff,nr);
    if (kerror != 0) {
        klen = -1;
    } else {
        nw = (klen + 3) / 4;
        for (j = 1; j <= nw; j = j + 1)
            buffer(j) = lbuff(j);
    }
}
if (klen < 0) {
    nc = 0;
} else {
    nc = klen + 1;
    call krputo(ASCLF,buffer,nc);
}
return(nc);
end

```

An additional argument is necessary as the HP 1000 file system is based on records, to that the record size to read must also be known. Special processing is necessary to handle terminal input, end-of-file checking, copying to the EMA buffer, the INTEGER*2 external references, etc. Also note that after the record is read, a trailing *newline* character is inserted.

The following routine, *syswrt*, is the STAT80 routine to write to any file or device from a buffer. The HP-UX version of this routine is:

```

integer function syswrt (fildes,buffer,nbytes)
* (write -- system dependent)
*-----
*
*
*   STAT80:  An Interactive Statistical Package
*
*   Copyright (C) 1985  Statware -- All Rights Reserved
*
*   Proprietary Software:  The contents of this routine shall
*   not be disclosed or made available, or any portion thereof
*   in any form whatsoever to any person other than the author
*   without prior written approval of the author.
*-----
*
*   Write nbytes of buffer(*) to file descriptor:  fildes (may be
*   a unit number on some machines).
*
*   @(#) syswrt.r 1.2 5/15/86 16:48:16
*-----
*
*   external references (function,subroutine,common)
*
*   external refs      write
*
*-----
$ALIAS write = 'write' (%val,%ref,%val)
*
*   external functions and subroutines
*
*   integer            write
*-----

```

```

*
*   non-common variables
*
integer          fildes,    buffer(1),  nbytes
*
return(write(fildes,buffer,nbytes));
end

```

For UNIX systems, all that *syswrt* does is provide a wrapper around the system routine, *write(2)*. The HP 1000 version of *syswrt* is as follows:

```

FTN7X,J
$CDS ON
$ALIAS krgeto, EMA
integer function syswrt (fildes,buffer,nbytes)
  (write -- system dependent)
*-----
*
*   STAT80: An Interactive Statistical Package
*
*   Copyright (C) 1985 Statware -- All Rights Reserved
*
*   Proprietary Software: The contents of this routine shall
*   not be disclosed or made available, or any portion thereof
*   in any form whatsoever to any person other than the author
*   without prior written approval of the author.
*-----
*
*   Write nbytes of buffer(*) to file descriptor: fildes (may be
*   a unit number on some machines).
*
*   @(#) syswrt.r 1.2 5/15/86 16:48:16
*-----
*
*   external references (function,subroutine,common)
*
*   external refs      fmpwrite,   xreio,    abreg
*-----
*
*   external functions and subroutines
*
integer*2        fmpwrite
*-----
*
*   non-common variables
*
integer          fildes,    buffer(1),  nbytes
ema             fildes,    buffer,    nbytes
integer*2       kctrl(2),  kerror,  nb,          klen
integer*2       ia,        kbuff(256)
integer*4       lbuff(128)
*
equivalence     (kbuff(1),lbuff(1));
*
include iounits.h
include files.h
include filunits.h
include ascii.h
*
kfp = fildes;
kc = krgeto(buffer,nbytes);

```

```

if (kfp == bakfp || kfp == stderr) {      # terminal output
  kctrl(1) = termiu;
  if (kc == ASCLF) {      # newline
    kctrl(2) = 0;
    nb = nbytes - 1;
  } else {
    kctrl(2) = 3000B;
    nb = nbytes;
  }
  kw = (nb + 3) / 4;
  kw = min0(kw,128);
  for (j = 1; j <= kw; j = j + 1)
    lbuff(j) = buffer(j);
  nb = -nb;
  if (kfp == stderr)
    kctrl(2) = kctrl(2) + 40000B;
  call xreio(2i,kctrl,kbuff,nb);
  call abreg(ia,klen);
} else {
  if (kc == ASCLF)
    nb = nbytes - 1;
  else
    nb = nbytes;
  kw = (nb + 3) / 4;
  for (j = 1; j <= kw; j = j + 1)
    lbuff(j) = buffer(j);
  klen = fmpwrite(filddb(1,kfp),kerror,kbuff,nb);
  if (kerror != 0)
    klen = -1;
}
if (klen <= 0)
  nc = 0;
else
  nc = klen;
return(nc);
end

```

The HP 1000 version requires special processing to handle terminal output, especially if a trailing newline isn't there. Error output is also handled differently. Before writing the buffer must be copied from EMA into a local buffer that is equivalenced to an INTEGER*2 buffer. If a newline is in the buffer, it must be stripped before writing. Only a single record is written with the HP 1000 version of *sysvrt*.

2.3 Command Line Arguments

Command line arguments under HP-UX are retrieved from FORTRAN by the use of either the *getarg*(3F) library routine or the *Fin_getarg* FORTRAN library call depending upon which HP-UX system is used. The UNIX operating system hands the program the arguments one at a time. For the HP 1000, the EXEC 14 system call was made to recover the command string that scheduled the program. This command string was parsed into separated arguments. Unlike UNIX, the recovery of the command string must be done before I/O is attempted; otherwise, it will fail.

The following is a listing of the subroutine, *args*, for the HP 9000 Series 550 to extract the command line arguments.

```

      subroutine args(argc,argv)
      * (return command line arguments, UNIX style)
      *-----
      *

```

```

*   STAT80:  An Interactive Statistical Package
*
*   Copyright (C) 1985  Statware -- All Rights Reserved
*
*   Proprietary Software:  The contents of this routine shall
*   not be disclosed or made available, or any portion thereof
*   in any form whatsoever to any person other than the author
*   without prior written approval of the author.
*
*-----
*
*   @(#) args.r 1.2 5/21/86 09:25:35
*
$ALIAS getarg = 'Ftn_getarg' (%ref,%ref,%ref)
*
      integer          getarg
      character*40     argv(0:19)
      integer          argc
*
      argc = 0;
      la = getarg(argc,argv(argc),40);
      while (la > 0 && argc < 19) {
          argc = argc + 1;
          la = getarg(argc,argv(argc),40);
      }
      return;
      end

```

The UNIX operating system numbers command line arguments beginning with 0. The 0th argument is the program name, and so on. The tricky part is if UNIX reports that `argc = 2`, then only one argument was specified to the program. If the C programming language is used, then argument extraction is much easier; e.g.,

```

main(argc,argv)
int argc;
char *argv[];
{
    :
}

```

The HP 1000 version of `args` requires more processing as the operating system doesn't do anything except return the command line. Note that STAT80 ignores the first argument (program name).

```

FTH7X,J
$INCLUDE alias.inc
$CDS 0#
      subroutine args(argc,argv)
*   (return command line arguments, UNIX style)
*-----
*
*   STAT80:  An Interactive Statistical Package
*
*   Copyright (C) 1985  Statware -- All Rights Reserved
*
*   Proprietary Software:  The contents of this routine shall
*   not be disclosed or made available, or any portion thereof
*   in any form whatsoever to any person other than the author
*   without prior written approval of the author.
*
*-----

```

```

*
*   @(#) args.r 1.2 5/21/86 09:25:35
*
integer          getarg
character*40     argv(0:5)
integer          argc
integer*2        kbuff(128)
integer*4        lbuff(64)
integer*2        ia,          ib
*
equivalence (kbuff(1),lbuff(1))
*
include ascii.h
*
  argc = 1;
  call exec(14i,1i,kbuff,-256i);
  call abreg(ia,ib);
  if (ia > 0)
    return;
  nb = ib;
  kntc = 0;
  mrkl = 0;
  for (j = 1; j <= nb; j = j + 1) {
    kchar = krgeto(lbuff,j);
    if (kchar == ASCSPACE || kchar == ASCCOMMA)
      kntc = kntc + 1;
    if (kntc == 2) {
      mrkl = j + 1;
      break;
    }
  }
  if (mrkl <= 0)
    return;
  repeat {
    mrkr = nb;
    for (j = mrkl; j <= nb; j = j + 1) {
      kchar = krgeto(lbuff,j);
      if (kchar == ASCSPACE || kchar == ASCCOMMA) {
        mrkr = j - 1;
        break;
      }
    }
    ls = mrkr - mrkl + 1;
    argv(argc) = ' ';
    call kritoc(argv(argc),lbuff,mrkl,ls);
    argc = argc + 1;
    mrkl = mrkr + 2;
  } until (mrkl > nb || argc > 5);
  return;
end

```

2.4 Program Scheduling

STAT80 allows a program to be scheduled "with wait." This task is relatively easy for both UNIX and the HP 1000. The *system(3)* call was used for UNIX and the *FMPRUNPROGRAM* call was used for the HP 1000. However, error handling differs between the systems in respect to the realm of possible program scheduling errors.

The following is a portion of code for HP-UX program scheduling. Note the use of the *\$ALIAS* directive.

```
subroutine cmd114(redo,errflg)
```

```

* (run command -- id 114 hp-ux version)
$ALIAS system = 'system' (%ref)
.
.
integer          system
.
.
kerror = system(runstr(1:lens)//char(0));
if (kerror >= 0)          * no error
    errflg = .false.;
return;
end

```

The HP 1000 version of this routine is almost identical, except that the FMPRUNPROGRAM external is used rather than *system(3)*.

```

FTN7X,J
$INCLUDE alias.inc
$CDS ON
$EMA /s80cb6/
subroutine cmd114(redo,errflg)
* (run command -- id 114 hp 1000 version)
.
.
character*128      runstr
character*64       runnam
character*30       errstr
integer*2          prams(5), kerror, fmprunprogram
.
.
kerror = fmprunprogram(runstr(1:lens),prams,runnam);
if (kerror < 0) {          * error
    call fmperror(kerror,errstr);
    call perrc('?? run: %s&n',errstr);
    return;
}
errflg = .false.;
return;
end

```

2.5 Date and Time

Under UNIX the date and time are retrieved using the *time(2)* and *localtime(3C)* system calls. For the HP 1000, the date and time are retrieved using the EXEC 11 system call. The information returned from the EXEC 11 system call is completely different from the UNIX *localtime(3C)* call and requires more processing; e.g., conversion from day of year to a month and day.

The following routine, *gtime*, is used by STAT80 on HP-UX to retrieve the date and time information. All of the parameters are passed by reference to simplify calling this routine by FORTRAN.

```

#include <time.h>

void gtime(month,day,year,hour,min,sec)
int *month, *day, *year, *hour, *min, *sec;
{
    struct tm *mtime, *localtime();
    long clock;

    clock = time((long *) 0);
    mtime = localtime(&clock);

```

```

*month = mtime->tm_mon + 1;
*day = mtime->tm_mday;
*year = mtime->tm_year;
*hour = mtime->tm_hour;
*min = mtime->tm_min;
*sec = mtime->tm_sec;
return;
}

```

2.6 Character and Bit Primitives

STAT80 makes use of a number of character and bit handling primitives for low level packing, unpacking, comparison, and move operations. For UNIX machines, there are usually done in “C” as it handles byte operations efficiently. For the HP 1000, I used the MIL-STD-1753 extensions to the ANSI 77 standard, in particular, the bit primitives. These primitives handle AND, OR, NOT, and bit shifting operations. They are not nearly as efficient as byte operations but are easily implemented. All of these used hexadecimal masks specified in DATA statements.

The character primitives most heavily used by STAT80 are *krgeto* and *krputo*, for getting and putting ASCII ordinates from a packed string, respectively. The HP-UX version of *krgeto* is written in C and is small and fast.

```

/* krgeto -- get character ordinate */

krgeto(text,npos)
char *text;
int *npos;
{
    return(text[*npos-1]);
}

```

The HP 1000 version of *krgeto* makes use of bit masking and shifting to extract the ordinates².

```

FIN7X,E,J
$CDS 0#
    integer function  krgeto (text,npos)
#
    integer          char
    integer          kword,      mask1,      mod
    integer          npos,      text(1)
#
    data mask1/z'000000ff'/
#
    char = 0;
    if (npos > 0) {
        kword = (npos + 4 - 1)/4;
        nshl = mod(npos-1,4)*8 - 24;
        char = ishft(text(kword),nshl);
        char = iand(char,mask1);
    }
    return(char);
end

```

Putting an ASCII ordinate into a packed string is just as easy as extracting it when in the C programming language; however, it is more difficult when only using bit primitives. The HP-UX version of *krputo* is:

²This could be done faster by using assembly code.

```
/* krputo -- put character ordinate */
```

```
void krputo(ord,text,npos)
char *text;
int *npos, *ord;
{
    text[*npos-1] = *ord;
}
```

The HP 1000 version of *krputo* is as follows:

```
FTN7X,E,J
$CDS 0#
    subroutine krputo (ord,text,npos)
*
    integer      ord,          mask2,    npos,      nror
    integer      kword,       mask1,    mask3,     mod
    integer      source,      target,   text(1)
*
    data mask1/z'000000ff'/, mask3/z'ffffff00'/
*
    if (npos > 0) {
        kword = (npos + 3)/4;
        nror = 24 - 8*mod(npos-1,4);
        mask2 = not(ishft(mask1,nror));
        source = ishft(ord,nror);
        target = iand(text(kword),mask2);
        text(kword) = ior(target,source);
    }
    return;
end
```

2.7 Other Externals

There were other system dependent routines that had to be converted, replaced, or modified to provide the equivalent function on the HP 1000. These dealt with retrieving the user name, working directory (and setting the working directory), checking if the standard output is a terminal, etc.

2.8 Data Segment

The most severe problem encountered when porting STAT80 to the HP 1000 was the limitation on the size of the data segment. STAT80 is designed for optimal performance on virtual memory computer systems and uses VMA on the HP 1000. However, this is at a cost of 3 pages of the data segment. STAT80 is loaded as a CDS program to reduce the size of the data segment; however, when all of the system dependent changes were made, the size of the data segment would have been about 80 pages. I had a long way to go to reduce this so that STAT80 would run as a VMA program.

I tracked the extremely large data segment down to one problem; function and subroutine arguments passed by value are protected by placing them into the static portion of the data segment rather than into a code segment. This protection means that if the called routine modifies a "read-only" argument, the program will not fail. Therefore, for every integer or real passed as a constant, 4 bytes of static memory were lost³. The worst problem was with character strings, particularly error messages. STAT80 has over 1600 error message strings that appear as a subroutine argument. A program was written to locate error message subroutine calls and strip the strings from the code

³STAT80 is compiled with the FTN7X "J" option for 4 byte integers by default.

into a type 2 file, and replace the string with an integer constant that is the record number in the type 2 file of the error message. The entire error message handling was rewritten to extract the error strings from a type 2 file. This reduced the size of the data segment to approximately 48 pages.

In addition to the error messages, STAT80 has its own I/O library that uses a conversion string similar to the "C" programming language. Thus there were many more strings in the code. Placing these strings in a type 2 file wasn't viable because file access is much too slow for the regular output. The only solution would be to get the FTN7X compiler to store the strings in the code segment.

I had two choices, use assembly code to store the strings into the code and then copy them into the data segment for processing, or else generate the code inline necessary to load them from the code segment. I chose the latter. The code is rather gruesome; however, it only uses more code, not data and since STAT80 is loaded CDS, the tradeoff was fine. For example, the following is some code before the translation:

```
call prtfi(stdout,fplog,'Valid cases = %6d&n',numok)
```

This was translated to:

```
iqhbuf(1) = 4hVali
iqhbuf(2) = 4hd ca
iqhbuf(3) = 4hsees
iqhbuf(4) = 4h= %6
iqhbuf(5) = 3hd&n
call prtfi(stdout,fplog,cqhbuf(1:19),numok)
```

At the beginning of the routine the following code was generated:

```
integer      iqhbuf(20)
character*80 cqhbuf
equivalence  (iqhbuf(1),cqhbuf)
```

This is anything but portable, but it causes the compiler to store the string (only when in Hollerith) in the code segment. The code generated is actually a series of loads from the code segment into a local data array. The speed is as good or better than calling an external routine. The equivalence used is extremely non-portable and almost guaranteed not to work on most machines due to alignment problems. However, this did the trick for STAT80 and all of the code was processed in this way.

The result is that a program just of 90,000 lines of code under HP-UX suddenly became a program over 120,000 lines of code on the HP 1000.

3 PORT/HP-UX

After having to do all of the aforementioned changes to STAT80 to port it to the HP 1000, I had serious doubts about that code *ever* working on a machine other than an HP 1000. I knew that this would be a good test of the PORT/HP-UX environment because if this code worked, then most anything should work.

3.1 Migration Analysis

Before you write your tapes and begin a migration, take the time to run HP's migration analysis program, MAU. This program will analyze your code for HP 1000 system dependencies, especially

in the area of calls to the system, DGL and AGP, IMAGE/1000, and DS/1000-IV. MAU will flag source lines with possible migration problems. The flagged lines will have one or more warning messages as follows:

MAU 0 This call is fully emulated.

MAU 1 This call is not emulated.

MAU 2 This call is fully emulated but might suffer from a performance degradation.

MAU 3 Certain options or parameters in this call may or may not be emulated.

MAU 4 This call is fully emulated but might have a slightly different interpretation on HP-UX.

MAU 5 This call exists on both RTE-A and RTE-6/VM but only complies to the functionality of RTE-A.

3.2 Loading the Files

I used the HP 1000 tape program, TF, with the "X" option to write the STAT80 files. The "X" option causes TF to write type 4 files in the UNIX *tar(5)* format. This tape is then read using *tar(1)* on the HP-UX machine. This works quite well except that TF will not convert type 3 files to the *tar* format. The PORT/HP-UX environment has a utility program, *fmpupd(1)*, which is used to convert between HP-UX and FMP files.

Unlike FMP files on the HP 1000, a file under HP-UX has no extra information associated with it. In order to support FMP files under PORT/HP-UX, HP has devised a scheme that utilizes a "companion file" in a parallel directory that contains the information on file type, record length, security code, creation time, exclusive access flag, record count, end-of-file position, etc. These companion files are quite small and the parallel directory in which they reside is named *.fmp*. This naming convention keeps the directory name from appearing in directory listings. For example, if you have an FMP file called */DAVE/MYFILE*, then it will have a companion file called */DAVE/.fmp/MYFILE*.

The *fmpupd* program handles the creation of these companion files. Now there is a tape handling program, *rtetar(1)* that is designed to read and write FMP files in the TF tape format⁴.

STAT80 uses two ASCII type 2 files and two binary, type 4 files. The type 2 files were converted using the *fmpupd* program. Note that when converting type 2 files, you have to make sure to specify the record length, etc., otherwise it will not work at all. The type 2 files are accessed via *FMPSETPOSITION* and *FMPREAD* as they were generated on the HP 1000. I didn't worry about extents and there were no problems. I don't know how extents are handled under PORT/HP-UX, or if all files are considered to have only one extent. Nevertheless, I left the "X" option in the *FMPOPEN* call so that all extents would be read.

The binary files were re-created under PORT/HP-UX using our utility program, *MAKTBL*. This program reads an ASCII grammar file for STAT80 and then produces a binary file containing the parse tables for the grammar. The second binary file is a site installation file and is generated the first time STAT80 is run. All of these files are handled with FMP calls.

In April of 1986, I had quite a bit more work to move all of the files into the FMP directory structure from the HP-UX file structure. This is mainly because the *rtetar* program wasn't available yet. Now this process should be painless. However, I must give one caution, HP-UX file names are limited to 14 characters whereas HP 1000 CI file names may be much longer. If this is the case on your

⁴The TF tape format is really just a modified *tar* format. *Tar* can perform a file extraction from any TF tape.

system, I suggest that you perform the file renaming on the HP 1000 before going to HP-UX as you might save yourself a lot of grief.

3.3 FTN7X

Once everything was loaded, it was time to begin compiling the code so see what would happen. Before compilation, I ran the migration aid, *ftncvt*. This is a shell script that scans FTN7X FORTRAN files and performs various automatic conversions to PORT/HP-UX. It can convert FTN7X statements, compiler directives, and the FTN7X control line. This was to comment out the CDS, ALIAS, and EMA directives as well as convert the FTN7X control line. It has a mode that allows it to perform “unsafe” conversions as well as “safe” conversions. I didn’t allow it to touch any “unsafe” conversions. This all ran quite smoothly.

PORT/HP-UX provides a FORTRAN 77 compiler script, *ftn7x(1)*, that simplifies compiling programs from an HP 1000. This script correctly specifies the external types for all PORT/HP-UX system calls. When moving code to HP-UX using the PORT/HP-UX system, you definitely want to use this script.

The compilation took awhile, but remember that this is April 1986, long before a production version of the compiler was available. But lo and behold, everything compiled successfully.

3.4 Building the Executable

After getting everything compiled, it was time to try and load STAT80. Also use the *ftn7x* script for this as it will specify the appropriate PORT/HP-UX libraries. Well, it loaded correctly and when I ran it—it died a horrible death with segmentation violations, core dumps, and other typical UNIX diagnostics.

A segmentation violation usually means that you have walked on code somewhere. The other typical UNIX fatal error for programs is “bus error” which usually means that you referenced too far into an array or some type of bounds error. Nevertheless, my optimism got to me as I didn’t compile the program with the debug option. Make sure that you do this on the first try, otherwise, you might end up doing it anyway. On the bright side, I had just gotten to the migration center and I had scheduled a whole week so I couldn’t do everything the first two days. Besides, this gave me a chance to work with the debugger, *xdb(1)*.

The debugger on the HP 9000 Series 300 and 500 is typical UNIX, line oriented and so on. However, the *xdb* debugger was a real treat as it combines the windowing of the HP 1000 DEBUG program with all of the features of the UNIX debugger. The result is a real nice debugging environment that utilizes three windows. The top of the screen is a window into the current source file, the bottom of the screen is where XDB commands are entered and the program input and output goes, and the middle of the screen is a line in inverse video that indicates the current file, procedure, and line number. Within the source window, a “>” points to the current location.

I spent the better part of two days in the debugger, displaying trashed values and a lot more, but still with no clue as to what was happening. So when all else fails, I sat down and read the PORT/HP-UX documentation, carefully this time⁵.

And there it was, a **USER ERROR**. STAT80 assumes that integers will have 4 bytes and used the “FTN7X,J” header on the HP 1000 to make sure of it. However, the *ftn7x* script generates

⁵ Again, because this port took place long before the first machine was ever shipped, documentation was changing on a daily basis, especially the PORT/HP-UX documentation.

the `$$SHORT` compiler directive so apparently the `ftncvt` script doesn't completely work with the `ftn7x` script. I should have caught this much earlier. Anyway, another half a day compiling and then it linked and ran right away—and everything appeared to work. So another caution, *read the documentation for PORT/HP-UX carefully before starting any migration*. It will save some frustration.

One thing I noticed immediately about the executable was its size—it was **huge**. The size of STAT80 on the HP 9000 Series 550 is approximately 1.5 Mb, but on the Series 840 it was almost 5 Mb. Once again, the `ftn7x` script was the culprit. It specifies the “-K” option to the FORTRAN 77 compiler which means to save all local data values in the static data area rather than dynamically allocate them on the stack. STAT80 has about 650 subroutines and functions and that adds up to a lot of local data. The HP 1000 compiler directive, `$$CDS ON`, should be enough to flag the `ftn7x` script to suppress the “-K” option. Compiling without this option made the executable a more respectable size, but still quite large. I was told that this was due to the emulation library causing “the world” to be loaded with the program, and wouldn't be a problem when the machine was released. I didn't check it further, because the release version of STAT80 on the Series 840 was done in native mode.

3.5 Compatibility with HP-UX

The PORT/HP-UX environment is really a closed world. The FMP files can *only* be accessed correctly using the FMP library. A regular FORTRAN OPEN statement will not work correctly for an FMP file and neither will FORTRAN I/O statements. Furthermore, the FMP library routines cannot be used with regular HP-UX files. Any data in HP-UX files must first be converted using the `fmpupd` program before processing.

3.6 Performance

The next step was to do some comparisons between the HP 1000 STAT80 and the PORT/HP-UX version. The standard benchmark I use to compare different versions of STAT80 is a simple procedure written in STAT80's internal *proc language* that causes STAT80 to execute approximately 5600 commands. This is roughly equivalent to an extremely large command file. An unloaded HP 1000 A900 runs this procedure in 80 seconds. The PORT/HP-UX version of STAT80 ran it in 20 seconds, a full four times faster than the A900. This benchmark is good as a compute intensive benchmark for integer and floating-point operations as well as some character handling; however, it does not test I/O in any way.

The I/O under PORT/HP-UX seemed slow, especially output to the terminal. Some I/O degradation should be expected as there are three layers of library routines between the program and the kernel; e.g., the FMPWRITE routine calls routines from the level 3 library, which in turn calls the level 2 library. There is also probably a lot of overhead to performing the FMP file operations rather than just blasting a stream of bytes as would normally be done under UNIX.

4 Native Mode

In October of 1986, I did a second conversion of STAT80 to the Series 840; this time in native mode. I wrote a tape using `tar` on our HP 9000 Series 550 that was easily unloaded on the Series 840. The source code for STAT80 is written in Ratfor and the `ratfor(1)` preprocessor is supplied with

all HP-UX computer systems. Rather than use the HP-UX *ratfor* I brought my own version and installed it on the Series 840 in about an hour.

The only code that had to be changed dealt with retrieving arguments from the command line; everything else was consistent with the Series 550. All of the HP-UX computer systems use the IEEE floating-point format; therefore, no constants need to be changed between the Series 300, Series 500, and Series 800 machines. Furthermore, every binary file I have tested, with integer and/or floating-point values, has worked perfectly on all HP-UX machines.

Even though the conversion in native mode was rather smooth, I still took about 2 weeks to complete the conversion because this was to be the released version of STAT80 and it required more profiling and testing. Statware has developed a validation suite for STAT80 that is used on all conversions⁶ before they are released for shipping.

4.1 Compatibility with PORT/HP-UX

Unfortunately, there is almost no compatibility between the native version STAT80 and the PORT/HP-UX STAT80. Only ASCII files are interchangeable and these still must be converted back and forth using the *fmputd* program. There is no compatibility between the binary data files.

4.2 Performance

Next I used the same STAT80 procedure for benchmarking the native mode version. It was blindingly fast—only 8 seconds versus the 20 seconds for the PORT/HP-UX and 80 seconds for the HP 1000 A900. For comparison purposes, the following table lists results for several different machines running this benchmark (rounded up to the second).

<i>STAT80 Performance Benchmarks</i>		
<i>Machine</i>	<i>System</i>	<i>Benchmark</i>
HP 1000	A900	80 seconds
HP 9000	320	26 seconds
HP 9000	550	40 seconds
HP 9000	840 (PORT/HP-UX)	20 seconds
HP 9000	840 (native)	8 seconds
VAX	8600 (VMS)	11 seconds
Harris	H1000	12 seconds
IBM	4381-13 (VM/CMS)	13 seconds

The HP 9000 Series 840 in native mode was the fastest machine tested and this version of STAT80 was built without compiler optimization enabled. With optimization, it should be even faster.

5 Summary

Both the HP 1000 and HP 9000 Series 550 versions of STAT80 were converted to the HP 9000 Series 840 with a minimal amount of effort. I expected the native mode conversion to be easy; however, I really never thought that the HP 1000 conversion would work—and I was pleasantly surprised.

⁶The validation suite was not run on the PORT/HP-UX version of STAT80.

The only changes to the HP 1000 version of STAT80 to get it running under the PORT/HP-UX environment was the setting of several machine constants, mostly dealing with the storage of floating-point values. Otherwise, *no* changes were necessary to migrate it to the Series 840.

The test I wasn't able to do was a conversion from the HP 1000 version of STAT80 to native mode. Since the internal design of STAT80 favors UNIX this would really be the same as the Series 550 to Series 840 conversion, but with a lot more work. What I learned is just what HP has been saying, the PORT/HP-UX facility will allow HP 1000 programs to be quickly migrated to the Series 840 and still achieve better performance than the HP 1000 A900⁷. There are two key requirements we used when migrating to HP-UX;

1. having the same applications on the new machine be able to run the same data so that there is no serious down time due to conversions, and
2. achieving better performance.

The PORT/HP-UX environment allows most HP 1000 users to make a smooth transition to the HP-UX community. The final goal should be a transition to native mode for portability and performance criteria.

⁷STAT80 isn't a *real-time* program; therefore, these benchmarks should not be applied to real-time situations.

A Table Driven Plot Program for Radar Data

Donald Leslie
Raytheon
PO Box B
White Sands, NM 88002

Introduction

This plot program is designed to support the analysis of radar instrumentation data.

A Quick Look data reduction program already existed at the time that design was begun. This program performs no analysis functions. The reduction program reads a message from a data tape and then decodes and prints it. The only selection is by message type and interval (i.e. time range).

It is necessary that a user be able to set limits on any parameter in the data. For example the following may be required :

- Range is 0, 30 km.
- Azimuth is 6000, 1000 mils (Note : 6400 mils = 360 deg)
- Only inbound targets (negative range rate)
- Only targets flagged as jammers

Three shapes of plots were required : rectangular, polar and circular. The purchasable plot software that was considered did not easily do other than rectangular plots. In addition a pre-processor would still be required to filter the data.

An additional requirement was that the user interface be as simple as possible, to allow the program to be run by a data clerk or an analyst who did not know the physical layout of the data.

It was decided that each message type would be described by a disc file. The file contains the mnemonic for the message and the field names and their type. The program reads these files at initialization.

In my environment there are two kinds of data, track data and non-track data. Track data has an identifier field whereby records can be associated. All records with the same identifier are displayed as connected points. Non-track data is displayed as discrete points. The program allows only one track message type per plot. A number of non-track sources are allowed. Non-track data can be plotted along with track data or alone.

The program consists of three phases :

- 1) User interface
- 2) Sorting and filtering of data
- 3) Plotting the data

Hardware / Software Environment

The program runs on a HP1000F computer running RTE/6. EMA is used and currently requires a partition of 154 pages. AGP is used to do the graphics calls. The following devices are currently supported.

- 7550 plotter
- 7475 plotter (incomplete)
- 2623 terminal
- 2390 terminal
- 2648 terminal
- 256x printer

User Interface

User input is requested via menus. The first menu requests what and how to plot. The second requests the message types to be plotted. Then for each message selected a menu will be displayed requesting the fields to be plotted and the fields for filtering the data. A sample session is given at the end of this paper.

The program is designed to be driven from tables. For each message type there is a disc file which describes the data. A file has the following format :

mnemonic
field name_i, field type_i i=1,number of parameters

The initialization function of the program reads these files and builds tables to process the data. For example the A0801 (track file) and the C010B (scan message) yield the tables below.

<u>Defined_Src</u>	<u>Source_symb</u>
A0801	at
C010B	sc

Prm_Nam

fn, fn, tn, tfa, rmg, rdot, az, el, azd, tq, lid, tsrc, sdsr, flg...

Prm_B_L

b , b, l , l , b , b , b , b , b , sl , sl , sl , sl ...

Defined_Src contains the message names.

Source_Symb contains the mnemonics used in the menus.

Prm_Nam contains the names of the fields in the message. The names are in order of their position in the data.

Prm_B_L contains the bound types for the fields, where 'b' is an arithmetic bound, 'l' is a list of arithmetic values and 'sl' is a list of character strings.

Describing data this way has several advantages. If a message is changed the menu does not have to be updated, only its file description. Also the user does not have to know where the field is located in the message. Data for a plot is likely to contain a number of message types. A field may be at a different location for each type of data.

The two fields shown as 'fn' in Prm_Nam do not appear in the data. They are place holders for values computed from other arithmetic fields. It is possible to write functions in reverse polish to compute the value for plotting. For example :

To plot the azimuth error of a source known to be at 100 miles, one could enter az,100,- .

To plot the rms value of azimuth and elevation, one could enter az, **, el, **, +, sqrt.

The operators +, -, *, /, **, sqrt, min, max, log10 are currently supported.

On completion of the selection process a set of tables is created which is used by the next phases. They contain the locations in the definition tables of the symbolic parameters entered by the user.

For example the user has requested the A0801 track file as well as the D2 raw data. The plot is to be polar (Range vs. Azimuth). For the A0801 data, track source and range rate are specified as filter parameters. The allowed track sources have the mnemonics 'sd' and 'cw'. Range rate is specified to restrict the plot to only inbound targets.

This results in the following tables :

<u>Selected_Src</u>	<u>S_Offset</u>	<u>Num_Prm</u>	<u>T_Off</u>	
A0801	1	4	3	
D2	6	2		
<u>Selected_Prm</u>	<u>P_Offset</u>	<u>Flg_Vec</u>		
Rng	5	Lea		
Az	7			
Tsrc	12		{ for A0801 }	
Rdot	6			
Rng	3		{ for D2 }	
Az	4			
<u>Prm_List</u>	<u>Limit_Type</u>	<u>Prm_Bnd</u>	<u>B_vec</u>	<u>Sl_vec</u>
Rng	d	b	-9999,-9999	
Az	s	b	4800,1600	
Tsrc	s	sl		sd,cw
Rdot	s	b	-999,0	

S_Offset contains the indicies in Defined_Src for the message type.

P_Offset contains the indicies in Prm_Name for the parameters.

After the tables are built the users choices are displayed. This allows the user to validate the selections. At this point one can proceed to the sorting process or return to the selection menus.

Sorting and Filtering

This phase reads the file and validates the messages read. The process is best illustrated by the pseudo code below.

```

Do While (.not. Eof )
  Read a record
  Get the record type and time
  If ( current scan is in scan bounds ) Then
    If ( message time is in time bounds ) Then
      If ( type is selected .or. type is scan msg ) Then
        Split record into component fields
        If ( all flags specified are in the message ) Then
          If ( all fields specified are within bounds ) Then
            Message is valid
          Endif
        Endif
      Endif
    Endif
  Endif
End Do

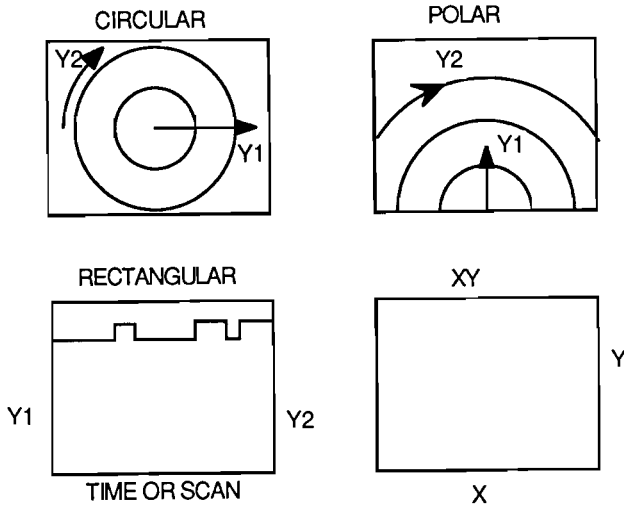
```

Valid messages are moved into an EMA file. To simplify the plot function the data in EMA is chained. Non-track data messages are chained by message type. Track data messages are chained by track number. This allows all messages with the same track to be plotted quickly. All scan messages are also chained together for plotting.

If track data is selected, a list of all track numbers found is displayed at the end of sort process. The user is allowed to select some or all of the tracks.

Plotting the Data

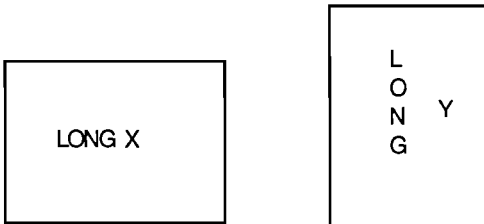
At the completion of sorting, the data is plotted. How the data is displayed is dependent on the shape selected. There are four possible shapes selectable : circular, polar, rectangular and xy.



There are two paper sizes and orientations for a plot.

- 3-> 11" x 17"
- 4-> 8 1/2" x 11"

X->Long axis along X axis
Y->Long axis along Y axis { only for 7550 plotter }



The following is true for all plot formats.

- Track data is plotted as connected points. The end points are labeled with the track number.
- Non-track data is displayed as discrete points.
- A title is displayed at the top of the plot.

In a rectangular plot one or two parameters may be selected. The data plotted and the y-axis labels are shown in one color. The x-axis is time or scan.

An xy plot is one parameter against another.

A polar plot is always that of 3200 mils/180 deg. If a track source is present in the data the user has two options. The pen color is either selected according to track source or is a random sequence of the track numbers.

In a circular plot the first parameter specified is used as a magnitude. The second is used as a rotation. The rotation is generated by dividing the value at a point by the range of the parameter.

When the plot is complete a replot menu is displayed. There are four choices possible.

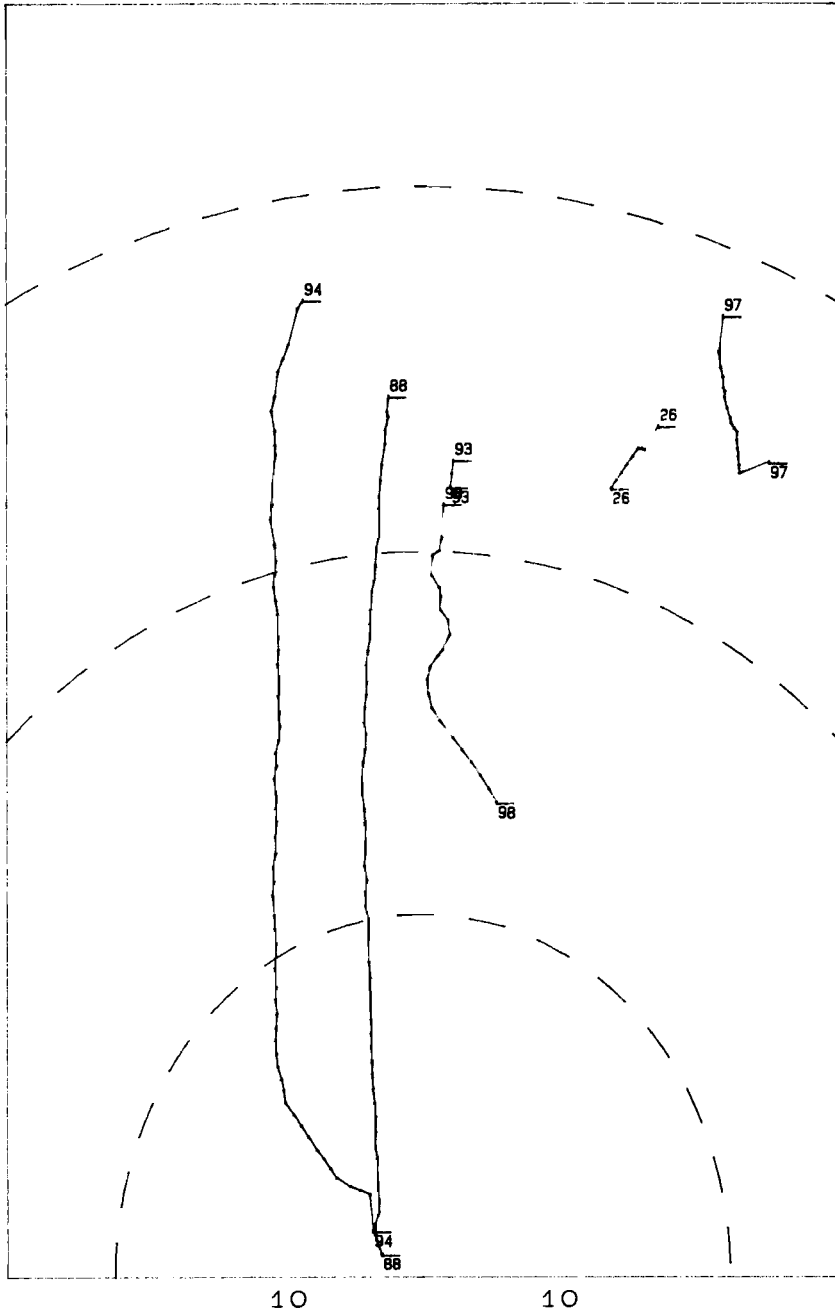
The user can exit the program. One can replot with the same or different Lu but no other changes. The plot limits or plot format can be changed. Since the data has been moved into EMA according to the selection process, the limits can only be further restricted. Increasing the limits will not provide more data. Reselecting limits or format returns the user to the selection menus. The choices made for the previous plot will be displayed in the menus. Lastly a new file can be requested. This allows a new file to be read and sorted.

Summary

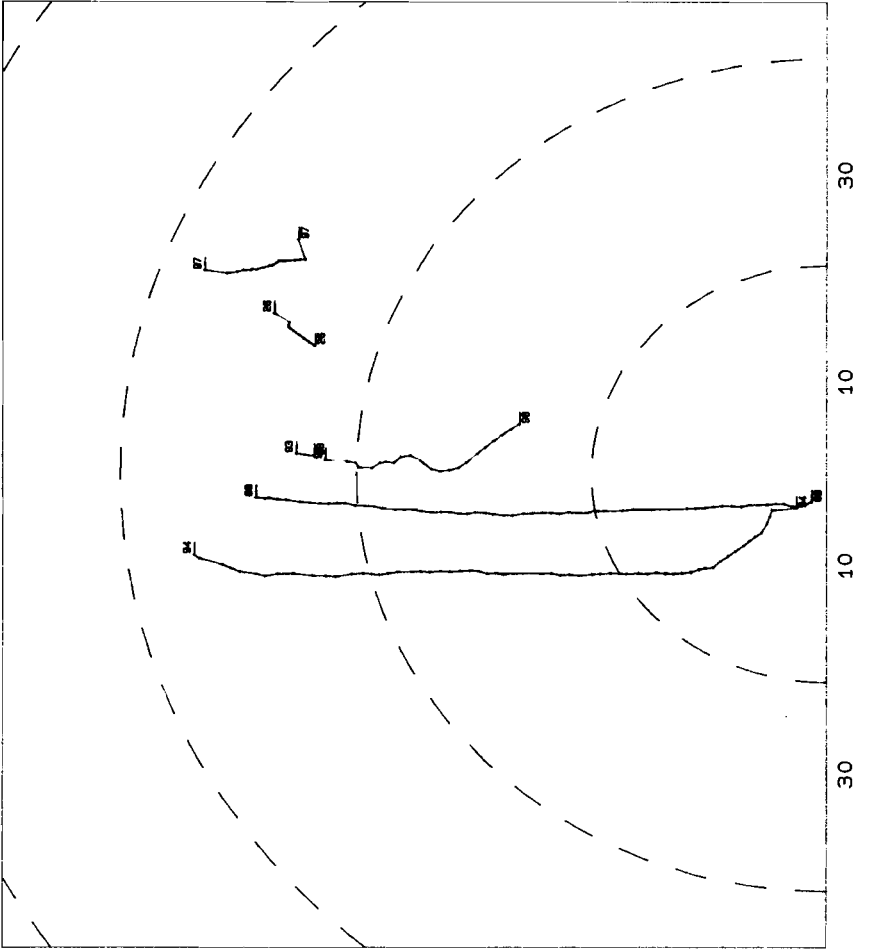
This program has proven very useful in performing analysis of radar data. It should also be useful in other environments. A copy of the source is available on the swap tape for this conference. The following items were specific to my environment and would have to be changed :

- special processing for scan messages
- the handling of launcher data
- default value for range on tracks with jammer data
- FPS-16 range radar processing
- processing of azimuth in mils

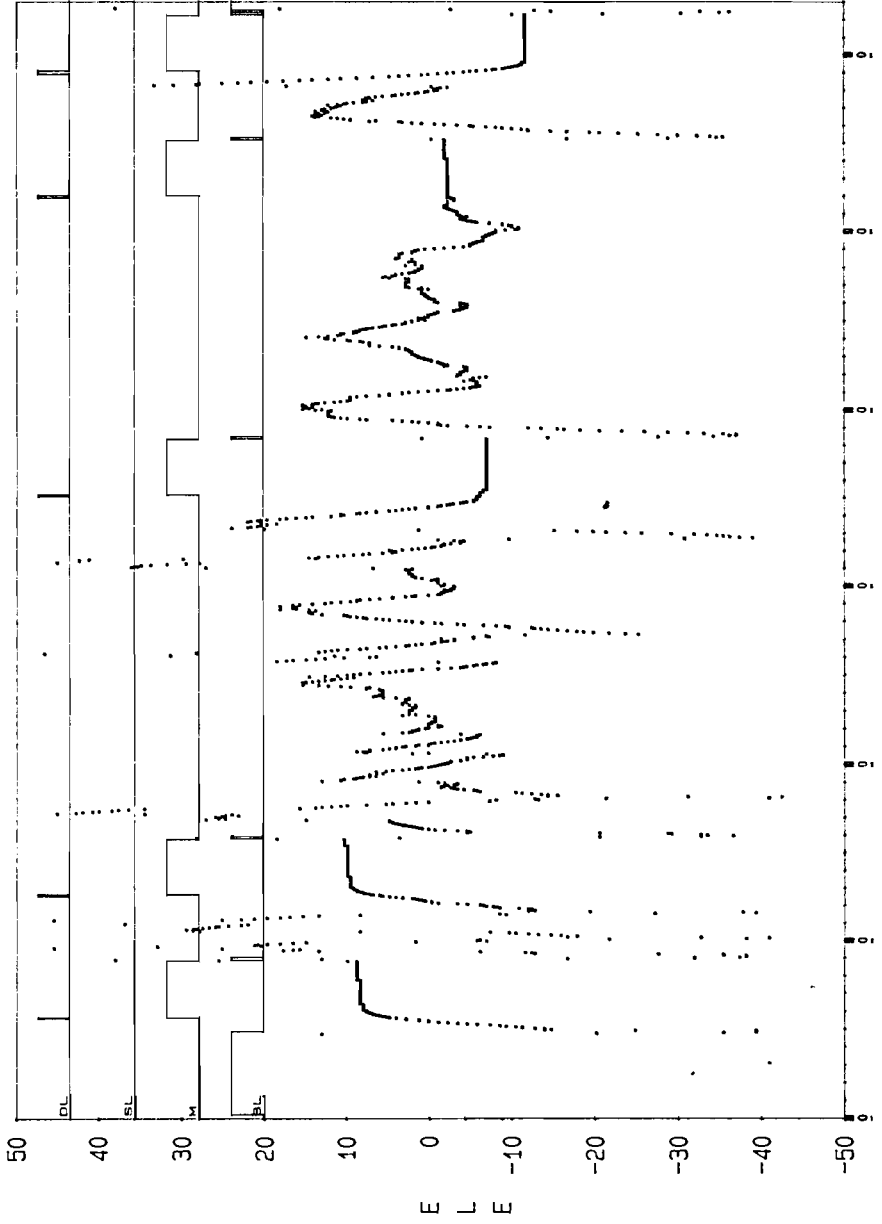
POLAR PLOT - RANGE = 0,70 KM. AZ. = 4800, 1600 MILS LONG Y AXIS



POLAR PLOT - RANGE = 0.70 KM. AZIMUTH = 4800. 1600 MILS LONG X AXIS



ELEVATION ERROR RAW DATA (EL, CONSTANT, -) AND RADAR EVENTS



An Example of the User Menus

PLOT SETUP...
VERSION 2.0

Press ENTER when done
Press TAB for next field

FILENAME : QLLIST

RECTANGULAR, POLAR,CIRCULAR, XY (R/P/C/XY) : P

SCAN/TIME/NONE (S/T/N) : T

LOW (S/T) : _____ HIGH (S/T) : _____

PAPER SIZE (3/4) AND LONG AXIS (X/Y) : 3X

PLOTTER LU : 36

EXIT (Y/N) : N

The following menu is displayed next :

Press ENTER when done
Press TAB for next field

Track source :
ADP track file, CWAR track file, PAR track file, SDP air picture
HPI D1

(AT,CT,PT,SA,D1) : AT

Non-Track Source (s) :
CWAR return file,CWAR TDL raw,CWAR DSP,PAR clutter map
PAR return file, FPS-16 (channel 1/2), HPI D2

(CR,CTR,DS,CM,PR,F1,F2,D2,L1) : _____

MAX OF 4 TRACK/NON-TRACK SOURCES MAY BE ENTERED

Flags (F) : _____

The following menu is then displayed since AT (A0801) was selected as a source :

Press ENTER when done
Press TAB for next field
A0801

Axes can be (RNG,RDOT,AZ,EL,AZD,TQ)
Limits can be Static or Dynamic (S/D)

Y1 axis RNG Y1 Limit D
Y2 axis AZ Y2 Limit S

RNG RDOT AZ EL AZD : RDOT
TQ LID TSRC SDSRC

FOL LEA J JH JT : _____

Tracks labeled with (TN, TFA) : TN

fn Y(1) : _____

fn Y(2) : _____

The following are then displayed to request limits :

SPECIFY LIMITS FOR AZ

Y low limit is : 4800

Y high limit is : 1600 { Defaults for Polar }

SPECIFY LIMITS FOR RDOT

Y low limit is : -999

Y high limit is : 0

For each source, the following summary is displayed for the selections made :

Reselect: N

NUM_SRC'S : 1

Source: A0801

NUM-PRMS : 3

FLAGS : LEA

--filters----- limits-----

RNG	0	
	0	{ 0,0 since range is dynamic }
AZ	4800	
	1600	
RDOT	-999	
	0	

If at this point changes need to be made, set RESELECT = 'Y' . This returns the user to the first menu.

Once the program searches the data file and extracts the information requested, it will then display all Track Numbers found and request the users requirements. i.e.

The following tracks were found

1 15 24 36 102 :

A0801 - Tracks - Some,All? S/A : _____

If S is entered the following will be requested :

> Number of tracks ? : _____

Track number : _____

Track number will be repeated for the number of tracks specified.

A Suite of System Generation Tools

**Christopher Nelson
General Foods Corporation
250 North Street
White Plains, NY 10625**

For most system managers, operating system generation is done once in a while with trepidation. The complexity of an answer file makes errors likely. Even an answer file that 'gens' may have mistakes in it that prevent system boot-up or, worse, corrupt the system beyond use.

Hewlett-Packard supplies the basic utilities (RTAGN, BUILD, CSYS, INSTALL, etc.) and makes recommendations on procedure (such as 'backup the existing system before generating a new one'). However, there are no standard tools that give the system manager high confidence in the safety and success of his new system generation.

In managing a development system, I have been required to re-gen frequently to test new devices or software sub-systems. This frequency has led to boredom and a measure of carelessness. To keep myself from making dangerous mistakes, I developed tools and techniques that help automate, debug, and verify system generation, installation, and start-up. This paper discusses these techniques and tools and the environment for which they were designed.

1 Introduction

Several terms may need clarification. LOCALIZATION refers to the adaptation of a known-good answer file, the PRIMARY, to fit the immediate local need. GENERATION sometimes refers to the entire process of getting a new system up and running but as used here will usually refer to the actual execution of the generator against an answer file. INSTALLATION is the process of putting the new system files into a bootable area of the disk (or other media). BACK-UP will refer not only to actually copying a system to a removable media but also to having disk resident fall backs.

2 Environment

The tools described here were developed for RTE-A starting at the A.85 revision. They take full advantage of IF/THEN/ELSE/FI and other CI programming constructs. The system is booted from a FMGR boot cartridge, LU 17, which is named BT. The rest of the system is located on CI volumes. Critical directories (/PROGRAMS/, /SYSTEM/, etc.) are all located on one volume called the SYSTEM VOLUME. Operating system relocatables are found in /RELOCS/RTEA/; VC+ files and special RPL's required for NS have been merged into this directory. Product relocatables (DS, NS, PCIF) are stored under their own global directories as shipped from HP. System generation files and tools are located in /SYSTEM/GENERATION/.

Two sets of System, Snap, and Boot Command files reside on the boot cartridge. One set references WELCOME1.CMD and starts up a simple system with only the system console enabled. The other set references WELCOME2.CMD to start up a fully operational system; this is the default.

3 Localization

Localization is perhaps the most complicated part of creating a new system generation. While it is certainly possible to manually edit an answer file to support a new device or sub-system, it can be extremely tedious to do so. The process can be automated in two steps. First, an answer file that includes more than you need can be marked with keywords that allow for an automated search. Second, a command file can be built that 'knows' the relationships between the keyworded parts of the answer file.

Listing 1 is an excerpt from an answer file with keywords added to allow for the automatic localization of disk model. For example, PRI7946 indicates that the primary disk is a model 7946. Notice that two separate parts of the answer file, the Table Generation Phase and the Node List section, must be modified; this is the type of thing that makes manual localization difficult. Listing 2 is an excerpt of LOCALIZE.UTL, a command file that knows how to use the disk keywords to install the correct model. LOCALIZE.UTL is usually scheduled by another command file which will be discussed later.

Listing 1 - Excerpt from keyworded answer file

```

* HP-IB #1 - Discs and magnetic tape          select code = 27b
* -----
IFT,RTEA/%ID*37,SC:27B
*
* BUS CONTROLLER LU                          HP-IB address 36
*                                          LU 9
DVT,,,LU:9,TO:2000,DT:77B,TX:0,DX:1,DP:1:36B,PR:0
*
* 7946/12/14 CS-80 Disc                      HP-IB address 0
*                                          LU 17-23
*
* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* Note: The default values for all but the driver parameters
* are the same for all CS-80 disks. Rather than include three
* sets of DVT's (one each for 7946, 7912 and 7914) the default
* for 7946, 7912 and 7914) the default values for smallest disk
* that will accommodate the LU are referenced and all driver
* parameters are given explicitly.
* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*
*-PRI7946 DVT,RTEA/%DD*33,M7945_CF:0,LU:17,DP:1:0:0:0:0:0,-
*-PRI7946          DP:6:400:48:0
*
* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* Note: Disk cache for CS/80 cartridge tape at LU 24 is
* between LUs 17 and 18. It takes 6 tracks, 288 blocks.
* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*
*-PRI7946 DVT,RTEA/%DD*33,M7945_CF:0,LU:18,DP:1:0:0:0:0:19488,-
*-PRI7946          DP:6:1358:48:0
*-PRI7946 DVT,RTEA/%DD*33,M7945_CF:0,LU:19,DP:1:0:0:0:1:19136,-
*-PRI7946          DP:6:190:48:0
*-PRI7946 DVT,RTEA/%DD*33,M7945_CF:0,LU:20,DP:1:0:0:0:1:28256,-
*-PRI7946          DP:6:334:48:0
*-PRI7946 DVT,RTEA/%DD*33,M7945_CF:0,LU:21,DP:1:0:0:0:1:44288,-
*-PRI7946          DP:6:2229:48:0
*
* Use ONLY with 7912 or 7914 ('Star out' if you have 7946).
*-PRI7912 DVT,RTEA/%DD*33,M7912_LF:0,LU:22,DP:1:0:0:0:3:20208,-
*-PRI7912          DP:6:821:48:0
*
* Use ONLY with 7914 ('Star out' if you have 7946 or 7912).
*-PRI7914 DVT,RTEA/%DD*33,M7914_LF:0,LU:23,DP:1:0:0:0:3:59648,-
*-PRI7914          DP:6:5413:48:0
*
* Use ONLY with 7958 ('Star out' if you have 7946/12/14).
*-PRI7958 DVT,RTEA/%DD*33,M7958_CF:0,LU:23,DP:1:0:0:0:3:59648,-
*-PRI7958          DP:6:4376:48:0
*
* Compatible cartridge tape cache (7946/12/14) HP-IB address 0
*                                          LU 24
DVT,RTEA/%DD*33,MTAPE,LU:24,DP:1:0:400b:100000b,-
DP:4:0:19200:0:0:0
*
*
* For all disks
*-PRI7946 NODE,17,18,19,20,21,-
* For 7912/14/58
*-PRI7912 22,-
* For 7914
*-PRI7914 23,-
* For 7858
*-PRI7958 23,-
* For all disks (CTD)
24

```

Listing 2 - Excerpt from localization utility

```
*-----*
* Primary disk is built of incremental LU's. Larger disks require
* volumes from smaller disks.
IF IS $1 = 'PRI7946'; THEN
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7946 //q|ER'
    RETURN
FI
IF IS $1 = 'PRI7912'; THEN
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7946 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7912 //q|ER'
    RETURN
FI
IF IS $1 = 'PRI7914'; THEN
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7946 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7912 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7914 //q|ER'
    RETURN
FI
IF IS $1 = 'PRI7958'; THEN
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7946 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7912 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7958 //q|ER'
    RETURN
FI
IF IS $1 = 'PRI7937'; THEN
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7946 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7912 //q|ER'
    EDIT $ANS,'SE AS OF|1 $ x/*-PRI7937 //q|ER'
    RETURN
FI
```

There are several ways in which keywords may be used. In the simplest case, a device or sub-system is independent of all others. Here, LOCALIZE.UTL would simply change all occurrence of '*-keyword' in the answer file to the null string.

The second possibility is that one device or sub-system includes another, for example disk models with incremental space. In this case, as shown above, LOCALIZE.UTL accepts the keyword for the largest device or sub-system and knows to bring in the rest.

The other cases involve combinatorics and dependencies. For example, NS requires XMB only for a model 900 CPU. In these cases, there are multiple keywords per line in the answer file and LOCALIZE treats the keywords as independent.

The ultimate step in automating the localization process is to build a command file that brings all local systems and devices into the answer file and performs appropriate file manipulations. Listing 3 shows a command file that does just that.

LOCALIZE.CMD accepts the name of a system to be localized as its first parameter. This system name is the root name of the data file which is input to the localization procedure, the generation answer file which is output from the localization procedure, and the system, snap and list files that are output from the generation procedure. These files are distinguished by their file type extensions (.DAT, .ANS, .SYS, .SNP, and .GLST respectively). The localization data file contains a list of keywords and optional comments to which LOCALIZE.CMD will apply LOCALIZE.UTL; Listing 4 is a sample.

Referring to Listing 3, LOCALIZE.CMD does several things besides applying LOCALIZE.UTL to the primary for each keyword. First, it checks to see if a system name has been specified. (The CI variable \$NULL has been set to the null string in the systems global login command file with "SET NULL =".) If no system has been specified, an error message is printed and the command file terminates. Next, if a system has been specified, LOCALIZE checks to see if its data file exists. (For the rest of the paper, LOCALIZE will mean LOCALIZE.CMD unless otherwise specified.) If the data file does not exist, an error message is printed and the command file terminates.

If the system data file exists, LOCALIZE checks to see if a system answer already exists. If one does, it backs the file up by renaming it. If the system answer file does not exist (or after it has been backed up), PRIMARY.ANS is copied so it may be edited to produce the new system answer file. LOCALIZE.UTL is edited to reflect the name of the system to be processed.

Not having to edit an answer file would be of small use if one had to review the whole thing to see what it included. The next step in LOCALIZE addresses this need; the localization data file is merged into the system answer file and turned into comments.

APPLY is a utility to apply a program or command file to a list of arguments listed in a file.¹ It is used to allow LOCALIZE.UTL to process each keyword in the system data file.

¹ The first argument is the name of the program or command file (process) to be APPLY'd. The second argument is the name of the file to be processed. The third argument is a switch that tells APPLY if the process is a command (TRansfer) file. APPLY reads one line at a time from the input file and appends it to the process name. If the process is a program, the resultant string is passed to FMRunProgram directly. If the process is a command file, the string is first prefixed with "CI,".

Listing 3 - Localization Command File

```
*****
* File: /SYSTEM/GENERATION/LOCALIZE.CMD <870521.1240>
*
* Command file to localize PRIMARY.ANS
*-----
* Tell Localize.utl which file to process
IF IS $1 = $NULL; THEN
    ECHO
    ECHO 'A data file must be specified.'
    ECHO
    RETURN
ELSE
    IF DL,$1.DAT,,0; THEN
        EDIT Localize.utl,'f/$2.ANS/|k|-1| set ANS = '$1'.ANS|ER'
    ELSE
        ECHO
        ECHO 'Error specifying data file.'
        ECHO
        RETURN
    FI
FI
*-----
* Save answer file if it exists
IF DL,$1.ANS,,0; THEN
    CO,$1.ANS,$1_ans.bak,d
FI
*-----
* Copy PRIMARY into specified answer file
IF IS $2 = $NULL; THEN
    CO,PRIMARY.ANS,$1.ANS,d
ELSE
    CO $2,$1.ANS,d
FI
*-----
* Change file name in new answer file
EDIT $1.ANS,1 $ f/PRIMARY.ANS/|X/PRIMARY.ANS/$1.ANS/|ER
*-----
* Merge the data file into the answer file as documentation
SET MARK = 'KA|+|KB|-' ;* Mark lines to merge between
SET CMNT = 'SE RE ON|SE AS OF|:A+1 :B-1 X/([*])/* &1/'
EDIT $1.ANS,f/.DAT/|$MARK|m $1.DAT|$CMNT|ER
UNSET MARK
UNSET CMNT
*-----
* Bring in each local feature listed in $1.DAT
APPLY localize.utl $1.DAT,TR
*-----
* Restore localize.utl
EDIT localize.utl,'f/'$1'.ANS/|k|-1| set ANS = $2.ANS|ER'
*-----
* Tell the user LOCALIZE is done
ECHO
ECHO '$1'.ANS localized.'
ECHO
```

Table 1 - Answer File Localization Keywords

Category	Keyword	Description
RPL	400	Model 400 CPU
	600	Model 600 CPU
	700	Model 700 CPU
	900	Model 900 CPU
System Console Interface	CON_MUX	System console on multiplexer
	CON_ASYNC	System console on 12005
Primary Disk	PRI2480	Integral disk as primary disk
	PRI7912	7912 as primary disk
	PRI7958	7958 as primary disk
	PRI7946	7946 as primary disk
	PRI7937	7937 as primary disk
	PRI7914	7914 as primary disk
Secondary Disk	SEC7914	7914 as secondary disk
	SEC7946	7946 as secondary disk
	SEC7912	7912 as secondary disk
	SEC7937	7937 as secondary disk
	SEC7958	7958 as secondary disk
Reel-to-reel Tape	LU8_7970	Model 7970E tape drive
	LU8_7974	Model 7974/8 tape drive
System Printer Interface	LU6_HPIB	System printer on HP-IB
	LU6_MUX	System printer on multiplexer
System Printer Model	LU6_2563	Model 2563 system printer
	LU6_2932	Model 2932/4 system printer
Auxiliary Printer Interface	LU2_HPIB	Auxiliary printer on HP-IB
	LU2_MUX	Auxiliary printer on multiplexer
Auxiliary Printer Model	LU2_2563	Model 2563 auxiliary printer
	LU2_LASER	Laser printer as auxiliary printer
	LU2_2932	Model 2932/4 auxiliary printer
Plotter interface	PLT_HPIB	System printer on HP-IB
	PLT_MUX	System plotter on multiplexer
HP System Communication	DS	Distributed Systems
	DS2	Distributed Systems with 2 links
	NS	Network Services
Foreign System communication	RJE	Remote Job Entry

Next LOCALIZE.UTL is restored to its original state so it may be used independently if desired. Finally, the user is told that the localization process is complete.

It is important to note that because keywords are processed one at a time, LOCALIZE will not catch things like not specifying any disk or specifying two printers or interfaces for the system printer. These will come out later in the generation or installation procedures.

Listing 4 - Sample localization data file

900	A900 CPU
pri7914	7914 as primary disk
sec7937	7937 as secondary disk
con_mux	System Console on Mux
lu6_hpib	System printer on HP-IB
lu6_2563	Model 2563 printer
plt_hpib	Plotter on HP-IB
ab	Allen-Bradley interface
ds	DS/1000

4 Generation

The generation procedure is rather straight forward. Listing 5 is a command file, GEN, that makes it just a little bit easier in three ways. First, it automatically deletes existing system, snap, and generation list files. This is safe since the list file can be recreated from the answer file LOCALIZE backed up, and the system and snap files exist on the boot cartridge.

Second, GEN takes a system name as its first argument. (This is the same system name passed to LOCALIZE and later to the installation procedure.) One obvious use of this system name is to maintain generation information for slave nodes on a master node.

Finally, GEN takes a directory name as an optional second argument. If specified, this argument is the parent directory of the RTE-A relocatables. This allows multiple revisions of RTE-A to be maintained on the disk for support of other systems or during upgrade. The default directory for RTE-A relocatables changes from release to release to coincide with the current revision.

Listing 5 - Generation command file

```
*****
* File: /SYSTEM/GENERATION/GEN.CMD <870327.0853>
*
* Command file to run RTEA system GeNeration (RTAGN)
* Chris Nelson, CPC
*
* History:
* Changed LIST file to @.GLST so that 'pu @.lst.e' does not
* purge it.
* Changed order of $1 and $2. $2 defaults to /relocs_4*1
*
* Usage:
* $1 is the root-name of the answer file to process. If $2 is
* not given, an error is returned.
*
* $2 is the root directory for system and HP product
* sub-directories. If $2 is not given, /relocs_4*1 is used as the
* default.
*
* This command file uses the following CI variables. They are UNSET
* at the end of the command file.
*
* swd save working directory
* rd root directory for relocatables
*-----
* Remember current working directory.
SET swd = $WD
SET NULL =
*-----
* $1 may specify the prefix for the answer, system, snap, and list
* files. If not specified, it defaults to GFS_nm.
* Build the file names based on $1 or default.
IF IS $1 = $NULL; THEN
    ECHO
    ECHO 'System name must be specified.'
    ECHO 'Available systems:'
    DL,@.ANS
    RETURN
ELSE
    SET ans = $1.ans
    SET sys = $1.sys
    SET snp = $1.snp
    SET lst = $1.glst
FI
*-----
* $2 may specify the root directory for system and product
* relocatables to be used. If it is null then use /relocs_4*1 as
* default
IF IS $2 = $NULL; THEN
    set rd = '/RELOCS_4*1'
ELSE
    set rd = $2
FI
```

Listing 5 continued

```
*-----
* Check to see that the root directory for the relocatables
* exists. If it does not then abort.
IF DL,$rd,,0; THEN
  WD $rd
ELSE
  ECHO
  ECHO 'No such directory: '$rd'. Generation not attempted.'
  ECHO
  UNSET rd
  UNSET swd
  RETURN
FI
*-----
* Purge existing list, system and snap files
IF DL,$swd/$lst,,0; THEN
  PU $swd/$lst
FI
IF DL,$swd/$sys,,0; THEN
  PU $swd/$sys
FI
IF DL,$swd/$snp,,0; THEN
  PU $swd/$snp
FI
*-----
* Run the generator with reference to the original working directory
rtagn $swd/$ans $swd/$lst $swd/$sys $swd/$snp
*-----
* Return to the starting directory and UNSET CI variables
wd $swd
UNSET swd
UNSET rd
*-----
* Tell the user we're done
ECHO
ECHO ' System generation complete.'
ECHO
*
```

5 Installation

Installation is the first stage at which one might corrupt the existing bootable system. One may prevent this by backing up the existing system in a form that may be restored 'off-line' but this is time consuming. On a production system, one cannot afford the downtime to perform the backup. On a development system, one would rather get on with the test.

An alternative to an off-line backup is to not overwrite the existing bootable system. This is accomplished by installing the new system to the boot cartridge as a second set of system and snap files referenced by a second boot command file. Once proven, the new system may be installed

over the old default. If a problem is encountered booting the new system, the default system may be booted and another attempt made at generating a new system.

T_INSTL accepts a system name and installs the corresponding system and snap files on the boot cartridge for test as T_SYS and T_SNP. These are the files referenced by T_BOOT, the test boot command file. If the test system boots, D_INSTL is used to install the test system over the old default in D_SYS and D_SNP, these are the files referenced by SYSTEM, the default boot command file.

Listing 6 - Test installation command file

```
*****
* File: /SYSTEM/GENERATION/T_INSTL.CMD <830401.1411>
*
* Transfer file to install a new system for test
*-----
SET NULL =
* If no parameter is passed, default file names
IF IS $1 = $NULL; THEN
  ECHO
  ECHO ' System name must be specified.'
  ECHO
  RETURN
ELSE
  SET ANS = $1.ANS
  SET SYS = $1.SYS
  SET SNP = $1.SNP
FI
*-----
* Copy the system file to the boot cartridge
CO $SYS T_SYS::BT D
IF IS $RETURN1 <> 0;THEN
  ECHO 'Copy error '$RETURN1' at stage one, install aborted normally.'
  RETURN
FI
*-----
* Copy the snap file to the boot cartridge
CO $SNP T_SNP::BT D
IF IS $RETURN1 <> 0;THEN
  ECHO 'Copy error '$RETURN1' at stage two, install aborted abnormally.'
  RETURN
FI
*-----
* Copy the answer file to the boot cartridge
CO $ANS T_ANS::BT D
IF IS $RETURN1 <> 0;THEN
  ECHO 'Copy error '$RETURN1' at stage three, install aborted abnormally.'
  RETURN
ELSE
  ECHO
  ECHO ' Installation successful. Re-boot with '%bcd27t_boot' to use.'
  ECHO
FI
```

Listing 7 - MEMSYS.COMD

```
*****
* File: /SYSTEM/MEMORY/MEMSYS.COMD                <870330.1022>
*
* CI command file to build a memory based system.
*
* This command file uses $1.SYS and $1.SNP in the /SYSTEM/
* GENERATION/ directory to create $1.MEM in the /SYSTEM/MEMORY/
* directory.
*
* $1 is the root name of the .SYS and .SNP files to be used
* to build the memory based system. BUILD.IN is copied to BUILD.UTL
* and edited to reference these files. The HP utility program
* BUILD is run with BUILD.UTL as input. BUILD.UTL is purged as
* part of the cleanup.
*
* 870330 CLM
*-----
* Purge old memory based system file if it exists.
IF DL,$1.MEM,,0; THEN
    ECHO 'Purging old binary file.'
    PU $1.MEM
FI
*-----
* Copy BUILD.IN to BUILD.UTL and edit to reference $1 files.
CO,BUILD.IN,BUILD.UTL,d
EDIT BUILD.UTL,'SE AS OF|1 $ x/??/'$1'/|ER'
*-----
* Run BUILD passing it BUILD.UTL as the name of the input file.
BUILD,BUILD.UTL
*-----
* Purge BUILD.UTL
PU,BUILD.UTL
*-----
* Tell the user what to do.
ECHO
ECHO
ECHO 'MEMSYS done.'
ECHO
IF DL,$1.MEM,,0; THEN
    ECHO 'Use CSYS to copy the system to tape for later use.'
    ECHO 'Purge '$1'.MEM after use to save disk space.'
ELSE
    ECHO 'Could not create memory based system file. There may '
    ECHO 'not be enough disk space. Purge extraneous file and '
    ECHO 'try again.'
FI
ECHO
```

6 Backup and recovery

A two stage system installation procedure reduces or eliminates the need for an off-line backup during generation. However, a major revision warrants an off-line backup, and there is still the occasional disk crash or file deletion to be dealt with. If one can boot the system, one can restore

applications and data. However, if the system itself is corrupted, an off-line backup is the only alternative. This backup is in two forms: a bootable memory based system on tape, and an ASAVE of the boot cartridge and system volume. These can be combined on one 600' CTD which can be kept in a safe place away from the system.

Listing 7 shows MEMSYS.CMD, a command file which accepts a system name (as used in Localization, Generation and Test Installation) and creates a memory based system file. This memory based system assumes a minimum of 512K memory available and provides ASAVE, ARSTR, and other utilities. CSYS is used to copy this file to CTD with ASAVE delimiters. Finally, ASAVE is used to append a copy of the boot cartridge and system volume to the tape. With some variation, this procedure can be used to ship a generation for a new system.

To recover a crashed system, the ASAVE CTD is placed in the drive and the memory based system is booted off of it. At the memory system prompt, ARSTR can be run to restore the boot cartridge and/or the system volume. The system can then be booted from disk and application programs or data restored from file based (TF, FST) backups.

7 Welcome file structures

Two welcome files are resident on the system. WELCOME1.CMD sets the system time and configures the system console; this is the minimum start-up processing necessary to make any use of the system. T_BOOT schedules CI as START with WELCOME1 as the welcome file. In this way, the system manager may log onto the console and test the new generation. This file is also useful for booting a nearly idle system for complete backups.

WELCOME2.CMD is used to start-up a fully functioning system; WELCOME1.CMD is called to set the time and enable the console, then software sub-systems are started, other devices are configured, and user terminals are enabled. (By referencing WELCOME1 to set the time and enable the console, potential conflicts resulting from changing WELCOME1 but not WELCOME2 are eliminated.) The default boot command file, SYSTEM, schedules CI as START with WELCOME2 as the welcome file.

8 Sample Session

The preceding sections have discussed the tool suite roughly in the order they would be used to modify a system generation. This section will narrate the steps to add a second disk drive using the tools. Assume that the system has been created using the tools, the localization data file is named LOCAL.DAT, and that PRIMARY.ANS is current, and a 7937 disk drive is to be added as a secondary disk drive.

- Step 1. Edit LOCAL.DAT and add a line with SEC7937 as the keyword.
- Step 2. Execute LOCALIZE with 'localize,local'.
- Step 3. Execute GEN with 'gen,local'.
- Step 4. Execute T_INSTAL with 't_instal,local'.
- Step 5. Reboot specifying T_BOOT as the boot command file.²
- Step 6. Attempt to mount volumes on new disk. Initialize them if necessary.
- Step 7. Modify the welcome file to mount the new volumes automatically.
- Step 8. Execute D_INSTL with 'd_instl'.
- Step 9. Reboot specifying with the default boot command file.

9 Conclusions

The preceding section shows the addition of a disk drive with minimal effort and no need for off-line backups. The entire process could be completed in well under an hour. This can be contrasted with other recommended procedures which would take several hours at best. Among the factors contributing to this increased efficiency are starting with a known-good answer file, automating most of the tedious editing and file manipulation, and eliminating the need for an off-line backup in most cases.

Opportunities to extend this tool set include automatic updating of the boot command and welcome files, a localize command to merge in locally created answer file segments, speeding up the localize utility by implementing it in a compiled language, and automating backup tape construction. Hopefully, the files that make up this tool set will be available on the swap tape. It should be easy to add keywords to your own answer file and use them with it.

² Sometime before the next step, the new HP-IB card is put in the SPU cage with select code 26b and the disk is cabled to it and configured for HP-IB address 0.

RELATIONAL VIEW OF IMAGE WITH REAL ZIP

Stephen R Carter

Eyring Research, Inc.
1450 West 820 North
Provo, Utah 84601

INTRODUCTION.

Consider the following:

A quality engineer determines the interaction of two critical quality measurements.

A process engineer launches an investigation to determine the actual impact of several process conditions on product quality.

A product engineer is zeroing in on a way to shorten total in-process time.

Another quality engineer questions the accuracy of test methods used during production.

Each of these scenarios has a common thread -- the need for current, valid manufacturing information in an accessible form. Indeed, the information explosion of the 1980's has taught us all that information is worthless if it can not be made available in an integrated form.

Typically (and currently at many plant sites), manufacturing information is available as handwritten reports or, worse yet, boxes of computer generated printouts. Indeed, this author has seen abandoned offices four feet deep in paper containing valuable manufacturing information. The approved method of access? Hip waders and a green visor.

PURPOSE OF PAPER.

Hewlett Packard has provided the Image¹ database as a tool on its technical computer series for the organization of plant information. Even with Image, however, several obstacles exist to hinder the ready integration and utilization of this information. Traditionally, the information

¹Image is a registered trademark of Hewlett-Packard

required will be spread over several computer systems throughout the plant, each system being managed by a different plant department. For example, quality standards and testing information is controlled by the Quality department while process standards and process conditions are controlled by the Process department. And so it goes through materials, production, waste, conversion, etc.

The purpose of this paper is to describe a method (using PRESTO, a product available from Eyring Research, Inc.) whereby a disjoint set of Image databases may be integrated into a single relational view. This integration method may be applied to multiple databases residing either on a single system or over a DS or NS network, or a combination of single and networked databases. The vehicle for this description will be a presentation of several case studies. The case studies are as follows:

- * Case #1: Product quality is observed to be cyclic daily. Is there a difference in product quality between lead operators?
- * Case #2: Certain process conditions are suspected of contributing more to quality variations than previously thought. Conduct a study to determine the product quality contribution of certain process variables.
- * Case #3: Determine cost effective ways of shortening total in-process production time.
- * Case #4: PRESTO is interfaced to the HP product QDM/1000².
- * Case #5: A manufacturing database is expanded due to natural plant and product growth.

BRIEF DESCRIPTION OF PRESTO.

PRESTO is the name of a family of software products available from Eyring Research, Inc. Highlights of the product include:

- * Extremely fast data extraction. Bench marks have shown PRESTO to perform 5 to 10 times faster than Query and 2 times faster than QDM/1000. Sophisticated

²QDM/1000 is a registered trademark of Hewlett-Packard

extraction path optimization methods employed by PRESTO have reduced the extraction time for a custom program using traditional Image access from 4.5 minutes to 20 seconds.

- * Integration of Image databases into a single relational view. Integration may include a single database with multiple data sets, multiple databases, or multiple databases over a DS or NS network.
- * Active database access. PRESTO can extract data while the data acquisition system is placing data in the Image database.
- * Friendly ad hoc user interface. A data dictionary provides friendly user interface with databases. On-line help and full on-line access to PRESTO manuals.
- * The PRESTO data dictionary allows pseudo fields, modification of Image field attributes, and user defined field and data set names.
- * Extracted data may be converted for statistical analysis (STAT/1000³, MINITAB, STAT/80), graphic reports (GRAFIT), printed reports (user programs, PRESTO Report), Spreadsheets (LOTUS 123), user programs, external systems, etc.
- * PRESTO supports data archival without shutting down the database.

GENERAL COMMENTS CONCERNING THE CASE STUDIES

PRESTO is installed in a wide variety of environments. End user environments include chemical batch, moving web, meterological, and waste control. One PRESTO OEM is integrating PRESTO into a medical lab system. The case studies, however, will be concerned more with the manufacturing environment. Examples include both the use of private databases and product databases such as QDM/1000.

³STAT 1000 is a trademark of Eyring Research, Inc.
STAT 80 is a trademark of Statware, Inc.
GRAFIT is a trademark of Graphicus, Inc.
MINITAB is a trademark of Joiner Associates
LOTUS 123 is a trademark of Lotus Corporation

CASE STUDY #1

- * **Statement:** Product quality is observed to be cyclic daily. Is there a difference in product quality between lead operators?
- * **Resolution:** The investigating engineer was faced with the task of extracting relevant information from several department databases maintained on separate (though networked) machines. PRESTO obtained the initial study data from the distributed database and made it ready for statistical analysis within 10 minutes.

For the purpose of this discussion, we will consider the example database as depicted in Figure 1. The actual database from which the case study was extracted is much more comprehensive and is being updated with plant information at a rate of between 15,000 and 20,000 records per day.

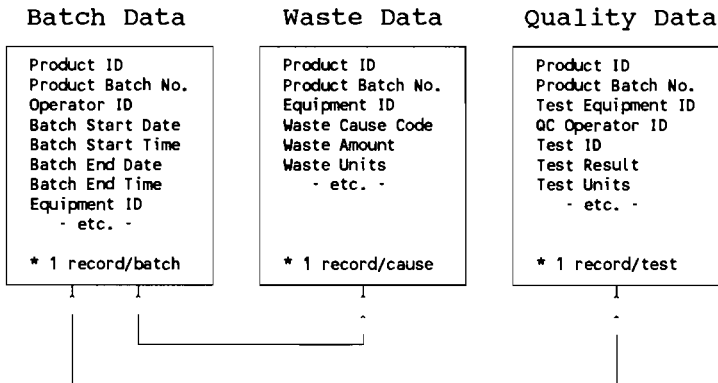


Figure 1 - Example Database for Case Study #1

The example database consists of three relations (data sets): 1) Batch Data (containing information pertaining to a unique instance of production), 2) Waste Data (containing information relating to waste at each step of the process), and 3) Quality Data (containing information on each quality

test performed at each step of the process. Note: in the example database, testing is performed via statistical sampling).

The relationship of each data set is depicted by the arrows between the different data sets. For each Batch there may be from 1 to many occurrences of records in Waste and/or Quality. Note that the common field between each data set is Product ID and Product Batch No.

The several steps utilized by the investigating engineer are very similar to those used in a relational model because of the relational view that PRESTO imposes on the Image database. These steps are as follows:

Step 1: Determine the information to be gathered. In a relational model we would "define the target relation." In this case, the target relation is:

Operator ID
Product ID
Product Batch No.
Waste Cause Code
Waste Amount
Test ID
Test Results

In the case under study, the investigating engineer desires to determine the statistical difference in product quality between lead operators. Only lead operator ID's are carried in the database; thus the inclusion of Operator ID will partition the rest of the relation by that factor.

To identify the quality information, Product ID, Product Batch No., Test ID, and Test Results are also included in the target relation. As an afterthought, the engineer has also included Waste Cause Code and Waste Amount to further determine any correlation between lead operator and waste.

Step 2: Determine the relations containing the target information. Figure 2 shows the location of each field.

Batch Data	Waste Data	Quality Data
Product ID Product Batch No. Operator ID Batch Start Date Batch Start Time Batch End Date Batch End Time Equipment ID - etc. - * 1 record/batch	Product ID Product Batch No. Equipment ID Waste Cause Code Waste Amount Waste Units - etc. - * 1 record/cause	Product ID Product Batch No. Test Equipment ID QC Operator ID Test ID Test Result Test Units - etc. - * 1 record/test

Figure 2 - Target Information

Step 3: Declare relationship links. In the case under study, Product Batch No. is always unique. Thus, the link between each relation is Product Batch No. (see Figure 3). If plant operation was such that Product Batch No. was unique only within product line, the link between each relation would have been both Product ID and Product Batch No.

Step 4: Declare study restrictions. The restriction declarations were as follows:

Batch Start Date	GT 1 Jan 1987
	AND
	LE 1 Apr 1987
	AND
Product ID	EQ MZ4231X
	AND
Test ID	EQ CD004

Note that the linking field (Product Batch No.) was not referenced in the restriction declarations. PRESTO will form a set of valid Product Batch No.s from the smallest relation (in this case, Batch Data) and then link across to the other, larger, relations (Waste Data and Quality Data).

Batch Data

Product ID
Product Batch No.
Operator ID
Batch Start Date
Batch Start Time
Batch End Date
Batch End Time
Equipment ID
- etc. -

* 1 record/batch

Waste Data

Product ID
Product Batch No.
Equipment ID
Waste Cause Code
Waste Amount
Waste Units
- etc. -

* 1 record/cause

Quality Data

Product ID
Product Batch No.
Test Equipment ID
QC Operator ID
Test ID
Test Result
Test Units
- etc. -

* 1 record/test

Figure 3 - Relationship Links

Step 5: Declare fields of study to form the target relation. Figure 4 shows the various fields selected.

Batch Data

Product ID
Product Batch No.
Operator ID
Batch Start Date
Batch Start Time
Batch End Date
Batch End Time
Equipment ID
- etc. -

* 1 record/batch

Waste Data

Product ID
Product Batch No.
Equipment ID
Waste Cause Code
Waste Amount
Waste Units
- etc. -

* 1 record/cause

Quality Data

Product ID
Product Batch No.
Test Equipment ID
QC Operator ID
Test ID
Test Result
Test Units
- etc. -

* 1 record/test

Figure 4 - Fields Selected for Target Relation

Step 6: Perform extraction. PRESTO provides several methods for performing the indicated data extraction. OEMs will find that PRESTO can be quickly integrated into their systems. PRESTO is constructed such that a well-defined interface exists to provide a clear line between OEM software and PRESTO.

End users may extract the information in foreground (PRESTO will display the progress of the extraction as it proceeds), immediate background (PRESTO will provide a complete status of the extraction upon request), or time scheduled background (again a complete status is available). The time scheduled option provides the user with the capability to define extractions and reports that run hourly, daily, weekly, monthly, etc.

Step 7: Convert extracted data. Once the information has been extracted and the target relation created, it may be converted to any of the following formats:

Statistical	Spreadsheet	External System
STAT/1000	LOTUS 123	
MINITAB		Graphics
STAT 80	User Program ⁴	GRAFIT
		ATA Plot ⁵

PRESTO Report

Note that new conversions are easily added to the PRESTO structure.

Case Study #1 Conclusion: Though the information required by the investigating engineer was provided and maintained by several departments within the plant (in the actual case, the data was located on several computers across a network), the final target relation was available for statistical analysis (in this case STAT/1000) in a matter of 10 minutes from a distributed database totaling over 1.2 gigabytes.

⁴PRESTO is an open system; the definition of the target relation file is readily available so that the results of the extraction may be passed to a user program or external system for further processing.

⁵ATA Plot is a trademark of Automated Technology Associates. ATA Plot is listed here to show that preliminary investigations necessary to create a conversion function to support ATA Plot's data formats have been completed.

CASE STUDY #2

- * **Statement:** Certain process conditions are suspected of contributing more to quality variations than previously thought. Conduct a study to determine the product quality contribution of certain process variables.
- * **Resolution:** A full investigation of both historical data and planned investigations was completed within one week, detailing the interactive impact of the suspect process variables on quality.

Figure 5 depicts the database example used for this case study. Though the example databases are being kept simple for the purpose of this discussion, the actual databases referenced by the case studies address entire plant environments.

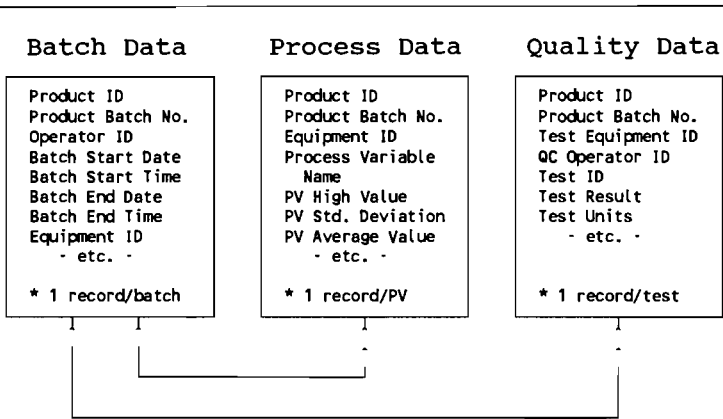


Figure 5 - Example Database for Case Study #2

The example database consists of three data sets (relations): 1) Batch Data (containing information pertaining to a unique instance of production), 2) Process Data (containing data acquired from the monitoring of process conditions), and 3) Quality Data (containing information on each quality test performed at each step of the process. Note:

in the example database, testing is performed via statistical sampling).

The relationship of each data set is depicted by the arrows between the different data sets. For each Batch, there may be from 1 to many occurrences of records in Process and/or Quality. Note that the common field between each data set is Product ID and Product Batch No.

The steps used to realize the solution are as follows:

Step 1: Determine the information to be gathered. The target relation defined by the investigating engineer was:

Product ID
Product Batch No.
Operator ID
Equipment ID
Process Variable Name
PV High Value
PV Low Value
PV Standard Deviation
PV Average Value
Test Equipment ID
QC Operator ID
Test ID
Test Result
Test Units

The study requires that the contribution of equipment and operator variance be properly handled before the actual impact of the suspect process variables can be determined. Hence, the investigating engineer created a target relation that preserved the necessary relationships.

Step 2: Determine the relations containing the target information. Figure 6 shows the location of each field.

Step 3: Declare relationship links. In contrast to Case Study #1, Product Batch No. is not unique between product classification. Therefore, Product ID is required as the link between each relation.

Batch Data	Process Data	Quality Data
Product ID Product Batch No. Operator ID Batch Start Date Batch Start Time Batch End Date Batch End Time Equipment ID - etc. - * 1 record/batch	Product ID Product Batch No. Equipment ID Process Variable Name PV High Value PV Std. Deviation PV Average Value - etc. - * 1 record/PV	Product ID Product Batch No. Test Equipment ID QC Operator ID Test ID Test Result Test Units - etc. - * 1 record/test

Figure 6 - Target Information for Case Study #2

Step 4: Declare study restrictions.

```

Product ID           EQ  XYZ123
  AND
Batch Start Date    GT  1 Apr 1986
                    AND
                    LT  1 Feb 1987
  AND
Process Variable Name EQ  Upper Oven Temp
                    OR
                    EQ  Lower Oven Temp
                    OR
                    EQ  Upper Oven Humidity
                    OR
                    EQ  Lower Oven Humidity

REJECT Process Data
  
```

The REJECT clause of the restriction is necessary to preclude data from the process data relation from being included if there is no matching entry in the quality data relation. Since quality measurements follow a statistical sampling program, there will be many entries in process data which have no matching entries in quality data.

Step 5: Declare fields of study for the target relation. Figure 7 shows the various fields selected.

Batch Data	Process Data	Quality Data
Product ID Product Batch No. Operator ID Batch Start Date Batch Start Time Batch End Date Batch End Time Equipment ID - etc. - * 1 record/batch	Product ID Product Batch No. Equipment ID Process Variable Name PV High Value PV Std. Deviation PV Average Value - etc. - * 1 record/PV	Product ID Product Batch No. Test Equipment ID QC Operator ID Test ID Test Result Test Units - etc. - * 1 record/test

Figure 7 - Target Information for Case Study #2

Step 6: Perform extraction. The various methods available to the user for this step are explained in Step 6 of Case Study #1.

Step 7: Convert extracted data. The various methods available to the use for this step are explained in Step 7 of Case Study #1.

Case Study #2 Conclusion: The investigating engineer used several statistical tools on the extracted data (via the STAT/1000 conversion) to determine the contribution of equipment and operator variation. The actual contribution was then determined and a full technical presentation prepared using the statistical, graphic, and tabular reporting available through conversion capabilities of PRESTO.

CASE STUDY #3.

- * **Statement:** Determine cost effective means of shortening total in-process production time.
- * **Resolution:** This investigation was conducted over the span of several months. The steps followed were generally those described in case studies #1 and #2. The end result was several well-documented changes to production methods resulting in decreased production time and reduced in-process inventory.

An example database is depicted in Figure 8. This case study will not detail each step of the investigation because of the many iterations of the solution method. The strength of this case study is the capability provided by PRESTO to answer ad hoc questions. During the investigation process, many additional questions will arise from the answers to other questions. Creative problem solving is greatly enhanced by the rapid answering of each creative question.

Batch Data	Process Data	Quality Data	Alarm Data
Product ID Product Batch No. Operator ID Batch Start Date Batch Start Time Batch End Date Batch End Time Equipment ID - etc. - * 1 record/batch	Product ID Product Batch No. Equipment ID Process Variable Name PV High Value PV Std. Deviation PV Average Value - etc. - * 1 record/PV	Product ID Product Batch No. Test Equipment ID QC Operator ID Test ID Test Result Test Units - etc. - * 1 record/test	Product ID Product Batch No. Equipment ID Process Variable Name PV Extreme Value Alarm Date Alarm Time - etc. - * 1 record/batch

Figure 8 - Example Database for Case Study #3

The investigating engineer proceeded with the study by gathering information, making small changes, and evaluating the changes by noting variations in the way data was reported to the manufacturing database. Of particular interest was the Alarm Data. The engineer used this relation as an indicator of the effect of the changes. If more alarms were noted in the particular area changed, the change was reversed and a corresponding downward variation in Alarms was watched for.

*** Case Study #3 Conclusion:** An iterative investigation was conducted by evaluating small, meaningful changes which accumulated to produce the result sought.



CASE STUDY #4

- * **Statement:** PRESTO is interfaced to the HP product QDM/1000.
- * **Resolution:** Presto may be easily interfaced to the HP product QDM/1000 with a resulting two times reduction in the time to produce reports and graphs.

The Data Dictionary inherent in PRESTO builds the first draft of the dictionary for a new database by interrogating the Image root file and defining the default attributes of each database field. The Dictionary facility then allows the user to alter the structures to better define the database as a set of relations. This redefinition provides the following functions:

- * Marking a relation for denied access
- * Marking a field for denied access
- * Creating pseudo fields by concatenating fields or splitting fields
- * Supplying meaningful names to each data item and relation

The particular case example required one hour to define a user friendly data dictionary view of the QDM/1000 Image database. This definition was then transported to a manufacturing site using QDM/1000. Within 15 minutes of the loading of PRESTO on the host system, reports and histogram graphs were being produced which duplicated the reports and graphs delivered by QDM/1000 but in half the time.

Case Study #4 Conclusion: PRESTO was used on a commercially available Image database to produce reports and graphs in half the time required by the application software delivered with the database.

CASE STUDY #5

- * **Statement:** A manufacturing database is expanded to a distributed database due to natural plant and product growth.
- * **Resolution:** Users experienced two hours down time; then business resumed as usual.

The natural increase of market demand for product and the need for plant growth resulted in an increase in the data

being gathered by the plant data system. This situation caused the database to become much larger and the incoming data traffic to saturate the data communication lines and the ability for Image to add the data to the manufacturing database.

Figure 9 describes the system "before" the change to remedy the problem. The solution chosen was to distribute the manufacturing database across two machines and to use a network (DS or NS) to link the two machines. PRESTO's ability to operate across a network was a major consideration in the decision.

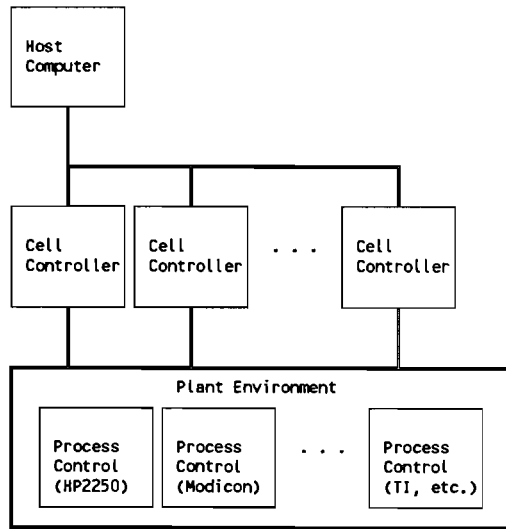


Figure 9 - Example System "Before Image", Case Study #5

Figure 10 shows that the host computer was replicated. The database was split between the two machines to provide additional bandwidth for both Image-to-disk and the communication lines. The network link between the two host systems provided for communication when PRESTO was extracting data from both databases.

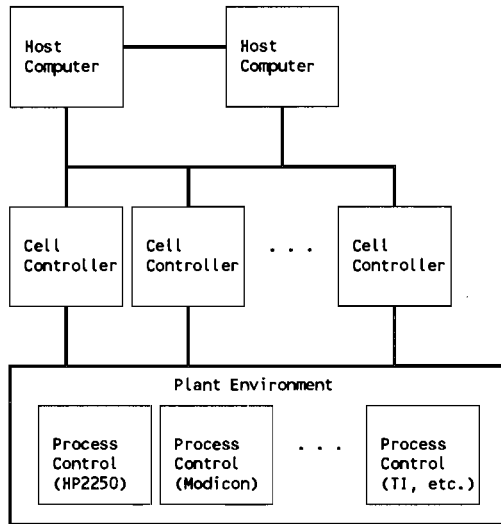


Figure 10 - Example System "After Image", Case Study #5

The planning and hardware installation required months of work. Careful installation of additional communication lines, new hardware, and testing was accomplished before the actual conversion to the new system. The actual accomplishments of the last few days of the conversion were as follows:

- * The PRESTO data dictionary was copied to the new host computer and the dictionary was modified on both hosts to indicate the new geography of the database. Time to complete: 10 minutes.
- * The day before the actual conversion a full archive of the data to be moved to the new system was made on-line without interrupting plant operation.
- * The day of conversion required two hours of down time to stabilize the database (an archive of the last

few hours of data was made at this time) and to move communication lines and move some terminals to the new host. Communication software was then reconfigured to cause data from certain areas of the plant to be communicated to the new host rather than the old host.

- * The system was then restarted and data began flowing into the new distributed database. A PRESTO "restore function" of the archived data was started on the new host. Because of the PRESTO data dictionary, the users saw no difference in operation. PRESTO got the data from the appropriate databases without user intervention. The restore required approximately two days. At the end of the restore, the data moved to the new host was purged from the old host database.

Case Study #5 Conclusion: A major reconfiguration of the plant data system and manufacturing database was accomplished with only two hours of planned down time. The net effect on the users was an increase in processing power with no retraining.

CONCLUSION.

The proper application of manufacturing data has become one of the most powerful tools available to modern industries. As costs have increased, it has become increasingly important to reduce inventories, waste, and in-process time. Effective reduction programs require carefully researched and executed plans to minimize impact upon plant operations.

The method of Image database integration discussed in this paper is an off-the-shelf solution. It provides a relational view of diverse Image databases on single systems or networked systems. The speed and flexibility of data extraction provides investigating engineers with a tool to conduct investigations in a fraction of the time traditionally required.

PRESTO has been successfully used by major manufacturing companies to provide access to the information necessary to produce and execute these plans. PRESTO has proven cost effective because of the flexible and comprehensive access tools provided to users for the interrogation of manufacturing databases. PRESTO also assists in the management of those databases.



Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

A. Reeves, N. Stass, and R. Wood
Telesat Canada
333 River Road
Ottawa Ontario
Canada K1L 8B9

Abstract

Sharing expensive peripheral equipment among computers is one method of optimizing the use of the peripherals. At Telesat, we have previously used this scheme to centralize the printing requirements of several HP1000 computers and an IBM mainframe on an HP2680A laser printer. We have now expanded our system to support a centralized graphics plotting and slide generation facility, using the HP7550A plotter (with automatic paper feed) and the HP7510A color film recorder.

This paper discusses the hardware and software details of our implementation, the fitting of an third party graphics package to the system (Graphic User Systems Inc. GRAFIT/1000), and some of the pitfalls we encountered. Also, our future plans for the graphics facility will be outlined.

Introduction

Telesat is the national domestic satellite communications company of Canada. Headquartered in Ottawa, Telesat also provides international satellite communications engineering consulting, and sells satellite flight dynamics software and orbital tracking services. Technical computing resources at Telesat consist of many varied Hewlett-Packard computers: HP1000 E-series, F-series, A600s, and A900s; HP150 PCs; HP Vectra PCs; and a recently acquired HP840 Spectrum system. As well, the business computing groups have an IBM 4381 mainframe and a variety of IBM-compatible PCs.

Over the past four years the need at Telesat for high-quality paper drawings and slides has increased dramatically. Part of these needs stem from purely internal requirements, but many of the drawings and slides are developed for customer documentation, business proposals, and business presentations. The production of these drawings and slides was becoming an expensive and time-consuming procedure; it was becoming obvious that a better system would have to be devised. When Hewlett-Packard announced the release of the HP7510A film recorder Telesat was immediately interested. However, the cost of the HP7510A was high enough to discourage the company from buying large quantities

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

of the machines. To get the benefits of a single HP7510A, it was decided to integrate the film recorder into the centralized spooling system.

The Telesat centralized spooling system concept has been discussed in a previous paper presented at the 1984 INTEREX conference by R. J. Meldrum. (See "Centralized Printing Using An HP2680 Laser Printer", Proceedings of the 1984 INTEREX Conference.) The major points of the system will be briefly restated here.

The centralized spooling system was created by Telesat so that the printing requirements of the HP1000 systems could be handled by one high speed HP2680A laser printer. Originally designed to handle printing for five 2117F computers, the system now accommodates six 2117Fs, two A900 computers, a Spectrum 840 computer, and (by tape transfer) certain print jobs from the IBM mainframe. (See Figure 1.) The controller for the centralized spooling system is an A900 computer, dubbed the Peripheral Management Computer (or PMC), which uses an HP2680A printer driver developed by HP as a special product. The HP1000 computers are connected to the PMC by high speed data links developed by Telesat, using parallel interface cards. DS/1000 was not used, since it was deemed to be too heavy a system resource user. Since the Spectrum is a very recently acquired product, it is presently connected to the PMC via serial MUX cards at both ends. The Spectrum uses a "dumb" printer model for the communications link, and places markers in its output to specify certain control operations to the PMC.

In order to facilitate the centralized spooling concept, the RTE-6 spooling system was both modified and enhanced to provide extra features and functions. The RTE-A spooling system was simply enhanced with a set of "overlays" that assist in programmatic opening and closing of spool files. (See Appendix A.) The fact that we have the source code for RTE-6 and RTE-A made these enhancements much easier. At this time, the Spectrum's UNIX system is being used in an "as-delivered" state, but with the dumb communication link mentioned above.

The PMC has its own special spooling system that is completely independent of the standard RTE-A spooling system. (The present PMC spooling system is displayed in Figure 2.) It actually is more reminiscent of the RTE-6 spooling system, in that there is a central spool controller (TSMP, similar to SMP), a control file (TPLCON, similar to SPLCON), and various output controllers (TPRIN, T2687, I7510, and I7550, similar to SPOUT.) TPRIN handles output to the HP2680A laser printer while a recent addition, T2687, handles output to a remote Laserjet printer in another building. As well, there is an interactive program that is used by the operators for spool routing and control, as well as for spooling system initialization (TASK, similar to GASP). I7510, I7550, and PLOT will be described later in the paper.

The central spooling system has now been operative for nearly four

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

years, and although it was more complex to develop than originally anticipated, it has been a very reliable and productive system. (In fact, the usage of the HP2680A printer has greatly exceeded expectations.) Therefore, it seemed a natural step to enhance the central spooling system by adding an HP7510A film recorder and an HP7550A plotter. (Since many people do not have access to a plotter, the addition of the high speed HP7550A plotter to the central plotting system would be a boon to our users.) By adding the HP7510A and the HP7550A to the PMC most of Telesat's HP technical computers would have access to the devices, and yet costs could be kept down by requiring only a few (instead of many) graphics devices. This is exactly what we wanted; flexibility at a reasonable cost.

Implementation

Before the installation of the plotter and film recorder could be done, it was necessary to analyze present graphics requirements on the HP1000s, and then decide how to make the graphics tools work with the central plotting system. The major graphics tool used on the Telesat HP1000s is an RTE-6 program called DRAW, whose function (as suggested by the name) is to generate drawings on HP graphics terminals, and then to output the drawings to plotters attached directly to the graphics terminals. DRAW was developed in-house by Ron Costanzo, Manager of Systems Software. Loosely patterned on BRUNO (a drawing program in the Contributed Software Library), DRAW is based upon HP's DGL package. It therefore seemed that the most logical approach to integrating DRAW and other standard DGL-based graphics tools would be to develop a DGL link to the central spooling system.

After some consideration, it was decided that a set of DGL "stubs" would be developed on RTE-6. These stubs would have the same names as the real DGL routines, but would connect into the central spooling system rather than perform the usual DGL functions. As well, a set of extra "DGL look-alike" routines would be developed to provide functions for color control on the HP7510A film recorder, paper size choice on the HP7550A plotter, spool control, and so on. (See Appendix B.) The expectation was that this method would keep spool files to a reasonable size, and allow for greater flexibility in processing the spool file at the PMC. In addition, programs could use the central plotting system with minimal changes. For reasons that will become clearer later in this paper, this scheme has been dubbed "emulation mode".

After considerable development work, the implementation of this scheme was installed in late 1986 on the RTE-6 systems. Rather than detail the actual development of the scheme, a simplified outline will be given of what happens when an RTE-6 program is loaded with the DGL stubs, the production of a graphics spool file, and the interpretation of the graphics spool by the PMC. (Refer to Figures 3 and 2.)

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

- 1) All the DGL stubs (including the DGL "look-alikes") are contained in a library called \$\$S7550.LIB::LIBRARIES. In order for a program or program segment to use the central plotting system, a user merely need search \$\$S7550 instead of the standard DGL library D0047 when LINKING the program.
- 2) When the program is run, the stub ZBEGN (initialize DGL system) is called to create a spool output file. ZBEGN in turn calls entry ZRECI. (ZRECI is one entry into a subroutine called ZREC. Other entry points are ZRECC and ZRECT, which will be discussed below.) The following actions now occur:
 - i) ZRECI calls a standard Telesat routine, TSPLO, to open a permanent non-list spool file. TSPLO returns the name of the spool file. The main point of this step is to create a uniquely named spool file on the desired spooling area. The spool is then closed and re-opened as a normal FMP file via a FmpOpen request. Strictly speaking, at this point the output file is no longer a spool file. However, for consistency this paper will continue to refer to it as a spool file.
 - ii) A dummy record 1 of 252 bytes is written into the file.
 - iii) The string "BEGN" is entered in a data buffer internal to the ZREC subroutine, indicating that the ZBEGN call has been made. As well, a DGL initialization flag is set locally in ZREC.
 - iv) A return is made back through ZRECI and ZBEGN to the user's program.
- 3) When the stub ZDINT (initialize graphics device) call is made, it in turn calls entry ZRECC. The following actions occur:
 - i) The DGL initialization flag is checked to ensure that the ZBEGN call has been made.
 - ii) A device initialization flag is set locally in ZREC to indicate that a ZDINT call has been made.
 - iii) The string "DINT" is entered in the internal data buffer along with the value of ZDINT's passed parameters.
 - iv) Device control information concerning the destination device (film recorder or plotter) and device configuration is initialized internally in ZREC. (The destination device is specified by replacing the device LU in the ZDINT call with ASCII FI (film) or PL (plotter). This is the only case where a stub DGL routine has a slightly different calling sequence from the standard DGL routine.)
- 4) All further stub DGL calls except for ZDEND or ZEND in turn call entry ZRECC to enter the DGL call in the internal data buffer,

**Implementing a Centralized Plotting and Slide Generation
Facility for Hewlett-Packard Computers**

provided the DGL initialization flag and the device initialization flag are set. The entry consists of recording the name of the DGL call in the file minus the leading "Z", followed by any parameters. For example, calling the stub DGL routine ZWIND with parameters 0., 240., 0., and 180. would result in the following entry in the data buffer:

<u>In Buffer</u>	<u>Meaning</u>
53511B	WI
47104B	ND
OB	0.0
OB	
74000B	240.0
20B	
OB	0.0
OB	
55000B	180.0
20B	

When the data buffer reaches a maximum of 1280 bytes, the buffer is written to the spool file as one record, and the buffer is cleared. The choice of 1280 bytes was made in order to accommodate the largest possible stub DGL request, which in this implementation is a polygon set with a maximum of 125 vertices.

Certain DGL look-alike calls are not stored in the data buffer, but affect the device control information which is kept internally in ZREC.

- 5) When the stub ZDEND call is made, the call is stored in the data buffer and the device initialization flag is cleared.
- 6) When the stub ZEND is called, it in turn calls entry ZRECT. The following actions now occur, providing the DGL initialization flag is set:
 - i) The DGL initialization flag is cleared.
 - ii) The call is stored in the internal data buffer.
 - iii) Any remaining information in the data buffer is written out to the spool file.
 - iv) The file is closed with an FmpClose call.
 - v) The file is now re-opened as a Type 1 file in update mode. The device control information in ZREC is written into record 1 of the spool file, and the file is again closed.
 - vi) Routine TSPLO is called to ready the spool file for transmission to the PMC. At this point, the spool file status is changed from "save" status to "purge" status, meaning that the spool system will automatically purge the spool file after transmission to the PMC.

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

- vii) Telesat routine EVC is called to specify the output LU on the PMC: LU 10 for the HP7510A film recorder, or LU 50 for the HP7550A plotter.
 - viii) Routine TSPLC is called to queue the spool file for transmission to the PMC. The spooling system now transmits the spool file to the PMC.
 - ix) A return is made to the user's program.
- 7) When the spool file is received at the PMC, program TSMP recognizes that the spool file is destined for the film recorder or plotter. The file is entered in the spool queue, but is put on "hold" status. The computer room operators can now use a utility program, PLOT, to list the spools waiting to go to the film recorder and plotter, and the output requirements of each file. (For example, a spool file for the film recorder may require Polaroid film instead of 35mm film.)
- 8) The operators will make any changes necessary to the film pack of the HP7510A film recorder, or the pen carousel and paper pack of the HP7550A plotter. When the appropriate device is ready to receive the spool file, the operators release the spool file.
- 9) When the spool file comes to the top of the list of released spools waiting to plot, program I7550 (for the HP7550A plotter) or program I7510 (for the HP7510A film recorder) interprets the contents of the spool. The following actions occur:
- i) Any necessary control information in record 1 of the spool is sent to the device.
 - ii) Starting at record 2, the name of each DGL routine is picked up. The routine is identified, and the appropriate number of parameters are retrieved from the spool. The real DGL routine is then called by I7550 or I7510. (These two programs are loaded with an indexed version of HP's D0047 DGL library.)
 - iii) When the spool contents have been completely consumed, a signal is sent to the spooling system indicating that the process has completed. The spooling system removes the spool from the active list.

A word should be added here concerning the selection of colors on the film recorder and plotter. It is possible for user programs to specify each and every color or pen to be used on the devices. However, this is not usually done. For the film recorder, eight standard palettes (numbered 1 through 8) have been pre-defined in the system. Each palette has eight color/linewidth combinations, and a background color. Under most circumstances, user programs merely specify which pre-defined palette to use. For the plotter, a standard pen pack has been defined. This pen pack has the same color/linewidth combinations as palette 7 for

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

the film recorder. Under most circumstances, user programs merely default to the standard pen pack.

The first program to be converted to use the central plotting system was DRAW. Ron Costanzo performed this reasonably straight-forward modification. Since DRAW is a segmented program, Mr. Costanzo simply extended the output choice menu to include a central plotting feature (including plot or film choice), and then added another program segment that (at LINK time) is loaded with the stub DGL library. DRAW was used to help debug and fine-tune the new central plotting system.

Fine-tuning the system took a little longer than expected, partly because of the unfamiliar nature of the HP7510A film recorder (since Telesat had one of the first HP7510As in Canada, even the local HP service engineers were unfamiliar with the device), and partly because of unexpected glitches. The most annoying glitch that had to be dealt with was interference between the HP7510A and the HP7550A, which were on the same HPIB interface. Apparently, the HP7510A at times virtually takes over the HPIB bus. If the HP7550A is also running, then the plotter slows down considerably or even stops. Several times a situation developed where the plotter stopped with its pen down, which left blotches on the drawing paper. At first, this problem was handled by instructing the computer room operators to release spool files to either the HP7510A or the HP7550A, but not both at the same time. The problem has now been fixed permanently by putting the two devices on separate HPIB interfaces. (We did try using a MUX card interface, but had problems with this setup. It was later discovered that HP's DGL does not support any device connected to a MUX card.)

Since DRAW, other internally-developed graphics programs have been adapted to use the central plotting system, with successful results.

Adaptation of Third Party Graphics Packages

So far, only one third party package has been adapted to our central plotting system, Graphic User Systems' (or Graphicus) GRAFIT/1000 graphing system. GRAFIT has been used at Telesat for some time now, and has proved to be a very useful program that has found many applications. Though the central plotting system was not developed with third party packages in mind, we decided to try to add GRAFIT to central plotting.

For those unfamiliar with the GRAFIT package, it consists of several cooperating programs. From our experience with the package, we have deduced the following: the core program, GRAFIT, does most of the work. However, actual graphics output is done by a set of graphics output handler programs. When the user requests that a graph be drawn on his terminal or a specified output device, GRAFIT uses tables (set up by the user) to determine which output program to use. GRAFIT then

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

schedules the output handler, and passes it two class numbers. One class is used for input into the handler; the other class is used by GRAFIT to receive replies from the handler. GRAFIT proceeds to pass data to the output handler through the handler's input class. When GRAFIT has sent all the necessary data to the output handler, GRAFIT finishes by sending a termination request to the handler. The handler then terminates, and the two classes are released.

At first, it was thought that it would be easy to adapt GRAFIT to use the central plotting system: all that would be necessary would be to load a special output handler using the stub DGL library instead of the standard DGL library. However, we quickly discovered that this didn't work. After some investigation, and discussion with Graphicus, it turned out that Graphicus had modified some of the HP DGL routines to perform functions that standard DGL will not support. Although various combinations of Graphicus routines and our routines were tried, GRAFIT would not always produce a graph successfully at the PMC.

Rather than continuing in this fruitless fashion, it was decided to modify the central plotting system. The "emulation mode" scheme was retained for in-house graphics packages, but we added a system that captures the output of standard DGL packages in a spool file, and then sends that spool file down to the PMC for interpretation. This scheme has been dubbed "verbatim mode". Verbatim mode was implemented by constructing a special version of the ZREC subroutine, and has the following features:

- 1) Only one initialization flag is used. This flag shows only that ZRECI has been called.
- 2) The spool file opened by ZREC is always linked to session LU 6, unlike emulation mode where any available session LU is used. Therefore, the HP ZDINT call in the device program must specify LU 6 when performing device initialization.
- 3) The choice of paper or film is specified by the call to ZRECI. (In emulation mode, the choice of paper or film is specified in the call to ZDINT.)
- 4) A flag is set in the spool file record 1 to indicate that the spool is a verbatim spool, not an emulation spool.

Verbatim mode does have four drawbacks over emulation mode: first, spool files are much bulkier. Second, certain dynamic control operations cannot be done in verbatim mode. For example, film recorder pen colors cannot be changed dynamically. Third, if errors occur an emulation spool file can be listed and visually checked for content. This is just about impossible with a verbatim spool file. Finally, certain ZOESC (send hardware dependent escape sequences to device) calls can have unpredictable results. For instance, on the HP7550A plotter a ZOESC call can be made to replot a graph a given number of times. However, if the internal buffer of the HP7550A overflows, then this replot request will not work properly.

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

The addition of GRAFIT to the central plotting system was completed using verbatim mode, after some trial and error. The GRAFIT central plotting system has the following major features in its layout (see Figure 4):

- 1) GRAFIT has been set up with special transfer files (CENTRAL1, CENTRAL2, CENTRAL4, CENTRAL5, CENTRAL8) to send a graph to the central plotting system. The user executes one of these transfer files, in GRAFIT, to start the central plotting procedure. (The number following each transfer file refers to the pre-defined film recorder palette being used. CENTRAL8, for example, uses palette 8. For reasons of color contrast, palettes 3, 6, and 7 are not used with GRAFIT.)
- 2) The transfer file specifies an output device configuration table, which in turn specifies one of five dummy output handlers (GRSPL1, GRSPL2, GRSPL4, GRSPL5, GRSPL8). Each handler is essentially the same; the only difference is that each handler specifies its own unique set of pen colors and widths. For example, GRSPL8 specifies the following pen colors for the HP7510A and the HP7550A:

<u>Pen #</u>	<u>Pen Color</u>	<u>HP7510A Line Width</u>	<u>HP7550A Pen Width</u>
1	Basic	16	.3 mm
2	Red	16	.3 mm
3	Blue	16	.3 mm
4	Green	16	.3 mm
5	Yellow	16	.3 mm
6	Violet	16	.3 mm
7	Basic	32	.7 mm
8	Blue	32	.7 mm

(Color "basic" is white for the film recorder, black for the plotter.)

In order to co-ordinate GRAFIT with the device hardware, the device configuration table for the specified GRSPL program has the same pen-color-linewidth definitions as the GRSPL program. This method is used because it is one way of coordinating the colors that GRAFIT thinks are available to the colors the central plotting system will actually use.

- 3) The appropriate GRSPL module is scheduled by GRAFIT. GRSPL retrieves the two class numbers passed by GRAFIT, and prompts the user for information concerning the output media (35mm film, Polaroid film, paper, or transparency). GRSPL then calls ZRECI, which in turn opens a spool file connected to session LU 6. As well, GRSPL schedules and sends a lock request to GNUMB. This program is simply a resource controller, and prevents spool control synchronization difficulties when the user is attempting many sequential accesses to the central plotting system.

Implementing a Centralized Plotting and Slide Generation Facility for Hewlett-Packard Computers

- 4) GRSPL then schedules with wait the standard Graphicus HP7550A device handler, G7550, passing the two class numbers to G7550 in the schedule string. Since the GRSPL device configuration table has been set up to specify LU 6 as the device LU, G7550 starts writing HP-GL material (HP's low level graphics language) to session LU 6, which is presently directed to a spool file.
- 5) Upon completion of the graph, G7550 terminates and GRAFIT continues. GRSPL then calls ZRECT to close the spool file, and sends an unlock request to GNUMB. GRSPL now terminates.
- 6) The spool file arrives at the PMC. When it is handled by I7510/I7550, these programs will identify the spool as being a verbatim file. Control information in record 1 is used to condition the appropriate destination device; all data from record 2 on is simply read and sent straight to the device.

Using verbatim mode, GRAFIT now works correctly with the central plotting system, and is capable of producing some very impressive slides. Depending on one's choice of palette, up to eight colors can be used in the graph.

FUTURE PLANS

Although the central plotting system has performed very well since installation, there is still some work to do on it. Despite the fact the central plotting feature is only six months old, the HP7550A plotter is already overworked. To rectify this situation, another HP7550A will soon be added to the PMC.

As well, the central plotting system was first developed for our RTE-6 systems. These systems are being phased out, though, and so some of the Telesat graphics packages, as well as GRAFIT/1000, are being transported to RTE-A. At the time of the writing of this paper (June 1987) the RTE-A spooling system is being adapted to handle the central plotting function. Because of the similarities between RTE-6 and RTE-A, this task has been fairly straightforward.

However, our ultimate goal is to start shifting more graphics utilities and engineering processing on to our new Spectrum 840 machine. (The Spectrum was purchased for the specific purpose of replacing our older RTE-6 systems.) Since RTE and UNIX are completely different, it is expected that adapting UNIX to support central plotting will be a difficult task, but a task we are optimistic can be accomplished.

CONCLUSIONS

Our experience with central plotting has been positive. The installation was successful, and central plotting is popular with the HP1000 users. As expected, developing unique systems like this with

Implementing a Centralized Plotting and Slide Generation
Facility for Hewlett-Packard Computers

newly released equipment brought the usual development headaches: unforeseen HPIB bus conflicts, DGL bugs, ambiguous manuals, and undocumented equipment "features" to name a few. Despite these problems we feel that the time invested to develop the software to support central plotting has been worthwhile in terms of the return.

Implementing a Centralized Plotting and Slide Generation
Facility for Hewlett-Packard Computers

Appendix A

Spooling additions and enhancements to RTE-6

Subroutines:

TSPLO Friendly library subroutine to open a spool file.
(Schedules #PROP to do the work.)

TSPLC Friendly library subroutine to close a spool file.
(Schedules #PROP to do the work.)

EVC Library subroutine to change attributes of spool files
destined for the Peripheral Management Computer (PMC).
EVC can specify multiple print copies, the PMC destination
LU, the laser printer environment, and other items.
(Schedules EV to do the work.)

PLBCI Print a banner title page.

SPFNM Return the name of a spool file given the LU.

SPLU Return the LU of a spool file given the file name.

Programs:

#PROP Program to provide a friendly spooling interface between
Telesat programs and SMP. #PROP also adds leading and
trailing banners to spool files when requested.

EV Receives requests from subroutine EVC.

SMP HP's SMP program, but modified to support the central
spooling system.

PROUT Replacement for HP's SPOUT program; controls the
transmission of spool files to the PMC.

FJOB Fake job module. JOB has been removed on most of
Telesat's RTE-6 systems; the purpose of this fake is to
keep GASP and SMP happy.

OPN Interactive program to open spool file to session LU 6.

TASK Interactive program to find the status of spool files
residing on the Peripheral Management Computer (PMC).

CFS Interactive program to close a spool file in any session.

Spooling additions to RTE-A

Subroutines:

TSPLO Friendly library subroutine to open a spool file.
(Schedules \$PROP to do the work.)

TSPLC Friendly library subroutine to close a spool file.
(Schedules \$PROP to do the work.)

EVC Library subroutine to change attributes of spool files
destined for the Peripheral Management Computer (PMC).
EVC can specify multiple print copies, the PMC destination
LU, the laser printer environment, and other items.
(Schedules EV to do the work.)

SPLNM Return the name of a spool file given the LU.

Implementing a Centralized Plotting and Slide Generation
Facility for Hewlett-Packard Computers

Spooling Additions to RTE-A (continued)

Programs:

\$PROP Program to provide a friendly spooling interface between Telesat programs and SMP. \$PROP also adds leading and trailing banners to spool files when requested.

EV Receives requests from subroutine EVC.

OUTPT Replacement for HP's OUTPT program; controls the transmission of spool files to the PMC.

OPN Interactive program to open spool file to LU 6.

TASK Interactive program to find the status of spool files residing on the Peripheral Management Computer (PMC).

KILSP Interactive program to kill any spool file.

Appendix B

<u>DGL stubs</u>	<u>Action at the PMC</u>
ZASPK	Redefine the aspect ratio of the virtual co-ordinate system.
ZBEGN	Initialize the DGL system.
ZCOLR	Set the color attribute for line primitives except polygon interior fill.
ZGSIZ	Set the character size attribute for hardware text.
ZDEND	Disable the enabled graphics display device.
ZDINT	Enable a graphics display device.
ZDLIM	Define the logical display limits of the graphics device.
ZDPST	Define the polygon style of an entry in the polygon style table.
ZDRAW	Draw a line from the starting position to the world coordinate specified.
ZEND	Terminate the DGL system.
ZLSTL	Set the linestyle attribute.
ZMARK	Display a marker symbol at the current position.
ZMCUR	Make the picture current.
ZMOVE	Set the starting position to the world coordinate position specified.
ZNEWF	Perform a new-frame-action on the graphics display.
ZOESC	Perform a device-dependent escape function on the graphics display device.
ZPGDD	Display a polygon set in a device-dependent manner.
ZPGDI	Display a polygon set in a device-independent manner.
ZPICL	Set the color attribute for polygon interior fill.
ZPILS	Set the linestyle attribute for polygon interior fill.
ZPOLY	Draw a connected line sequence starting at the specified point.
ZPSTL	Set the polygon style for polygon sets.
ZTEXT	Output graphics text to the graphics device.
ZVIEW	Set the boundaries of the viewport in the virtual coordinate system.
ZWIND	Define the boundaries of the window.

DGL

Look-alikes Action at the PMC

ZDEFC	Define a color to be used by the HP7510A film recorder in terms of the red-green-blue components and the pen width.
ZDEFB	Define a palette of eight colors to be used by the HP7510A film recorder in terms of the red-blue-green components and pen widths, plus the background color.
ZESCT	Change the size of the buffers in the HP7550A plotter.

Implementing a Centralized Plotting and Slide Generation
Facility for Hewlett-Packard Computers

ZNAME	Store the user's name or identification tag in the spool file for output return.
ZPALT	Select a color palette for the HP7510A film recorder from one of the eight standard palettes.
ZPAPR	Set the paper or film type and size.
ZPENS	Select the eight pen types, widths, and colors for the HP7550A plotter.

Underlying

Routines Action

ZRECC	Store DGL stub calls in the spool file.
ZRECD	Return the device code from the spool file.
ZRECI	Create a spool file for output to the PMC.
ZRECP	Return the paper/film type and size from the spool file.
ZRECT	Flush the spool file buffer, write out record 1, and close the spool file.

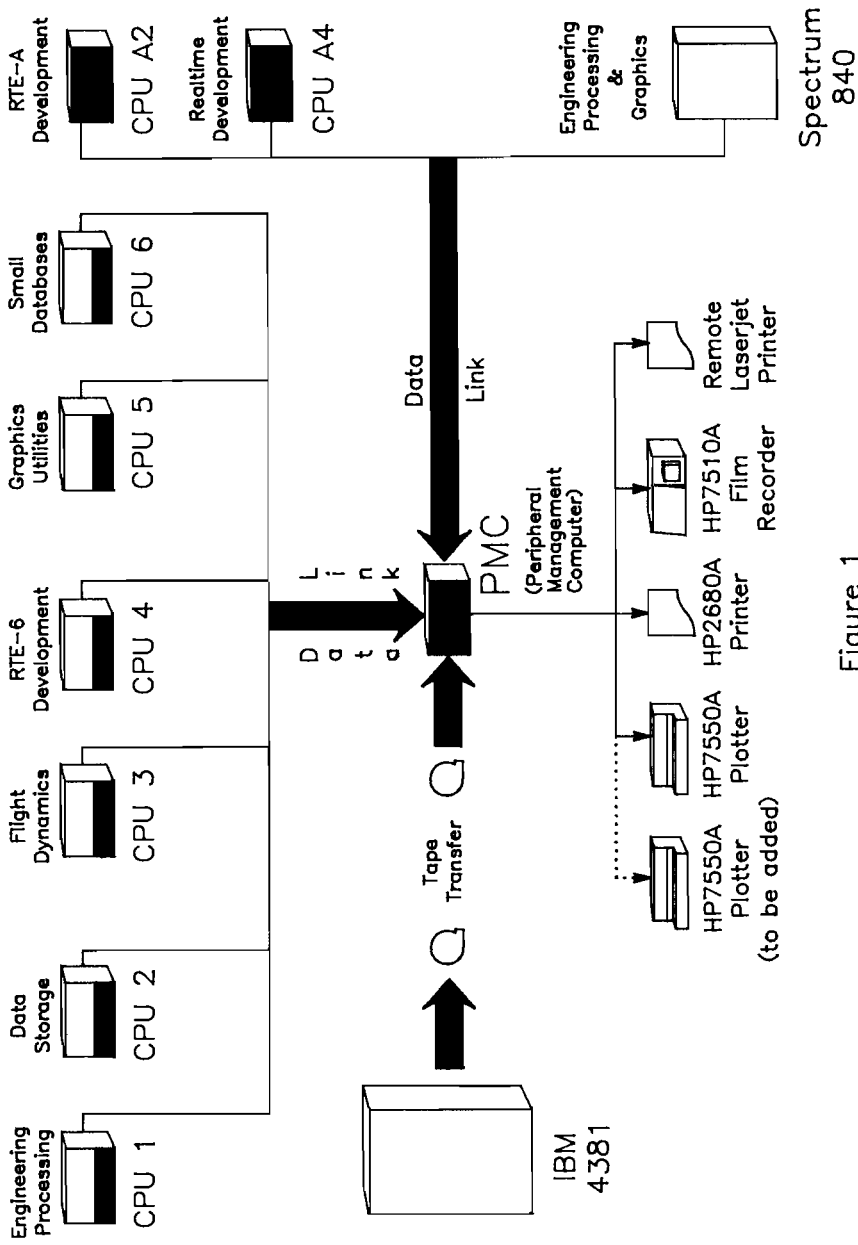


Figure 1
The Central Spooling System

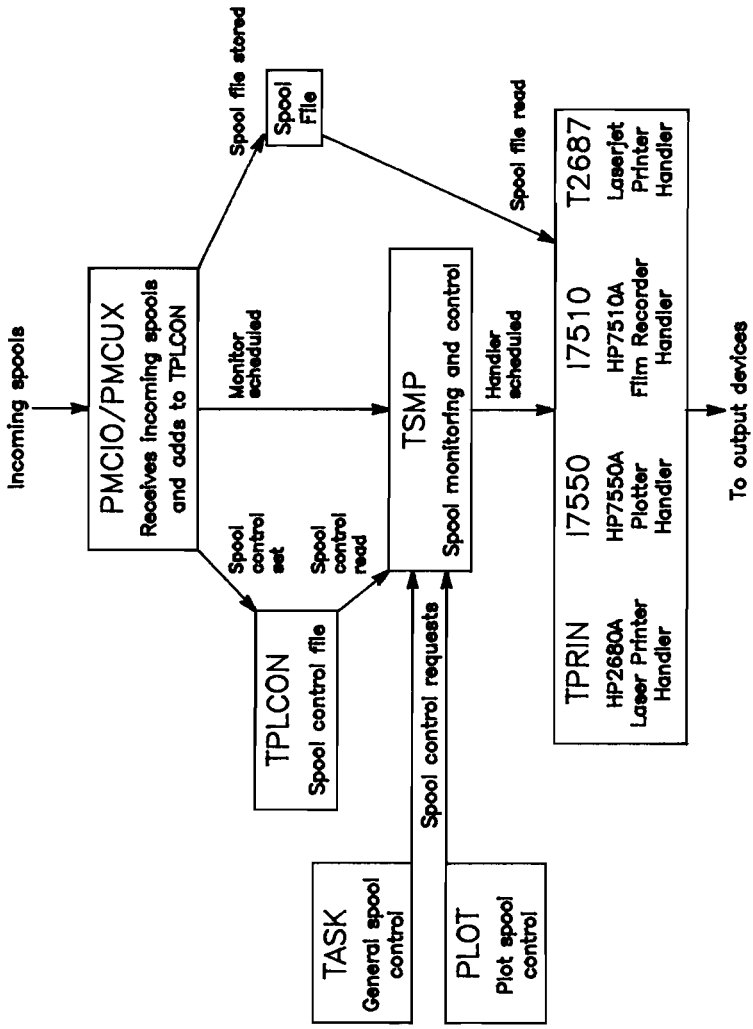


Figure 2
Peripheral Management Computer
(PMC)
Spooling System

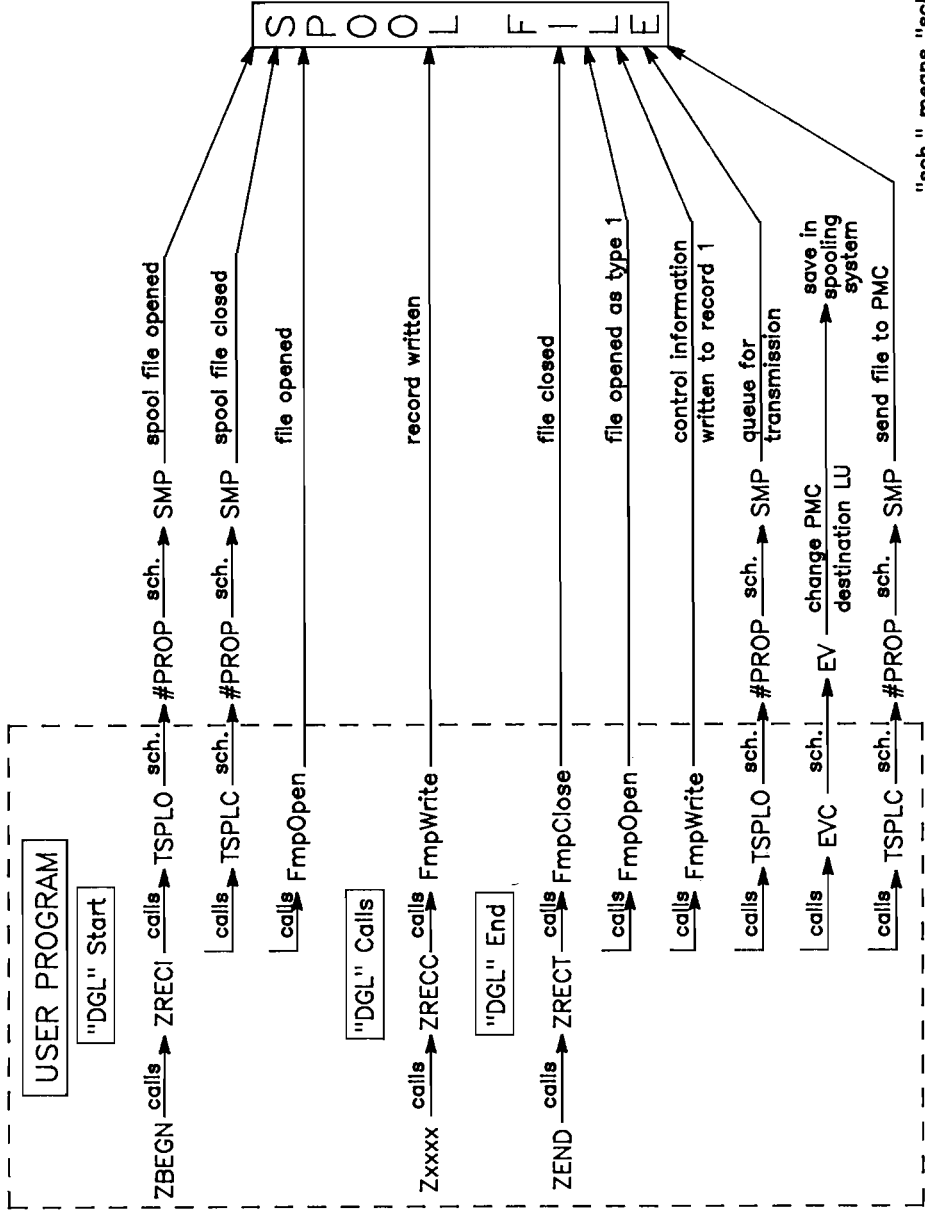


Figure 3 Stub DGL Flow

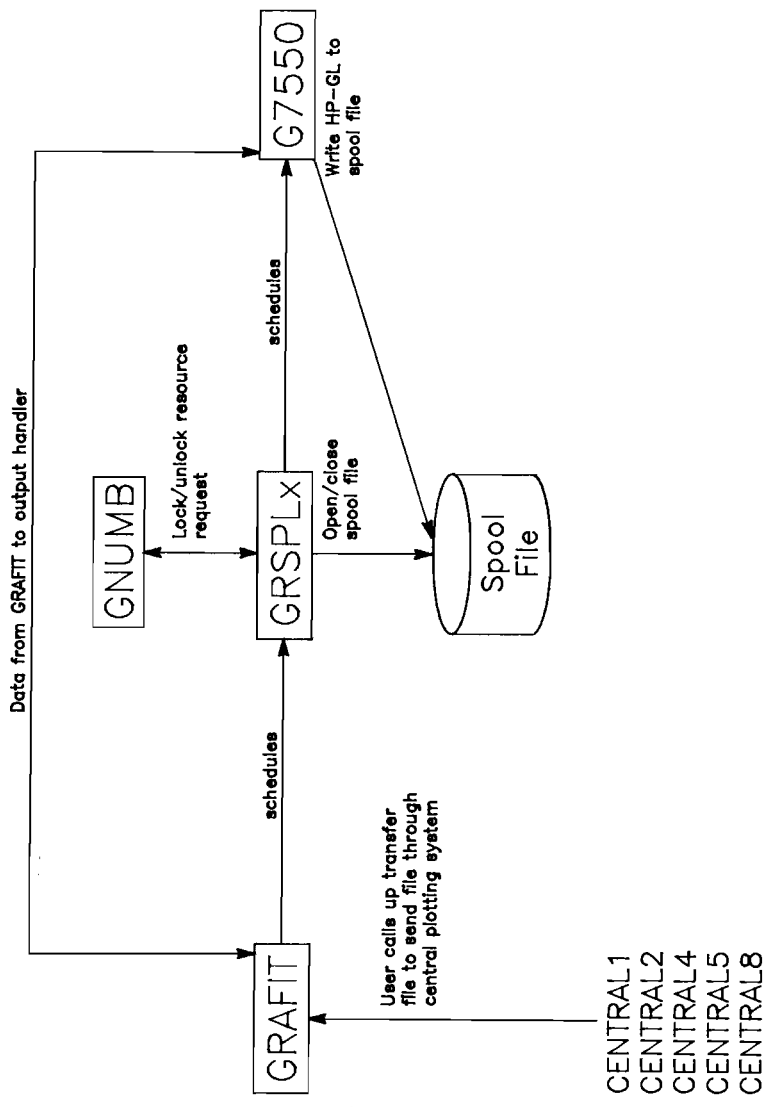
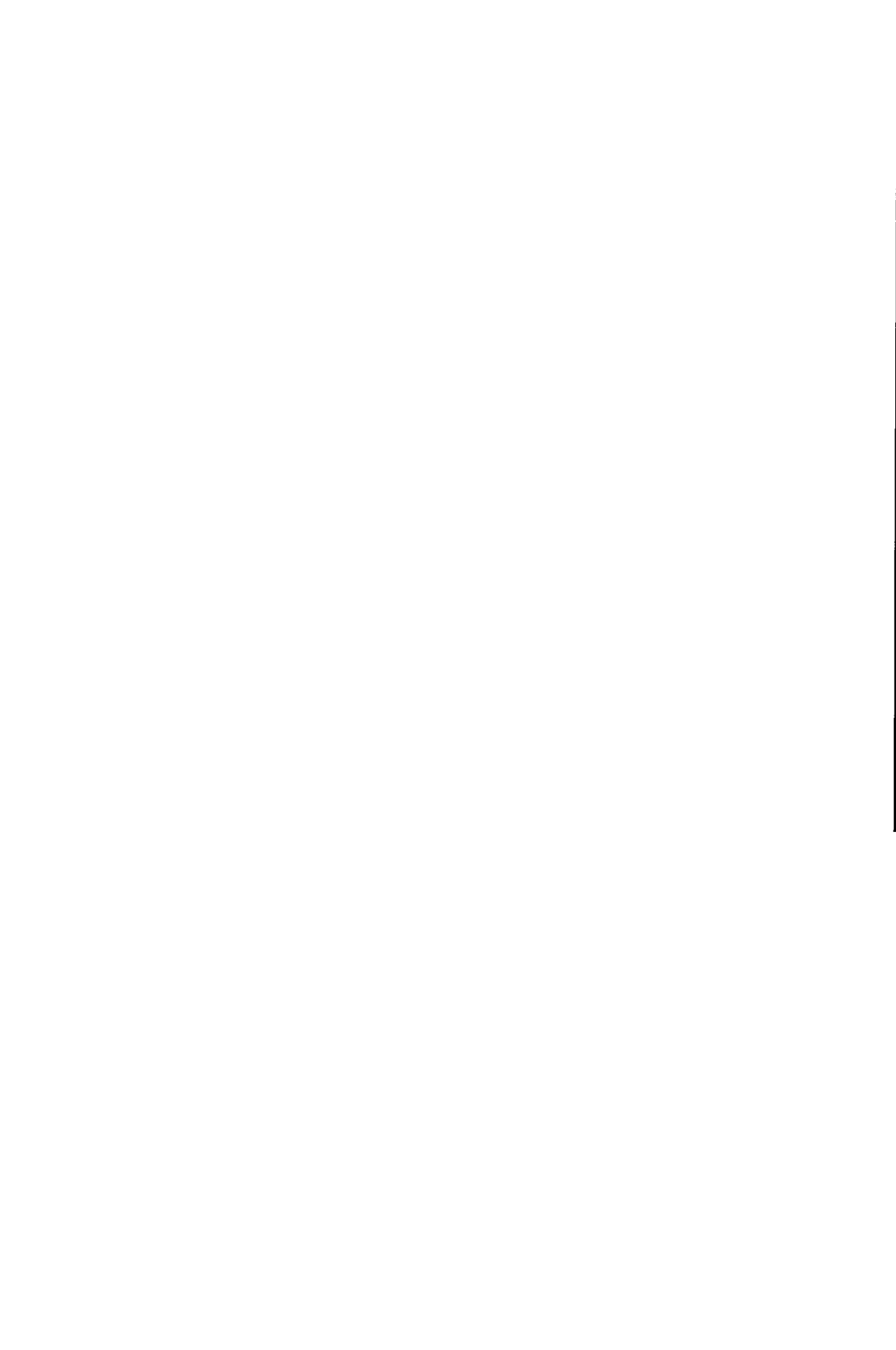


Figure 4
Use of GRAFIT/1000 in the Central Plotting System

- CENTRAL1
- CENTRAL2
- CENTRAL4
- CENTRAL5
- CENTRAL8







Elements of Good Graphing Techniques

Geralyn Clucas
Dan Schober

Graphicus
160 Saratoga Ave. Suite 32
Santa Clara, CA 95051

Abstract

What constitutes a "good graph"? When a graph is designed, quantitative and qualitative data are encoded into a pictorial representation. Inspection of graphs involves the visual/cognitive decoding of the graphically encoded data. A good graph exploits this visual/cognitive decoding process to communicate data quickly, thoroughly and effectively. Knowledge of the graphical perception process facilitates the design of good graphs.

This paper will give general strategies for designing good graphs which are relevant for all areas of science, technology and business. Suggestions regarding effective use of labeling, symbols, legends, scaling, tick marks, and reference lines will be presented. Examples will be given to show how areas, densities, distance, and graph layout affect perception and interpretation of graphs. Comparisons of good and bad graphical techniques and a discussion of how each might lead to different conclusions about the data will be presented.

Introduction

Graphs are employed to consolidate information, exhibit relationships, uncover hidden facts and summarize conclusions. Most importantly, graphs are utilized to communicate ideas. Without careful attention to graphic design, a graph may fail to communicate the pertinent ideas, or worse, may lead to incorrect conclusions.

Considering the proliferation of graphics hardware and software, there has been little emphasis in business or scientific literature on strategies for graphic design. Furthermore, most articles on the subject present practical guidelines which merely reflect personal taste. While graphic design emphasizes an aesthetic component, perceptual considerations are at least equally important. Furthermore, the characteristics and expectations of the audience deserve equal consideration. General strategies for designing graphs shall be presented based on research findings in perceptual science and principles of graphics design. These strategies include the role of audience expectations in graphical perception. From these general strategies, specific guidelines for graphic elements are presented.

Perception Studies

W. Cleveland [2] theorized that certain elementary graphical-perceptual tasks are used to decode information from graphs, and that some of these tasks are performed more accurately than others. Weber's Law [1] for instance, suggests that judgments of position are more accurate than length judgments. S. S. Stevens [6] performed numerous empirical studies of basic perceptual tasks to formulate Stevens' Power Law. Stevens' Law suggests that length judgments are less subject to bias than area and volume judgments.

Cleveland conducted several experiments which attempted to order the accuracy of the graphic perceptual tasks for judgment of magnitude. The findings of his experimentation resulted in the following ordering of these tasks by accuracy:

1. Position along a common scale
2. Position along identical, nonaligned scales
3. Length
4. Angle – slope
5. Area
6. Volume

Cleveland's research also indicated that distance between graphical elements as well as detectability of these elements are also key factors influencing graphical perception. Specifically, greater distance between graphical elements decreases the ability to decode the values of the elements. Obviously, graphical elements which are not easily detectable (i. e. due to crowding) cannot be decoded easily, if at all.

The implications of these studies suggest the following guidelines for graphical design:

- Insure that graphical elements are detectable.
- Minimize distance between related graphical elements.
- Choose graphs which utilize judgments of length and position.
- Use additional visual cues when area judgments are necessary (e. g. pie charts).

Principles of Graphic Art Design

The growing body of literature on graphical design borrows heavily from graphic arts design principles. While perceptual science stresses issues of detectability in graphical design, graphic arts stresses aesthetic concerns. Not only must graphs be easily decoded, they must be pleasing to the eye.

Graphic arts principles [3] which apply to graphical design are as follows:

- Strive for simplicity. Avoid distracting, unnecessary elements. Every element of a graph must be justifiable.
- Maintain unity. Each graphical component must contribute to a unified whole.
- Emphasize only the intended elements. Emphasis must not destroy the unity of the chart.
- Promote balance of graphical elements in graph.

Practical Concerns

The theoretical principles of graph design must be considered within the context of practical applications. Different guidelines apply in different situations. These situational factors may be categorized by the intent of the graph, method of presentation of the graph, and the expectations of the audience.

Statistical literature emphasizes choosing the best graphical method for the type of data presented. Choice of graphical technique also depends on the intention of the graph. For instance, pie charts are used to compare parts of a whole, bar charts are useful for comparisons, and curve charts frequently are used for time series graphing.

Business literature on the use of effective graphics propose different guidelines depending on how the graphs are utilized. For instance, graphs presented in documents are often subject to reproduction and/or reduction. With these constraints, one should choose different linestyles or fill patterns to distinguish data sets within a plot rather than color. Furthermore, all elements of printed graphs must be legible after reduction. Graphs used for publication are usually more formally constructed so as to conform to publication standards. Printed graphs may include more complexity than graphs used in presentations since readers will have more time for examination. Conversely, graphs prepared for projection at presentations must be simple enough to interpret at a glance. All graphs designed for a given presentation should maintain a consistent design format. Good contrast is especially essential if graphs are presented as slides. Color and informal graph layout may be used in presentations.

Characteristics of the audience must also be taken into consideration when designing graphics. The complexity level of each graph must take into account the knowledge level of the audience. Communication is more effective when graphs utilize the perceptual/cognitive habits of the target audience. For instance, Westerners typically read from top left to bottom right. Sorting groups in bar charts takes advantage of the left to right scanning. Horizontal bar charts are usually preferable to vertical bar charts since audiences often associate time comparisons with vertical bar charts. However, vertical bar charts effectively represent data which is expected to vary "up and down", such as temperature. Viewers expect that multiple plots on one page sharing the same dependent and independent data

will have identical scales. Using different scales on one or both axes in this case is very misleading. Color usage should exploit users' expectations. Profit/loss charts should use red to indicate "in the red".

Humans perceive objects in space in terms of distance from self; close objects are in the foreground while farther objects are hidden behind foreground objects. Graphs should be constructed similarly to conform to the adage "no two objects occupy the same space". Allowing prominent graphical elements to hide rather than intersect less important graphical elements provides the illusion of depth as well as emphasis of foreground elements.

The following practical concerns must be included in the graphic design strategy:

- Choose the correct graph for the situation.
- Consider the medium of presentation when designing graphs.
- Conform to the audience's expectations.
- Insure that important graphical elements hide less important elements.

Design of Graphical Elements

The strategies previously discussed will now be applied to generate specific guidelines for generating good graphs. These guidelines address the implementation of specific graphic elements as well as their integration into the completed graph once the choice of graph type (e. g. pie chart vs. bar chart, etc.) has been made.

Labeling

Few graphs successfully communicate solely with pictures. Text is used to focus attention immediately on the topic of the data as well as to label the data. Due to the important role played by text in graphs, labels should always be prominently featured.

Any text on a graph deserves prominence over other graphical elements. Graphical prominence is achieved by placing the emphasized element in the foreground of the graph. Therefore, text should always be *blanked* against other graphical elements. Blanking refers to the assurance that prominent graphical elements are not intersected by other elements, thus promoting the illusion that the prominent elements are in the foreground of the graph hiding the less important background elements. Figure 2 provides an example of text blanking.

Cleveland [2] emphasized the role of distance in graphical perception, specifically stating that related data should be placed in close proximity. One implication of this principle is that data elements should be labeled directly on the graph rather than on a legend. Other graphics experts agree with this guideline [3], although not universally [5]. Figure 1 illustrates the conventional use of legends with curve charts. A more natural placement of the labels in this plot is next to each curve, blanked against the background, as in Figure 2. Pie chart slices could be labeled next to each slice. Since area is not an effective indicator of

value [6], pie charts should be labeled with the data labels and values. Figure 3 illustrates the standard use of a legend with a pie chart, while Figure 4 illustrates more effective labeling of the values and data types directly next to the pie slices in a pie chart.

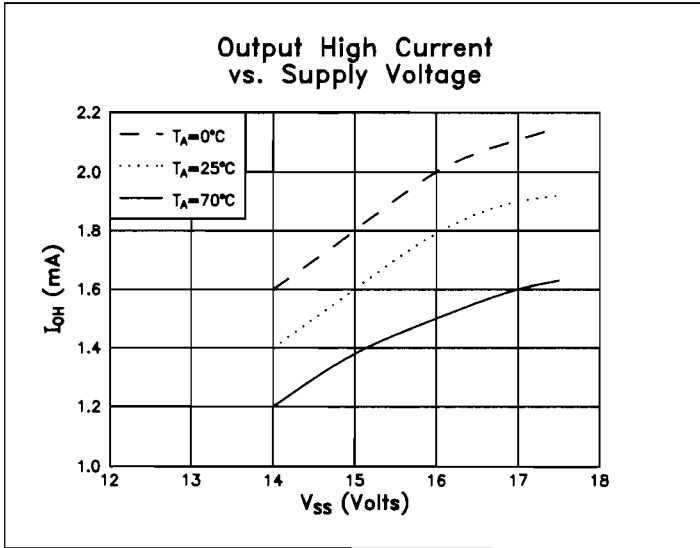


Figure 1: Curve Chart with Legend

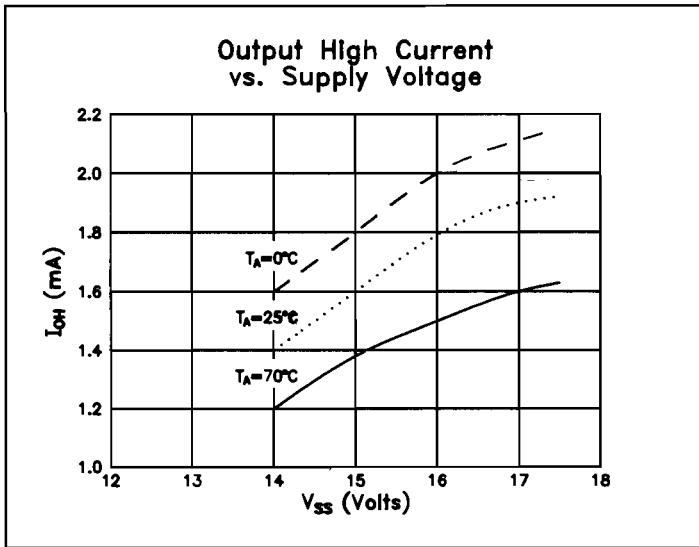


Figure 2: Curve Chart with Blanked Text Labels

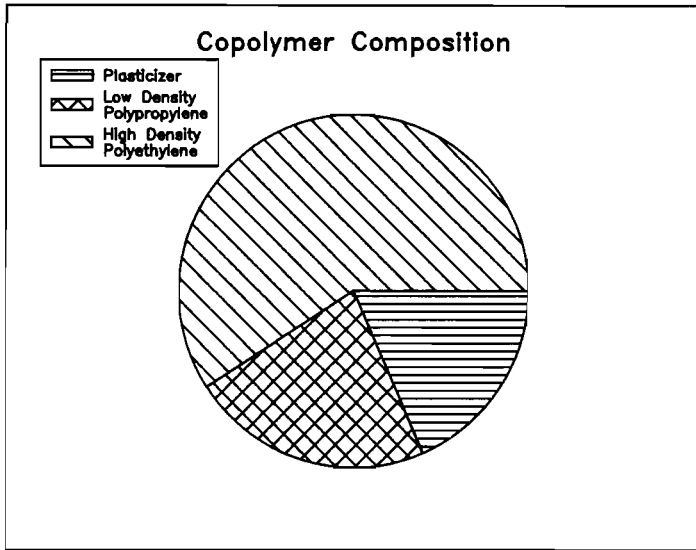


Figure 3: Pie Chart with Legend

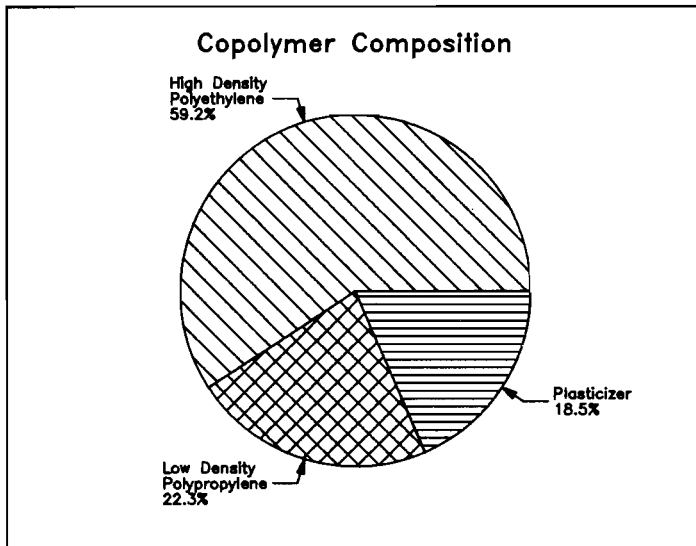


Figure 4: Pie Chart with Values Labeled

Whenever labeling data directly produces a cluttered graph, legends are the better alternative. The legend should be placed in balance with other elements of the graph without obscuring the data.

Certain text characteristics may be utilized to produce more effective graphs. The text font should be a simple sans serif font. The sans serif style (characterized by absence of stylistic details) is easier on the eye, and reproduces better than more ornate text styles. Text should be presented in mixed upper and lower case. The contrast between mixed-case text results in more readable text. Text should be placed so that it may be read without turning the chart (or the head). This implies that all text (even the Y axis name) should be upright and parallel to the horizontal axis. Placement of the Y axis name should also take into consideration the audience; business graphics typically place the Y axis name upright at the upper left corner of the graph, while scientific graphics place the label perpendicular to the Y axis. The perpendicular placement of the Y axis label is also preferable to upright placement when the label is long.

Labels should never be redundant. Extraneous labeling produces clutter and de-emphasizes other important features of the graph. For instance, if the X axis data is specified in the title, it need not be redefined on the axis.

Other guidelines are relevant for text-only graphs. Text should be presented flush-left rather than centered or flush right. Flush left works best since the leftmost position is used as an anchor, enabling the eye to track across each line and return to the same starting position. Full justification is not as effective since variable word spacing is less readable than fixed word spacing. Text items should consist of few short phrases or keywords per graph (no more than 6 per page). One text font and no more than two faces (e. g. basic face and italic or basic face and bold) should be used for all graphs in one paper or presentation.

Scaling

Choice of scales on the X and Y axes is critical for providing a clear, undistorted summary of the data. For instance, data with very little fluctuation may appear quite variable by choosing a very small scaling interval. Another example involves comparing several plots with the same dependent and independent variables. All plots which are compared in this manner should use identical X and Y scales. Figure 5 illustrates how using different scales for this type of comparison might lead to the wrong conclusions. Figure 6 shows the correct way of presenting comparisons of this nature.

Scales should be chosen which best represent the data ranges. Log scales for one or both axes should be chosen when the data follows a logarithmic trend. The base of the log should be naturally easy to interpret, such as 2 or 10. Similarly, the non-log scale intervals and labels should be rounded to easily interpreted values, such as 0, 10, 20, rather than 0, 7, 14.

An oft-ignored scaling issue involves the representation of discontinuity in the axes and/or data. Breaks in either axis should be graphically represented; otherwise, viewers of the graph might erroneously use distance cues to discern difference in magnitude. A good example of this error is illustrated in Figure 7; the distance on the X axis between $-\infty$ and -7 is not a meaningful indication of the magnitude difference between those data values on the graph. Figure 8 calls attention to this illusion by emphasizing the break in the X axis and the data curves.

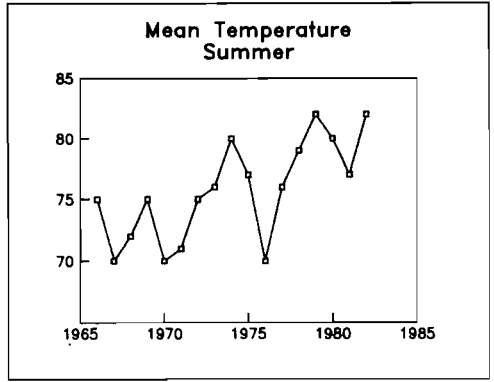
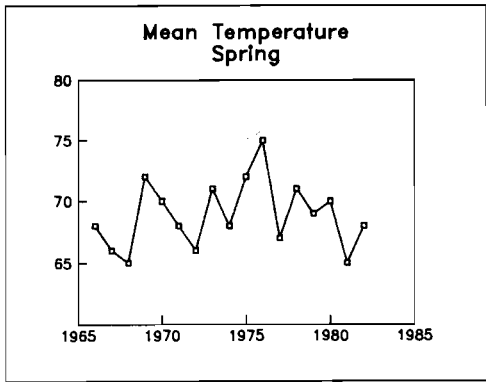


Figure 5: Misleading Use of Scales: Multiple Plots

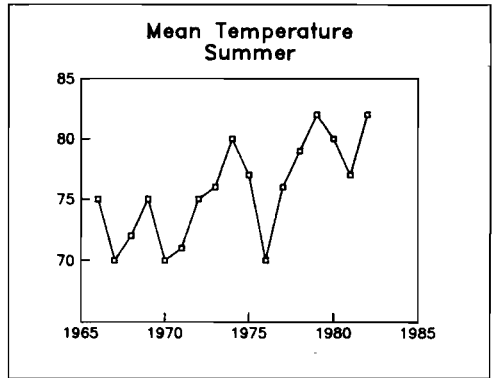
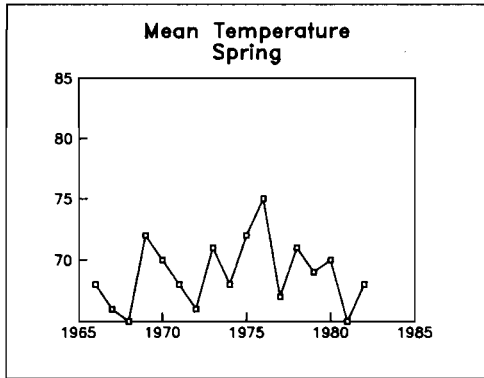


Figure 6: Correct Scaling for Multiple Plots

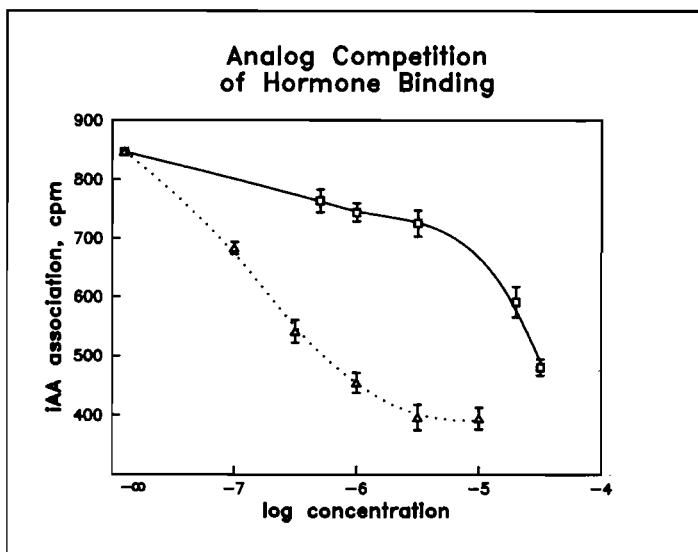


Figure 7: Misleading Use of Discontinuous X Axis

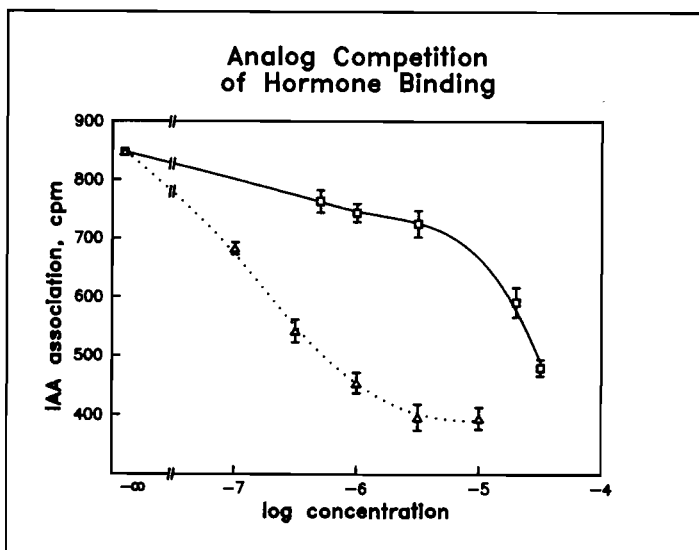


Figure 8: Correct Indication of Break in X Axis

Another potential source of distortion involves missing data values. Generally, connected curve charts with missing data values are presented as continuous curves in spite of the discontinuity. The absence of data values should be emphasized if the absence is important or if the absence somehow distorts the implications of the graph. For instance, if the absent value would generally reflect an unrepresentative value on the graph, its absence should be emphasized. Missing data values are typically emphasized by breaking the continuous curve at the discontinuous point (thereby creating 2 curves with the same linestyle) as illustrated in Figure 9.

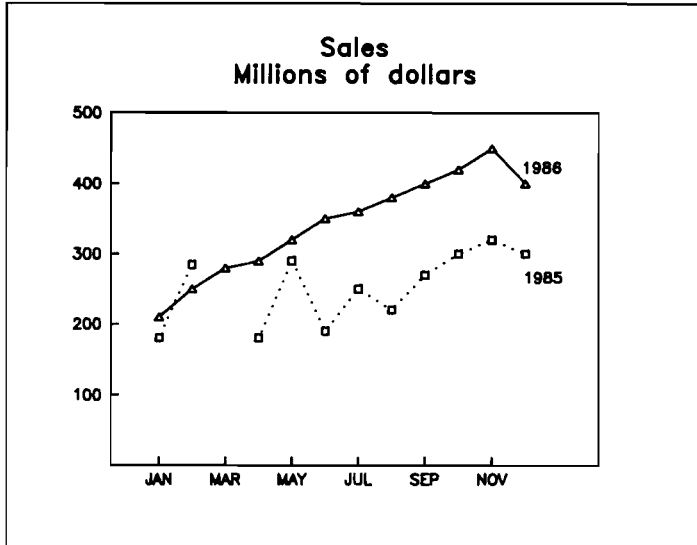


Figure 9: Representation of Missing Data Values

Tick Marks and Reference Lines

Tick marks facilitate interpolation of the exact data values represented in the graph. Ineffective use of tick marks may create clutter and confusion.

Good graphs have only as many tick marks as are necessary for interpretation of the data. Publication quality graphs place tick marks either completely inside or outside the graph frame rather than crossing the frame. Inside placement of ticks serves to focus attention within the plot frame. However, bar charts or graphs with filled areas bordering either axis should use outside placement of ticks so that the ticks do not obscure the data area. Tick mark labels should naturally describe the data values (e. g. month labels rather than numbers should be used to describe monthly data). Finally, when the graph size must be reduced for publication, the graph designer must insure that the tick marks and labels are legible after reduction.

Grid lines are used instead of tick marks when data values must be interpreted more exactly. In this case, grid lines should never clutter the graph. Grid lines should be de-emphasized with respect to the data, axes and labels. Therefore, grid lines should be hidden behind other graphic elements in a less noticeable linestyle (e. g. thinner or dotted). Under no circumstances should grid lines obscure the graph by appearing "in front" of the data. Grids are often used in conjunction with log scales since the unequal spacing between tick intervals interferes with visual tracking across the graph.

Tick marks on both sides of the graph provide a good compromise between the use of grids and use of tick marks. This scheme produces a less cluttered graph than a comparable plot with grid lines, but does not provide the ease of interpolation provided by grids.

Use of Color

Color is primarily used to differentiate and emphasize elements in a graph. Color may be used quite effectively in presentation graphics, but should be avoided when the graphs must be reproduced in black in white for publication. Colors should be chosen carefully to maximize their utility.

Colors may be used to distinguish between data types in curves, bar and pie charts. One may combine the use of different linestyle and color for each data curve in a plot. Color and fill styles on bar and pie charts, however, should be used interchangeably to minimize distraction. One should limit the number of colors used in a graph to a maximum of five.

Color provides emphasis when used sparingly. The use of black and white graphs with one color used for emphasis is simple and surprisingly effective. Text-only graphs should always have a maximum of three colors, one background color, one primary text color, and one text color used for emphasis. The same color should be used for emphasis for all graphs in a report or presentation.

Color may be used with graph position to order data categories. The darkest color on stacked bar charts should be reserved for the bar representing the most important category in order to lend informal emphasis. Also, the most important category should be placed closest to the axis so that the eye is drawn to the base of the plot by graph position and color, and then proceeds outward from the base to the less important data elements.

Color choice should be guided by symbolism, utility, aesthetics, and avoidance of optical illusions. Using blue as a background color in sales presentations takes advantage of the symbolic association of blue with authority. Pure saturated colors are good for emphasis in text and for use with data lines in curve graphs, but should be avoided when representing data in bar and pie charts. Brightly colored areas focus undue attention to the colors at the expense of the more important graph elements, and often invoke unpleasant after-images. Graph designers should avoid placing complementary colors next to each other in a graph (e. g. blue and yellow) to avoid the optical illusion of a wavering line at their border. Colors should be chosen such that they blend well in the graph, yet provide enough contrast to be distinguishable.

Fill Styles

Fill styles are used to distinguish or emphasize data areas on a graph. Fill styles should be used instead of color for distinguishing areas on a graph slated for black and white reproduction. Since fill patterns are not as easily detected as color, the graph design process must carefully take into account the presentation medium. Reduction of graphs for documents may render fill patterns less detectable. Similarly, projection of graphs might affect the ability to distinguish between similar fill patterns in close proximity.

When different fill styles are placed in close proximity, the eye is usually drawn to the most dense pattern. This optical tendency may be exploited to emphasize the more important elements of the graph. For instance, stacked bar charts should be ordered by importance from the axis outward and should also vary in fill style from dense to light. Matching dense patterns with large pie slices, however, throws a pie chart out of balance. Instead, large pie slices should be matched with large, less dense fill patterns, while smaller slices use smaller, more dense fill styles.

The fill patterns used in a graph should be simple and few. Attention should be given to the ordering of fill styles to promote differentiability between classes and avoid optical illusions. For instance, placing fill styles with lines which go in opposite directions next to each other produces the illusion of motion (see Figure 10). Fill patterns should progress from solid, to shaded, to hatched, to half-hatched. If more fill styles are needed, the patterns may be extended by using different linestyle. Since the progression is from dark to light, the fill patterns should be varied in order to maximize contrast. Figure 11 illustrates the effective use of fill styles in a stacked bar chart.

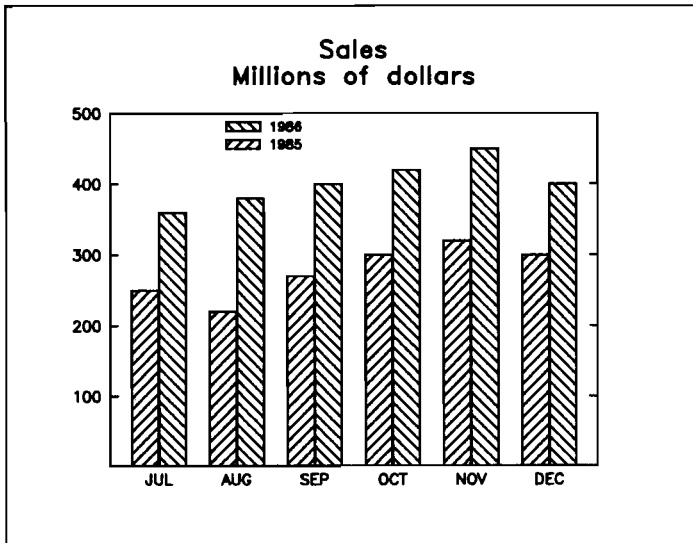


Figure 10: Fill Pattern Creating Optical Illusion

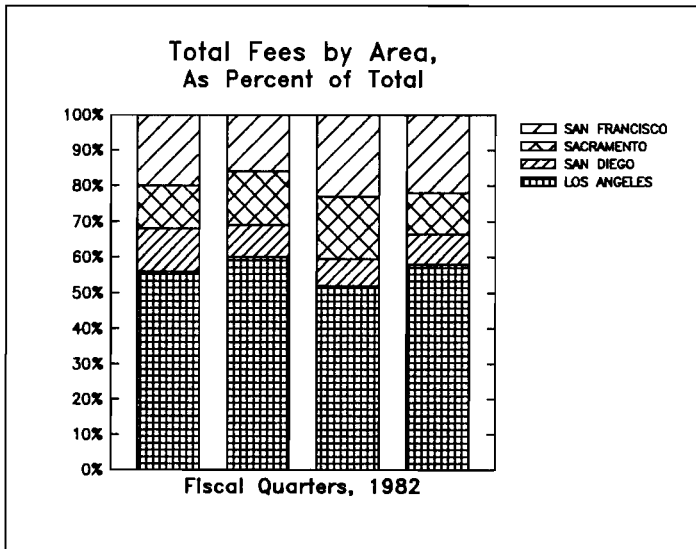


Figure 11: Effective Use of Fill Patterns

Graph Layout

The overriding principle guiding the design of good graphs is simplicity. Each graphic element must be justifiable and necessary to convey the desired message. This principle also applies to the presentation of several classes or types of data on one graph. The graph designer should avoid using more than five curves per curve chart or six slices per pie chart whenever possible.

Emphasis is another important component of graph layout. Internal emphasis of important graphical elements may be achieved using blanking. Text blanking is desirable so that the explanatory labels are not obscured. Symbol blanking is another desirable feature which promotes detectability of the data elements. Figure 12 shows a typical curve chart, and Figure 13 illustrates how the same data presentation is improved by creative use of labeling and symbol blanking.

Graph placement may be used to emphasize important graphical elements. The most important class in a stacked bar chart should be placed nearest to the X axis for vertical bar charts, or nearest to the Y axis for horizontal bar charts, with the other classes following in descending order of importance. Combining this placement scheme with the use of fill styles or color increasing from dark to light provides a most effective presentation. Emphasis of one slice of a pie chart may be achieved quite dramatically by "taking a slice" of the pie. Slice *exploding* achieves this effect.

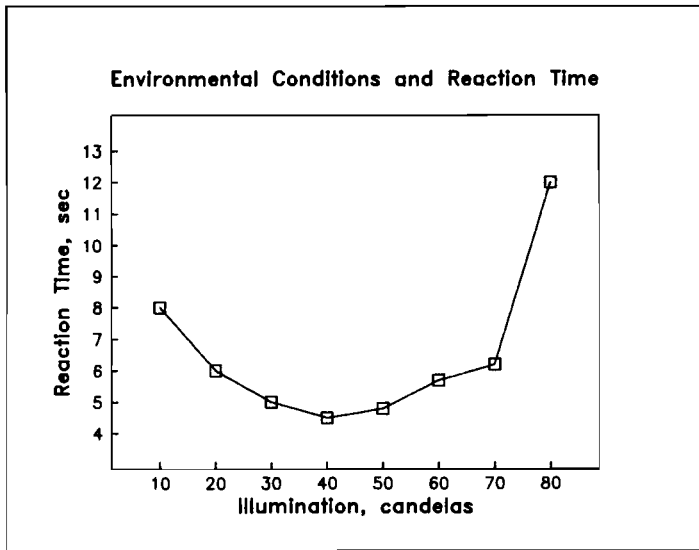


Figure 12: Typical Curve Chart

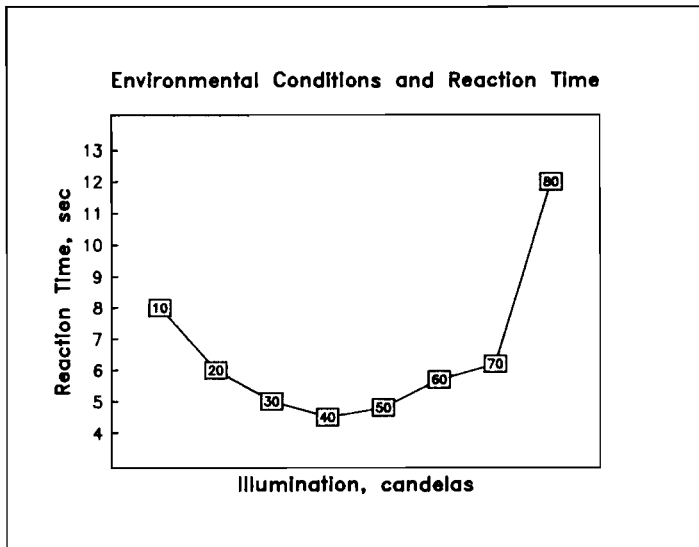


Figure 13: Curve Chart Using Symbol Blanking

Good graphs are well-balanced and convey a sense of unity. Imbalance may be corrected by clever use of text; there is no rule, for instance, that titles must be centered above the graph. Using a border around the graph lends a sense of unity to the graph, while using the axes frame further focuses attention on the data.

Continuity of design should be used for all graphs in one document or presentation. The same colors, fill styles, linestyles, and symbols should be used consistently for all graphs, as well as text font, placement on page (vertical or horizontal), and background color.

Summary

Good graphs are produced only through careful and critical planning, with special consideration given to the intention of the graph, characteristics of the audience and the medium of presentation. Use of these guidelines combined with concern with situational constraints and careful choice of graph type will facilitate design and implementation of good graphs.

All figures in this paper were generated using the Graft software package on an HP9000/320 computer running HP-UX. Graft is a commercially available interactive graphing package developed by Graphicus. Graft generates publication-quality graphs on HP computers running HP-UX and RTE-A for business, technical and scientific applications. This package provides all the necessary graphical elements and options previously outlined for producing quality graphics.

References

- [1] Baird, J. C. and Noma, E. 1978. *Psychophysical Analysis of Visual Space*. New York: John Wiley & Sons.
- [2] Cleveland, W. S. 1985. *The Elements of Graphing Data*. Monterey, CA: Wadsworth.
- [3] Matkowski, B. S. 1983. *Steps to Effective Business Graphics*. San Diego, CA: Hewlett-Packard.
- [4] Mathews, E. 1986. "Graphics That Count", *Proceedings of the National Computer Graphics Association*, 1:83-95. Fairfax, VA: National Computer Graphics Association.
- [5] Rawlins, M. G. 1986. "Getting More Out of Your Graphics Software and Hardware Investment", *Proceedings of the National Computer Graphics Association*, 1:34-68. Fairfax, VA: National Computer Graphics Association.
- [6] Stevens, S. S. (1975). *Psychophysics*. New York: John Wiley & Sons.



Graphical Techniques in Data Analysis

Bill Carson

Graphicus
160 Saratoga Ave. Suite 32
Santa Clara, CA 95051

Introduction

Graphically portraying data provides a clearer and more penetrative understanding of data. It tends to show data sets as a whole, allowing us to quickly summarize the general behavior as well as studying detail. Up until now, books on graphical techniques were either too incomplete, stopping at a histogram or pie chart, or were too technical and not readily available in computer programs. Furthermore, many graphical techniques were just appearing in statistical journals and thus were not readily accessible to the statistically unsophisticated data analyst.

With the recent rapid proliferation of graphics hardware accompanied by a steady development of software this is no longer the case. Therefore, this paper will give an overview of various old, but not widely known, and new methods of graphically portraying data. The graphical techniques presented will be relevant in all areas of science and technology. Techniques for graphically exploring labeled data, distributions, two-way tables, and relationships between variables will be presented. Advanced techniques in regression analysis, analysis of variance, time series, and multivariate analysis will also be touched on.

Graphically Exploring Data

There is no statistical tool that is as powerful as a well-chosen graph when we want to understand the basic characteristics of a set of data. In short, we need to understand the distribution of the set of data values such as where they lie along the measurement axis and what kind of patterns they form. This often means asking additional questions. Are any of the observations outliers, that is, values that seem to lie too far from the majority? Are there repeated values? What is the density or relative concentration of observations in various intervals along the measurement scale? Is the data symmetrically distributed? What kind of relationships exist between the observations?

One way to represent a set of data is to present the data in a table. Many questions can be answered by carefully studying a table, especially if the data has been ordered. In a sense, a table contains all the answers, because apart from possible rounding, it presents all of the data.

However, many distributional questions are difficult to answer just from peering at a table. Plots of the data can be far more revealing, even though it may be harder to read exact

data values from a plot. In the following sections we will show some graphical techniques that can be used to explore shapes and patterns of a set of data.

Labeled Data

The analyst often needs to display measurements of a quantitative variable in which each variable has a label associated with it. The most common way to do this has been with a pie or bar chart. These techniques are widely used and understood by nontechnical people, however, each has some drawbacks.

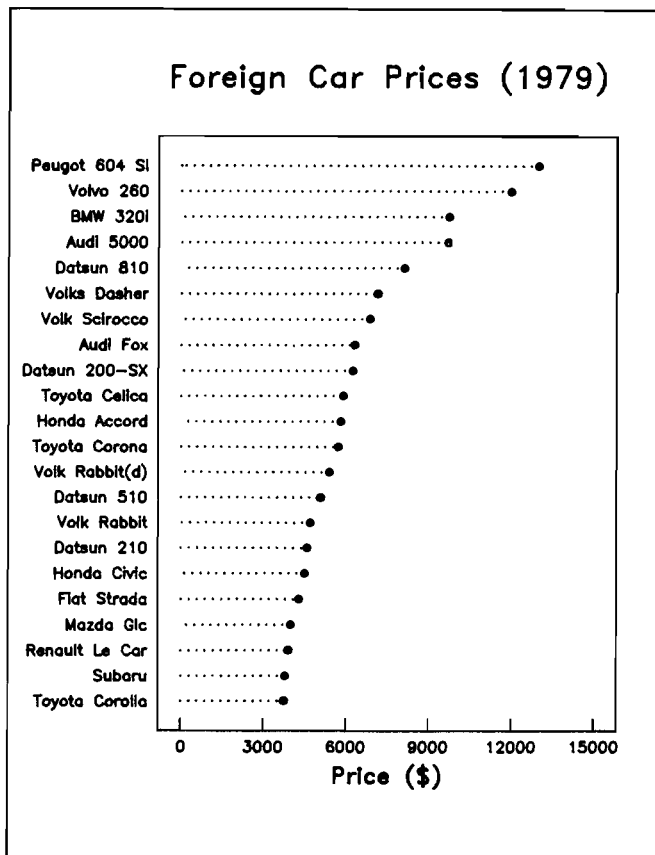


Figure 1: Dot Chart

The major drawback with a pie chart is that it can be very difficult to make angle judgments. This leads to difficulty in interpreting the relative sizes of each slice in the pie, especially if most of the slices are about the same size. The major drawback with a bar

chart is that it requires a meaningful baseline if the length and area of the bar is to be meaningful. Since a meaningful baseline value is usually zero the bars waste space and degrade the resolution of the values if the values are large.

A graphical technique that is used without these drawbacks is called a *dot chart*. Each data value is represented on the chart by a dot and is plotted in order from smallest to largest. Each dot is connected with its label by a dotted line. This method allows the analyst to make judgments along a common scale rather than angle judgements. Data that can be portrayed in a pie chart can always be portrayed in a dot chart.

When there is a zero on the scale of a dot chart then the dotted lines can end at the data dots. The dotted lines should go across the graph when the baseline value has no particular meaning. The reason for this is that when the dotted lines stop at the data dots there are two aspects to the plot – the lengths of the dotted lines and the relative positions of the data dots along the common scale. This gives the visual appearance of a bar chart. However if the baseline has no meaning then the lengths have no meaning. By making the dotted lines go across the graph the portions between the left vertical scale line and the data dots are visually de-emphasized.

Figure 1 shows a dot chart of the prices of 22 foreign cars in 1979. The chart shows that 17 of the cars have prices ranging from just over \$3,000 to just over \$6,000. The last 5 cars then increase rapidly to over \$12,000.

Single Distribution

When the analyst needs to get a visual impression of the distribution then a histogram is commonly used. The problem with a histogram is that the visual impression depends on the fairly arbitrary choice of the number and placement of the intervals. When a histogram is made, the interval width of the histogram is generally greater than the data inaccuracy interval, so accuracy is lost. As we decrease the interval width of the histogram, the accuracy increases but the appearance becomes more ragged.

The stem-and-leaf diagram is a compact way of recording the data. Instead of having a separate table listing the data and a histogram to show the distribution, the stem-and-leaf diagram combines both sources of information into one plot. This can be important for reports and published papers where data is presented and analyzed.

The diagram is used in much the same way as the histogram. It gives the analyst information about the symmetry and skewness of a distribution. Figure 2 shows a stem-and-leaf diagram for the prices of 74 car models sold in 1979. Each value on the left of the colon represents the price in thousands. Each value on the right represents hundreds. The diagram shows that most of the car prices range from \$3,300 to \$6,800. The prices that occur most frequently are \$4,500 (5 times), \$4,200 (4 times), and \$4,700 (4 times).

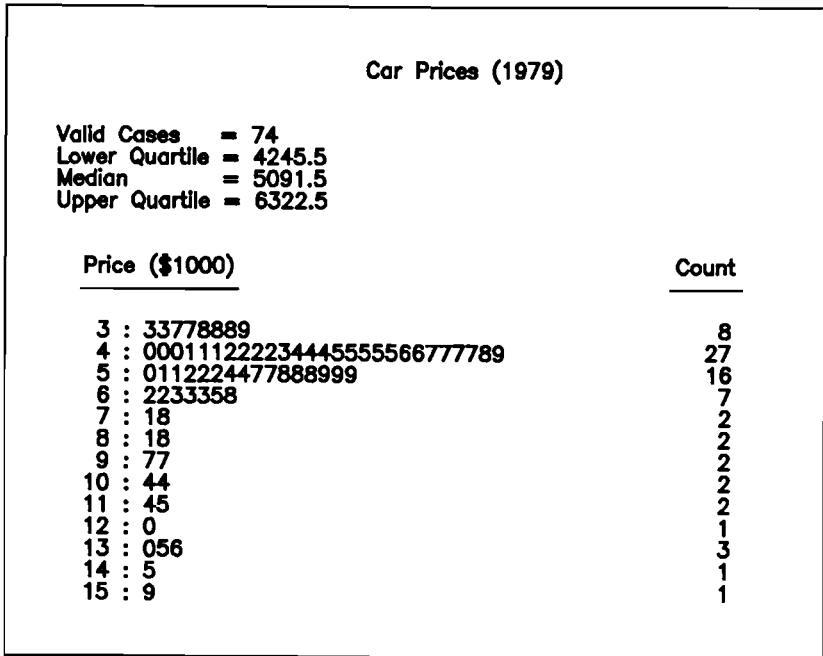


Figure 2: Stem-and-Leaf Diagram

Comparing Distributions

While the techniques described above are usually good for looking at one distribution of data they are relatively poor when trying to compare more than one but similar distributions. Two simple but not as commonly used techniques are the *percentile comparison plot* and the *box plot*.

When distributions are compared, the goal is usually to rank the categories according to how much each has of the variable being measured. The most effective way to investigate which of the two distributions has more is to compare the corresponding percentiles. The *percentile comparison plot* graphs the percentiles of one distribution against the corresponding percentiles of the other distribution. The advantage of a plot like this is that not all the data has to be plotted to characterize the differences between the two distributions. Data distributions can be complicated, and when they are, the percentile comparison plot can reveal just how complicated it is.

Figure 3 shows a percentile comparison plot where the payoffs from the 1976 New Jersey lottery are plotted against the payoffs from the 1981 New Jersey lottery. Typically, the percentiles that are plotted are 1,2,...,5; 10,20,...,90; and 95,96,...,99. If the distributions

are similar then the points should lie near the 45 degree reference line. From the plot, the analyst can conclude that the two distributions are similar when the payoff is small but they are very different when the payoff is large. The 1976 lottery seems to have higher payoffs more frequently.

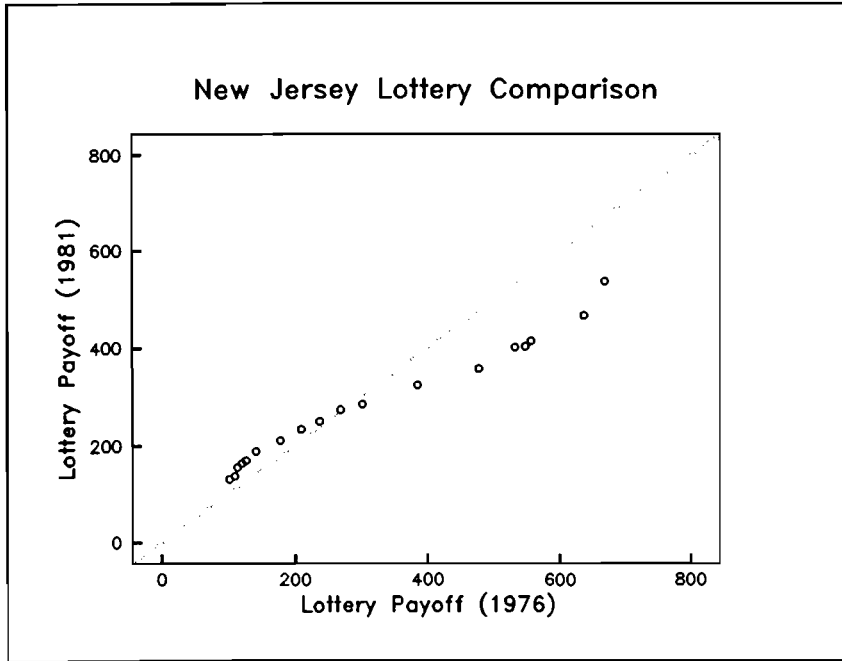


Figure 3: Percentile Comparison Plot

While the percentile comparison plot is a good way of looking at all the data there are stages in the analysis procedure where it is useful to summarize the distribution. The *boxplot* gives a quick impression of the locality, spread, and skewness of a distribution. In the boxplot the upper and lower quartiles of the data are portrayed by the top and bottom of the rectangle and the median is portrayed by a horizontal line segment within the rectangle. The lines that extend from the ends of the box show how stretched the tails of the distribution are. The individual values outside the lines give the analyst an opportunity to consider the question of outliers.

Figure 4 shows a boxplot for the payoffs of the New Jersey lottery in 1976, 1977, and 1981. From the plot the analyst can see that the median payoff for each lotteries is about the same but the spread of the payoffs are different. There were more high payoffs during the 1976 lottery than either the 1977 or 1981 lottery.

Boxplots have many strengths. For one, it can show the symmetry of a distribution. If the distribution is symmetric then the median cuts the box in half, the upper and lower lines are about the same length, and the outside values at the top and bottom, if any are about equal in number and symmetrically placed. Another strength is to be able to compare distributions by comparing corresponding percentiles.

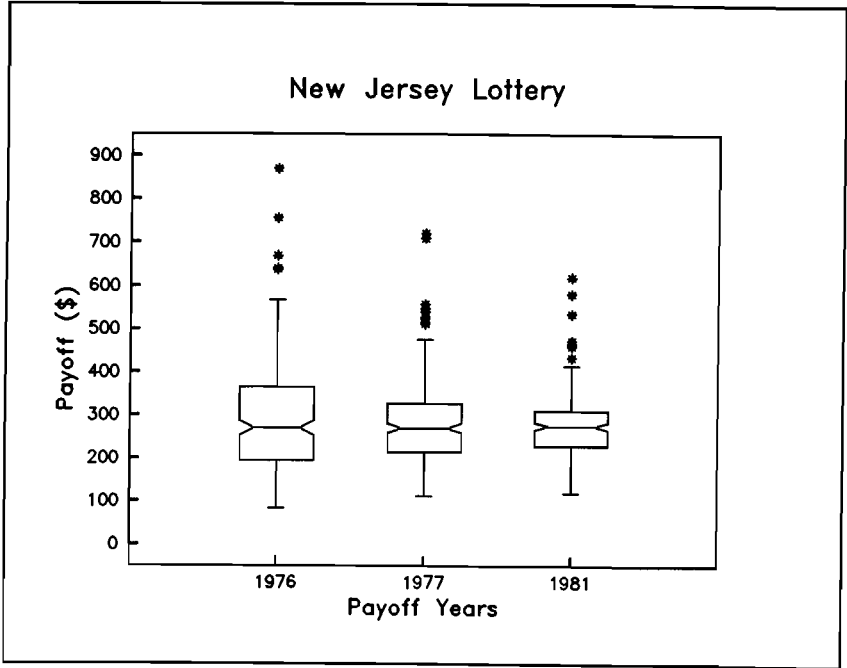


Figure 4: Boxplot with Notches

In applications where comparing locations is important box plots can be drawn with notches in their sides to help guide our assessment of relative location. A suitable informal interpretation of the plot with notches is that if the notches for any two boxes do not overlap then the analyst can regard it as strong evidence a difference in their medians exists at the .05 level. The notches provide an approximate 95% test of the null hypothesis that the true medians are equal. In Figure 4, each of the notches overlaps with each other leading the analyst to conclude that there is no difference in their locations.

The advantage of this plot is that it is a useful guide for comparing median levels even when the requirements for the hypothesis test are not strictly met – which is very frequently the case. However, the analyst should be careful when comparing more than two sets of data because the notches are not adjusted to take into account that several hypothesis tests are being carried out simultaneously. This is the so-called "multiple comparison" problem.

Technical adjustments are possible, but generally unnecessary, as long as the notched box plots are used informally.

Two-way Tables

Often the relationship between two categorical variables is represented in a two-way frequency table. Each cell of the table contains the number of observations, and row, column, and total percentages. The problem with looking for a relationship from a table like this is that it is difficult to pick out densities from the numbers in the table.

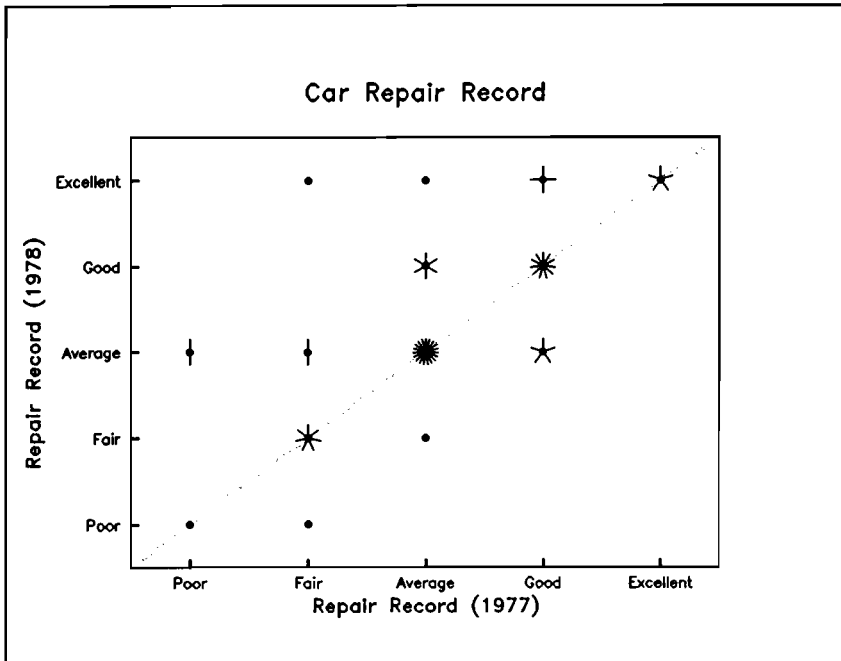


Figure 5: Scatterplot with Sunflowers

Typically, the most powerful way to look at a relationship between two quantitative variables is with a scatter plot. The scatter plot can also be used to look at the relationship between two categorical variables. The scatter plot for two categorical variables has one inadequacy: there is a large number of overplotting of points because of the discrete data. Each plotted point on the scatter plot could represent more than one point in the table which would mislead the analyst about the density of the data in different regions.

One solution to the overlap problem is to take the number of points in each cell and portray the counts by symbols called *sunflowers*. A single dot is a count of 1, a dot with two line

segments is a count of 2, a dot with three line segments is a count of 3, and so forth. Figure 5 shows a scatterplot with the repair record rating of 74 car models in 1977 against the repair record rating of the same 74 car models in 1978. Since the ratings are on a scale of 1 to 5 there were only a few places to plot 74 points. The plot shows the highest density of points along the 45 degree line. The analyst concludes that the ratings usually stayed the same from 1977 to 1978. The most frequent combination was 'average, average' (19 times) and since there are more points above the line than below then the ratings that did change from 1977 to 1978 were for the better.

Sunflowers, since they provide the analyst with a portrayal of counts of points in different regions of the plot, are a type of two-dimensional histogram. Thus their use extends beyond portraying overlap to any situation in which seeing count information is helpful.

Relationships Between Two Variables

Scatter plots are often used to judge whether there is a dependence between two quantitative variables. This might not always be an easy judgement to make from the scatter plot if a large number of points are plotted. Often a line is drawn through the points to represent the dependence between the variables. The classical method for fitting a line to the data is to use polynomials, usually straight lines or quadratics. The problem with polynomials, even those with degrees higher than 2, is that they are neither flexible nor local. What happens on the extreme right of the scatter plot can very much affect the fitted values at the extreme left. Also, polynomials have difficulty following patterns on scatter plots with abrupt changes in the curvature. A better "smoothing" procedure needs to be used.

By smoothing we mean computing and plotting another set of points. A smoothing technique, called *lowess* (locally-weighted scatterplot smoother), gives the analyst an accurate impression of whether the data is linear or non-linear. Lowess has a robustness feature in which, after a first smoothing is done, outliers are identified and then downweighted in a second smoothing. Figure 6 shows a scatterplot with the miles-per-gallon of 74 car models in 1979 against the price of the cars. The fitted line shows that the higher priced cars don't get very good mileage but the lower priced cars range across all the mileage figures. This shows that the relationship between these two variables is definitely not linear.

The amount of smoothness is controlled by a parameter that ranges from 0 to 1. The closer this parameter is set to 1 the straighter the line through the points. The closer this parameter is to 0 the more the curve goes through the individual points. In most applications this parameter is usually set between .5 and .8.

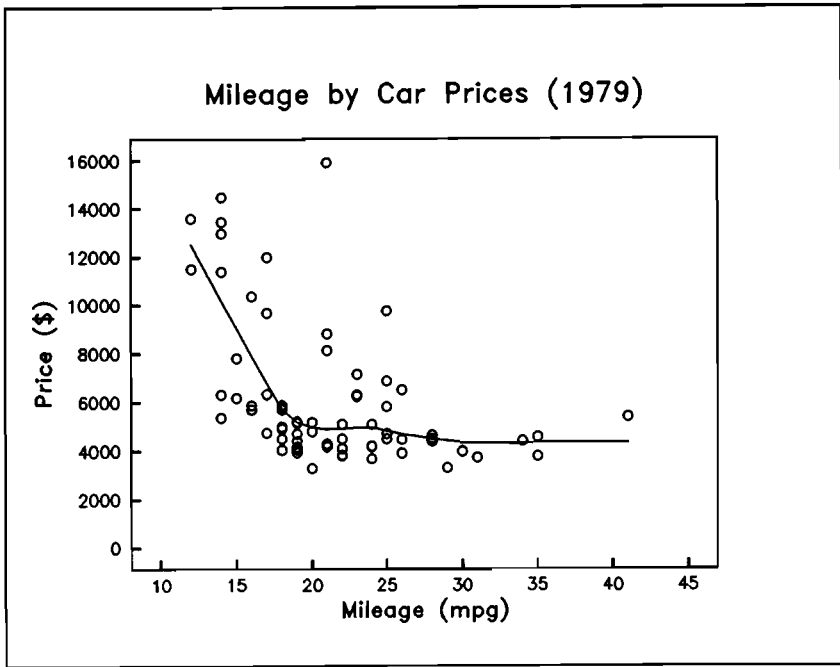


Figure 6: Scatterplot with Locally Weighted Smoothing

Another technique that gives the analyst summary information from a scatter plot is to superimpose a *rangefinder box plot* on the graph. This plot is particularly helpful in the exploratory stages of an analysis when the analyst is on the lookout for unusual values and combinations of values.

The rangefinder box plot contains precisely the same information as the box plots described in the previous section. The two central line segments intersect at the cross-median values. Figure 7 shows that the median car model is about \$5,000 and gets 20 miles to the gallon. The vertical line segments cover the interquartile range of the price and the horizontal line segments cover the interquartile range of the mileage. The length of these lines correspond to the size of the box in a typical box plot. The plot shows that two-thirds of the cars range between \$4,200 to \$6,500 and 18 to 25 miles per gallon. The lower and upper lines (both vertical and horizontal) correspond to where the whiskers of a typical box plot end. Here the plot shows that almost all the cars are within 12 to 35 miles per gallon range but that many of the cars fall outside the \$3,500 to \$9,000 price range.

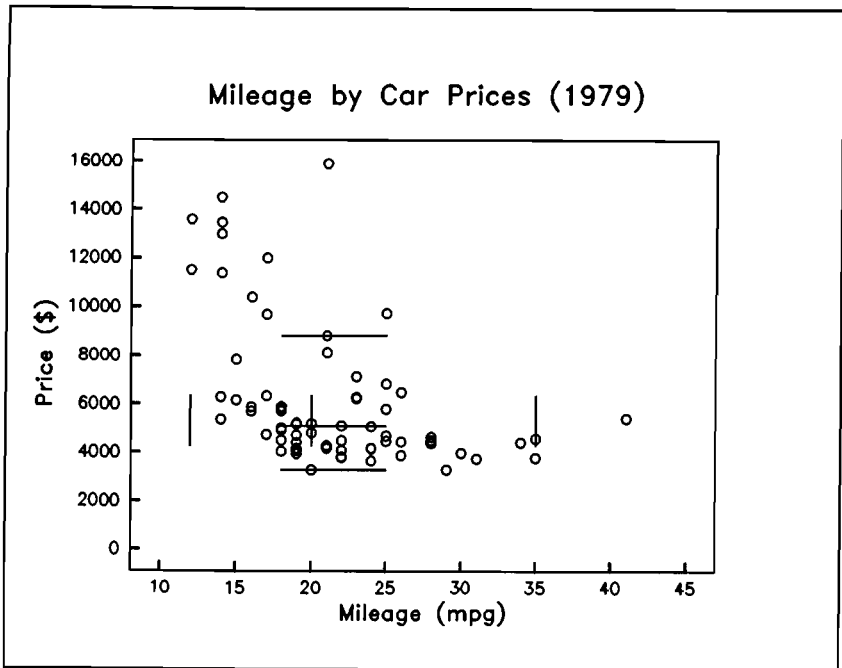


Figure 7: Scatterplot with a Boxplot

Relationship Between Three Variables

When the third variable on the scatterplot takes on only discrete values then the plotted points can be replaced by multiple letters or symbols. One advantage of the letters is that it is easy to remember the groups that they represent. The disadvantage is that they do not provide high visual discrimination with each other. Using symbols such as circles, squares, and triangles all filled and unfilled provide a better visual picture.

When the third variable on the scatterplot is on a continuous scale then the plotted points can be replaced by the size of a plotted symbol (with some conveniently chosen largest and smallest size). The diameter of the circle is drawn so it is proportional to its size. Figure 8 shows the weight of 74 car models against length. Miles-per-gallon is encoded in the size of the circle, with the large circle denoting high mileage. Generally speaking, the analyst can conclude that the longer and heavier cars get poor mileage while the shorter and lighter cars get good mileage.

Usually, an unfilled circle is the best symbol to use because they can tolerate substantial overlap and still maintain their individuality.

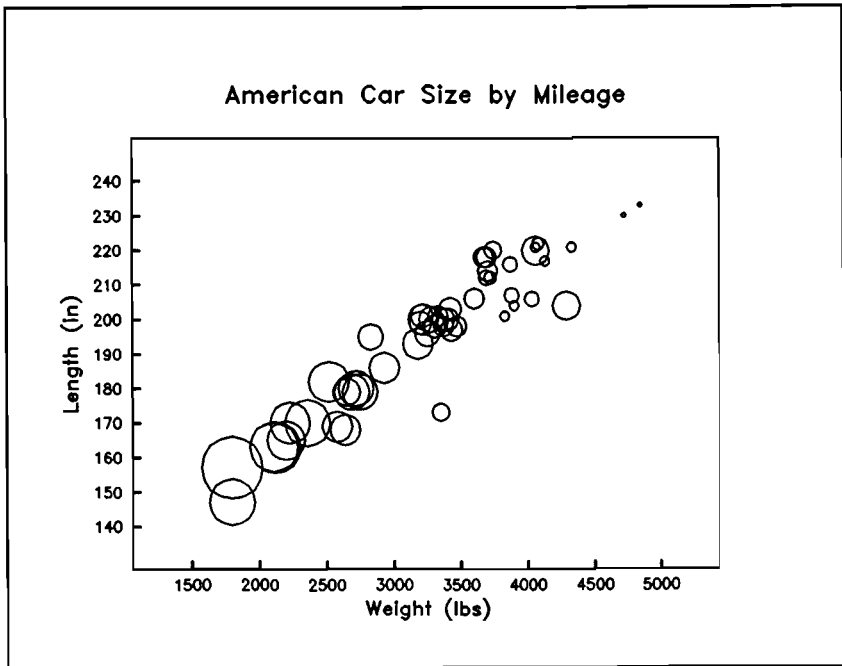


Figure 8: Scatterplot with a Third Variable

Relationship Between Four or More Variables

Science and technology would be far simpler if data always stayed in two or three dimensions since there are a wealth of plotting techniques for portraying data. Unfortunately, data can live in four, five or any number of dimensions. How is the analyst to graph them to understand the complex relationships? How does the analyst peer into four or five-dimensional space and see the configuration of points?

One simple method to look at multi-dimensional data is creating a *scatter plot matrix*. The idea behind the scatter plot matrix is to arrange the graphs so that every two variables in a matrix has shared scales. This means that the analyst can visually scan a row or column and see one variable graphed against all other variables. This makes it easy to track an interesting point or group of points from plot to plot. Figure 9 shows four variables plotted against each other. The tick labels correspond to the minimum and maximum values for that plot. The point of interest, the small car with the high mileage, is denoted with a box around it. The corresponding point is then highlighted in the other plots. It is found that this car is short, light, gets good mileage, and is inexpensive.

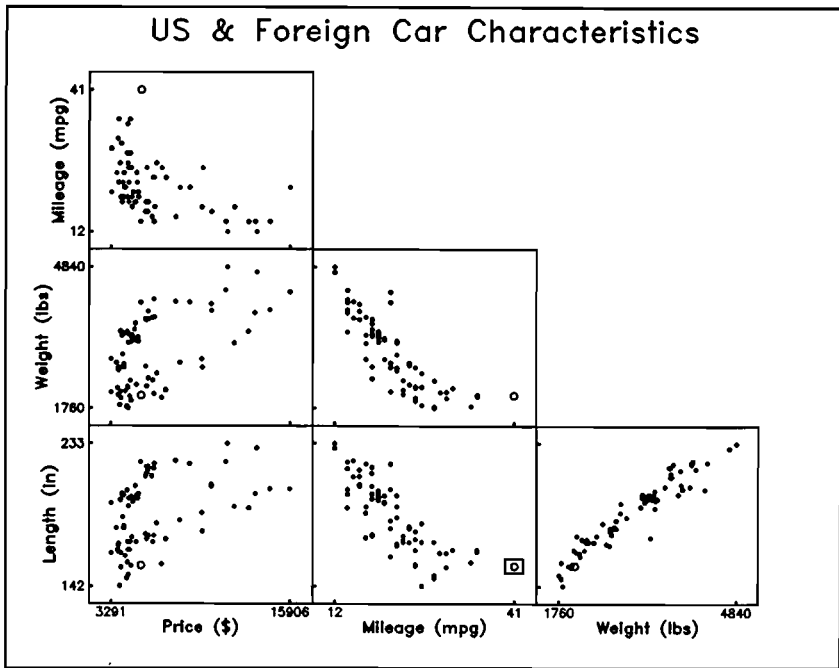


Figure 9: Pairwise Scatterplot

When it is important to identify relationships between observations instead of between variables then the *variables* can be portrayed simultaneously in the plotting symbol. One such method is called a *profile plot*. A profile symbol is created for each observation in the data set. The profile consists of all the variables plotted in the "favorable" direction across a line that represents the average. For example, "high" mileage would be considered favorable whereas "high" price would not. The price variable would need to be multiplied by -1 before being plotted.

The purpose of this technique is two-fold. First, the analyst should be interested in profiles that have most of their points above (below) the center line. Figure 10 shows profiles for 20 foreign cars from 1979. The profile for the Volkswagen Scirocco is interesting because all the points but one (Trunk Space) is below average.

Secondly, the analyst can look for pairs or groups of symbols with similar shapes. Alternatively, the analyst may want to identify observations that are very different from the rest. From Figure 10 it can be seen that the Volkswagen Scirocco and Dasher have similar profiles.

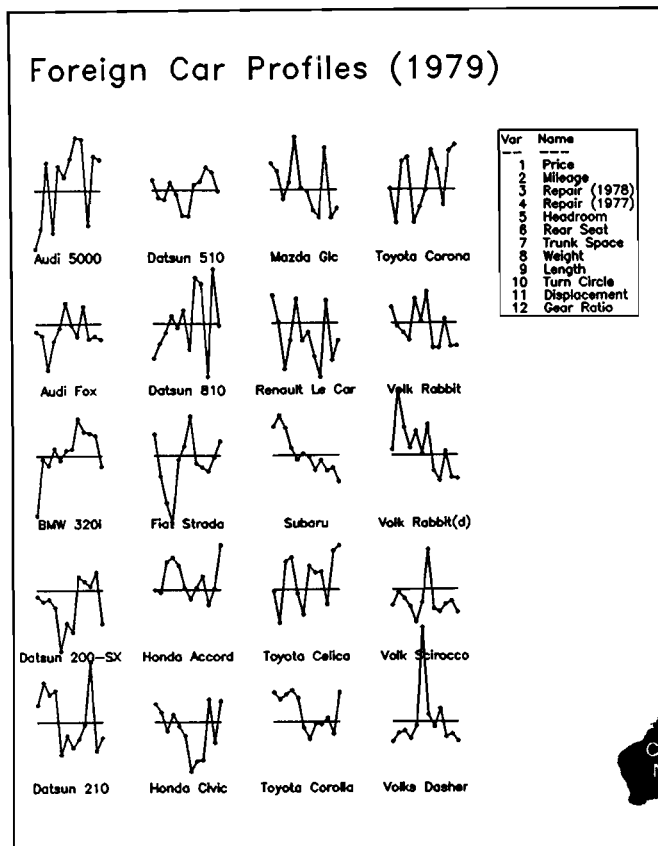


Figure 10: Profile Plot

Graphical Techniques in Regression

It is good practice to look closely at the raw data to get as much insight as possible before carrying out a multiple regression analysis. The objective is to discover any interesting relationships, unusual behavior, and exceptional points that can help guide the choice of models and fitting procedures. It is especially important for the analyst to become familiar with the raw data because the analyst is likely to fit the regression with a computer program that is blind to many anomalies in the data.

There are many questions to answer: Are there any outliers? Do variances appear constant? Do functional relationships look linear or curved? Would transformations help? Are there repeated values in some of the variables? Does the data cluster in interesting ways? Answers to some of these questions can be solved by using the scatter plot methods talked about in

the previous sections to see possible relationships between variables. Two other methods will be described here. The first will use residual plots to study how well the regression model fits a given set of data. The second, for situations in which a number of potential explanatory variables are available, is to guide the choice of a small and effective subset of variables for inclusion in the model.

Residuals

Residual analysis is of vital importance in any regression analysis. Residual plots are used for identifying any undetected tendencies in the data, as well as outliers and fluctuations in the variance of the dependent variable. In all residual plots the pattern that indicates an adequate model and well-behaved data is a horizontal band of points with constant vertical scatter.

Because the interpretation of the residual plot is very subjective a smooth curve, such as the one described in a previous section, could be drawn through the points that might show some systematic pattern. If the smooth curve is nearly horizontal and close to the baseline then random scatter of the residuals can be assumed. Figure 11 shows the residuals from a regression analysis that estimated the daily amount of evaporation from the soil. The residuals were computed from the 4 variables that composed the fitted regression line.

The plot of the residuals shows that regression fit works well for low values of soil evaporation but shows a curving trend at the higher values. This curvature suggests that a transformation of the explanatory variable(s) may be needed or that some of the variables could be dropped from the model.

Variable Selection

A meaningful regression analysis requires a high correlation of each of the explanatory variables x_1, x_2, \dots, x_p with the dependent variable y , while at the same time having a low correlation with each of the other explanatory variables. Explanatory variables that are very linearly dependent are termed collinear. This leads to unstable regression coefficients being computed and erroneous inferences about the model being made.

A graphical method called a *ridge trace* can be used to eliminate the variables that might be causing the multicollinearity. Estimators of the standardized regression coefficients are computed for different ridge parameters in the interval $(0,1)$. Each estimator is then plotted against various values of the ridge parameter.

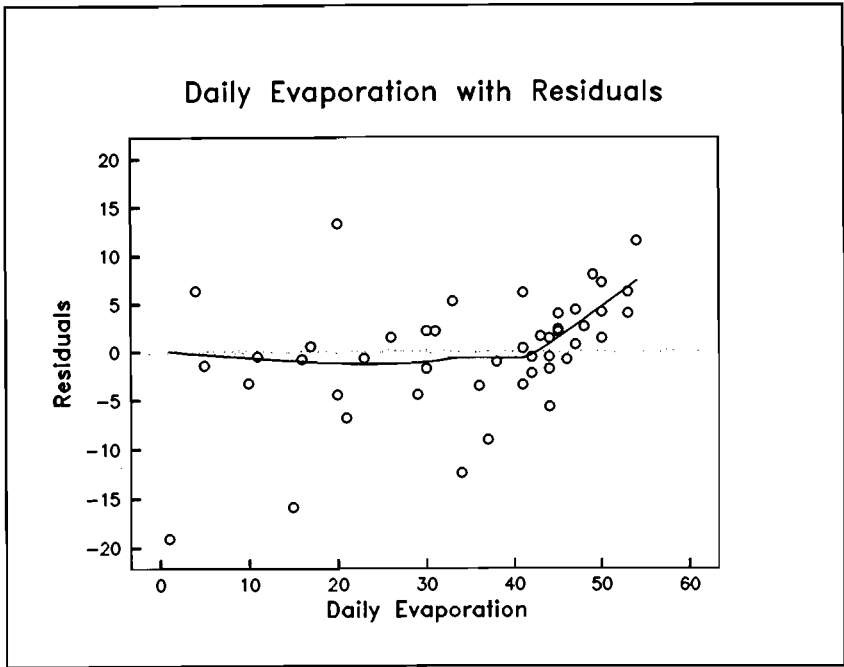


Figure 11: Residual Plot

The variable selection is done by examining the ridge traces on the graph. The rules for elimination are :

1. Eliminate variables whose coefficients are small (usually less than .2). Since the method is applied to standardized data, the magnitude of the various coefficients are directly comparable.
2. Eliminate variables with unstable coefficients that do not hold their predicting power, that is unstable coefficients that tend to zero.
3. Eliminate variables with unstable coefficients where the coefficient changes sign.

The variables remaining from the original set are used to form the regression equation.

Figure 12 shows 10 variables that were used to form the regression equation. Using the rules stated above, *wind*, *maxh*, *minat*, and *mazat* would be eliminated under rule 1, *minh* under rule 2, and *minst* under rule 3. The four remaining variables, *maxst*, *avat*, *avh*, and *avst* would compose the regression. This can be verified by doing a backward stepwise regression procedure.

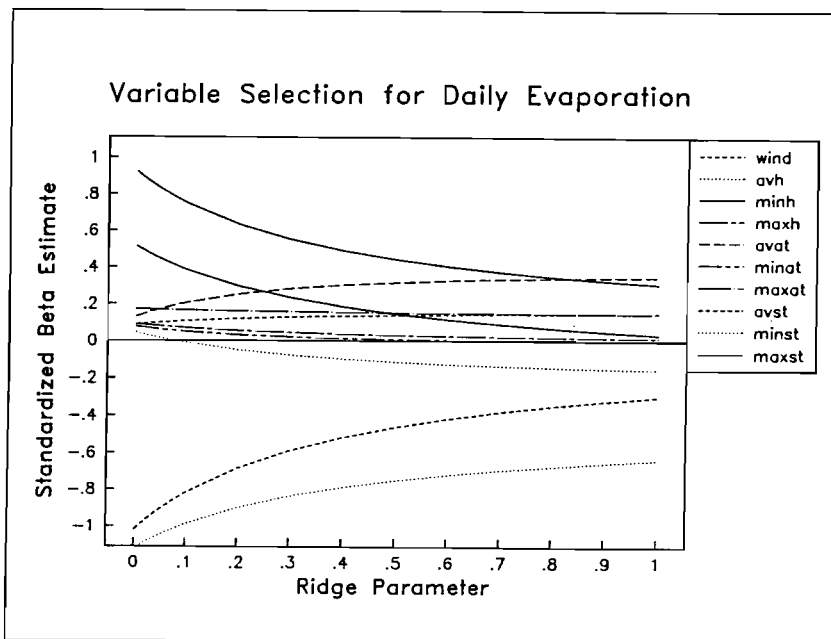


Figure 12: Ridge Trace

It should be noted that the variable selection procedure is a mixture of art and science, and should be performed with care and caution. It must be emphasized that variable selection should not be performed mechanically as an end in itself, but rather as an exploration into the structure of the data. The explorer should be guided by a combination of theory, intuition, and common sense.

Graphical Techniques in Analysis of Variance

The aim of analysis of variance is to determine whether the means of several populations differ from one another. The different populations are usually associated with different treatments which are carried out independently of one another on some experimental units. Two commonly asked questions are: Where is the difference between the treatments coming from? Is there a significant interaction present? These two questions can be answered graphically with the *pairwise comparison plot* and the *interaction plot*.

Pairwise Comparisons

It is often desirable to isolate the sources responsible for a significant treatment effect. There are a number of tests that are available in computer programs to help the analyst

determine the sources. These tests include Newman-Kuels test, Duncan's test, Tukey's test, and Scheffe's test. Each of these tests has their advantages and disadvantages and the choice of which test to use will be left to the analyst.

A graphical method for determining whether a significant difference is present between treatments is to plot the ordered means on a plot. Next, compute the critical range between treatments from one of the tests listed above and draw a perpendicular line through each point such that the center of each line goes through the mean. Last, draw a horizontal line to the right border of the plot from the top of each interval. Any pair of treatments for which the intervals are *not* joined by a common horizontal line, differ significantly.

For example, the analyst has completed an experiment of how the use of 4 different drugs controls the increase in systolic blood pressure. The ANOVA table shows a significant difference between the drugs, but where? Figure 13 shows that Drug 3 and 4 is significantly different from Drug 1 and 2 because the horizontal lines from 3 and 4 don't cross the vertical lines of 1 and 2.

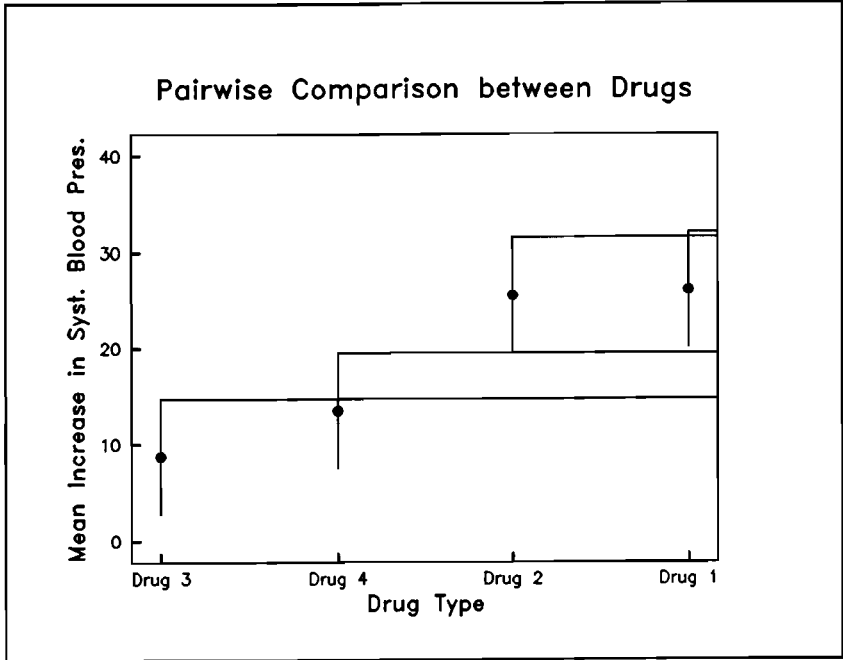


Figure 13: Pairwise Comparison Plot

Interactions

When the analyst finds that an interaction is present (by looking at the ANOVA table) it is usually a good idea to plot the results of the experiment. The interaction plot has the dependent variable along the y-axis, one independent variable along the x-axis, and a curve drawn for each level of the second independent variable. The shape or form of the interaction will become apparent. An interaction will be revealed by nonparallel or crossing curves for the variable plotted within the body of the plot.

Figure 14 shows a significant interaction effect because the difference between *Seed 1* and *Seed 2* at the medium level of fertilizer is not the same as at the other two levels. The two lines would be parallel if the interaction wasn't significant.

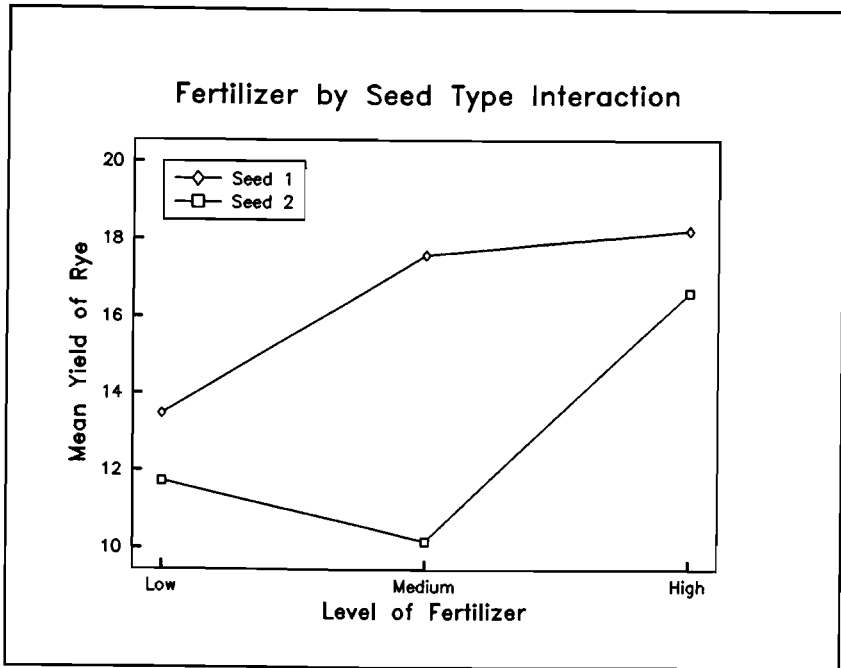


Figure 14: Interaction Plot

Graphical Techniques in Time Series

A time series is a special case of the broader dependent-independent variable category, where in this case, time is the independent variable. One important property of most time series is that for each time point of the data there is only a single value of the dependent variable; there are no repeat measurements. Furthermore, most time series are measured at

equally spaced or nearly equally-spaced points in time.

There are a number of ways to graph a time series. A line graph is appropriate when the time series is smooth or when the interest is in the shape of the series instead of individual values. A vertical line graph is appropriate when it is important to see individual values, when the analyst needs to see short-term fluctuations, and when the time series has a large number of values.

Seasonal Decomposition

A *seasonal decomposition plot* uses two line graphs and two vertical line graphs to show the breakdown of the time series into components. The breakdown consists of trend, seasonal, and irregular components. The sum of these three components is equal to the original series.

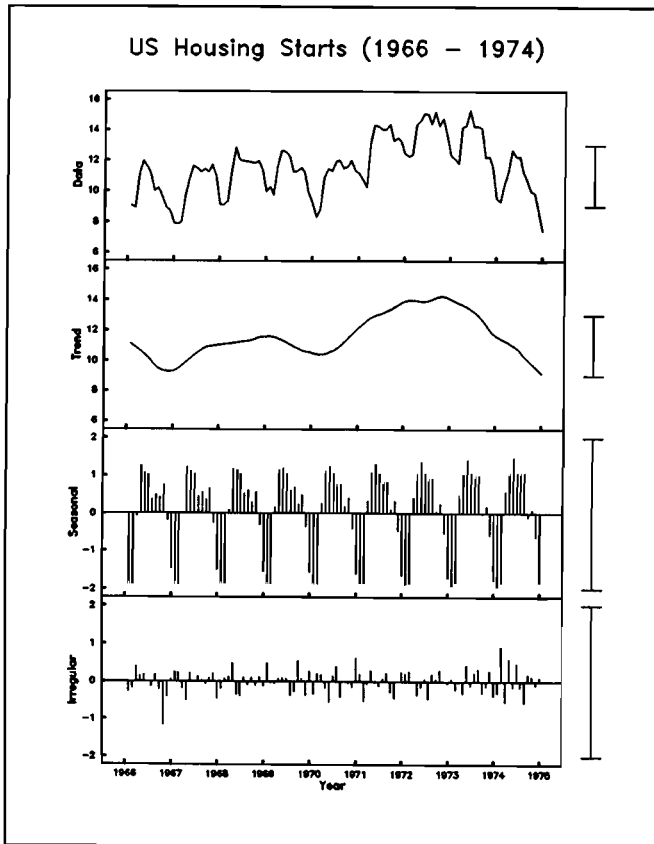


Figure 15: Seasonal Decomposition Plot

and irregular components. The sum of these three components is equal to the original series.

The decomposition is robust to outliers, helps to choose a transformation for the data that assists in the decomposition, and can be adjusted for calendar effects.

The line graph is used to plot the full time series and trend components since it is only important to see the shape. Figure 15 shows the number of housing starts from 1966 to 1974. The trend is reasonably constant until a sharp drop starts in 1973. The vertical line graph is used to plot the seasonal and irregular components since it is important to assess behavior over short periods of time. Figure 15 shows that the seasonal fluctuations are reasonably constant from year to year while the irregular component (or noise) is a random scatter, as it should be.

Seasonal Subseries

A *seasonal subseries plot* is used to study the behavior of a seasonal time series or the seasonal component from the *seasonal decomposition plot*. This plot is most often used to look at the behavior of a time series for each month graphed for successive years. However, it could be used for any two spans of time (day and week, week and month, etc.).

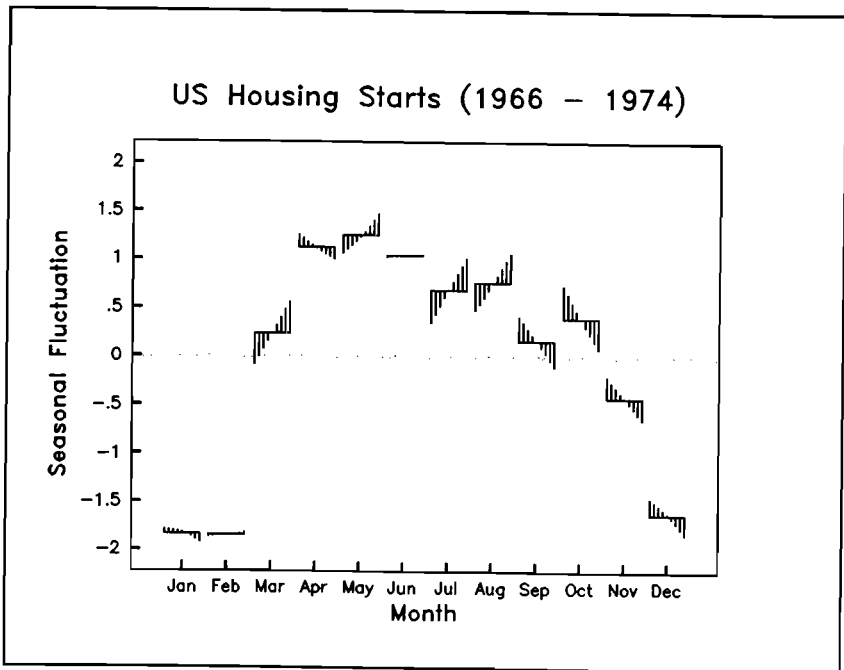


Figure 16: Seasonal Subseries Plot

For each monthly series, the mean of the values is portrayed by a horizontal line. Figure

16 shows that not many housing starts are done in the January, February, and December months as compared to other months of the year. The yearly values of each of the monthly subseries are portrayed by the ends of the vertical lines. From Figure 16 the months of March, May, July, and August show an increase in housing starts from 1966 to 1974. The months January, April, and September thru December shows a decrease in housing starts. Therefore, this graph allows an assessment of both the overall monthly pattern as well as the overall yearly pattern of the data.

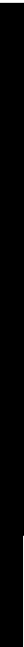
Summary

The intent of this paper was to show graphical techniques for the most commonly occurring types of data in all areas of science and technology. For that reason, each of the techniques presented could not be explored in detail and many specialized methods had to be omitted. The reader should refer to the references listed for a more detailed discussion.

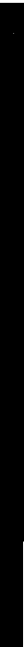
The graphs in this paper were generated using the Statit and Graft software packages from Graphicus on an HP9000/320 computer running HP-UX. These packages are currently available on the Hewlett Packard technical computers running HP-UX and RTE-A. Statit, a general purpose statistics package, was used to prepare the data, perform the computations (some of which were very computer intensive), and send the plotting commands through an interface to Graft. Graft, a general purpose graphing package, took the commands and output the graphs on a LaserJet Plus printer. Each of the graphs were generated using simple commands available in Statit. These commands have many options that allow the graph to be customized to the specifications of the analyst.

REFERENCES

- Beckett, S. and Gould, W. (1987). A Note on Rangefinder Box Plots. *The American Statistician* 41(2): 149.
- Chambers, J.M., Cleveland, W.S., Kleiner, B. and Tukey, P.A. (1983). *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.
- Chatterjee, S. and Price, B. (1977). *Regression Analysis by Example*. New York, NY: Wiley. [Chapter 8].
- Cleveland, W.S. and McGill, R. (1984). The Many Faces of a Scatterplot. *Journal of the American Statistical Association* 79: 807-822.
- Cleveland, W.S. (1985). *The Elements of Graphing Data*. Monterey, CA : Wadsworth
- Du Toit, S.H.C., Steyn, A.G.W., and Stumpf, R.H. (1986). *Graphical Exploratory Data Analysis*. New York, NY: Springer-Verlag.







HP1000 A-SERIES CRASH DUMP ANALYZER

Russ Scadina
Hewlett-Packard Co.
1266 Kifer Road
Sunnyvale, CA 94086

Introduction

System crashes (the CPU has halted or is looping in a meaningless fashion) have long haunted the customer and the manufacturer of the system. There is so much data to analyze; so much in-depth knowledge of the operating system required; so much understanding of assembly language needed. These prerequisites for analysis and troubleshooting dictate that a system expert be consulted. The question is which tools may be utilized prior to calling the expert in? The crash dump analyzer is one such tool which automatically and comprehensively diagnoses the state of the crashed system in a fast and thorough manner.

History

Historically, the analysis of an HP RTE system that had crashed was traumatic at best. Typically, hardware was swapped first starting with the CPU. After this effort failed to fix 'the problem', the next step was to perhaps minimize system activity to see if 'that' worked or, at least, decreased the frequency of crashes. Since many customers run their system in a production environment, this approach was costly or impossible. Perhaps then a system expert would sit down in front of the 'front panel' and 'look' at some registers and memory locations. Unless it was his lucky day, he would find nothing wrong and provide the old 'try something different (regen?) and see if it goes away' trick. Eventually, assuming this was a widespread problem, someone somewhere tripped across a data corruption or some such thing and the race was on to solve the 'elusive bug'. The culprit was detected by creating patches or using logic analyzers to trace computer activity leading up to the condition that caused the crash.

About a decade ago an engineer at HP decided that if a crash dump (moving memory contents to a medium) could be performed and restored onto a healthy system that a system expert could analyze the system that crashed while allowing the customer to restart and use his system. The tool that evolved was called CMM4 (later CMM6 and then CMM4). This program is still in use and has utility as an on-line memory access or crashed system access tool. What it doesn't do is automatically analyze the system - that is left up to the user to do step by step (a little more efficiently than via the front panel).

What was still needed was a tool to 'look' automatically at memory and determine what state the operating system and its data structures were

in. This is where the HP1000 RTE-A Series Crash Dump Analyzer (called ACDA) came in. It automatically determines the integrity of the data structures in the system (as a result of or leading up to the crash!!).

What is still needed is a tool that might point to the culprit that caused the crash. This analysis is still left up to the system expert but ACDA is a definite step in the right direction.

The rest of this paper describes ACDA.

ACDA

ACDA ties several troubleshooting techniques together to assist in the early stages of analysis of the cause of a crashed system. It performs:

- a systematic search of system lists and data structures to determine their validity.
- a reconstruction of the structure of System Available Memory (SAM) to determine if SAM corruption has occurred.
- a reconstruction of the state of every program in the system so the user may quickly survey (WHZAT style) their state at the time of the crash.

ACDA takes as input the SNAP file and crash dump from the VCP to analyze the entire system and then provides easy access to all of the dumped physical memory. ACDA also provides tutorial assistance and ease of troubleshooting by annotating some data structures and displaying the source of the data, e.g., the label of the head of a list.

The major data structures that ACDA analyzes for gaps, overlap and inconsistencies are the memory descriptors, program state lists, IO tables and SAM. The memory descriptors are analyzed to ensure program partitions are valid and program state lists are traversed to ensure they're intact. The DVTs, IFTs and IO control blocks are traversed to ensure proper linkage between them and SAM. SAM is completely reconstructed from the DVTs, Class table, session table, runstrings and free memory to ensure SAM is consistent.

After analysis, ACDA provides easy access to any map or mapped entity. This means that the user may easily view the system map, a user program (including code or data), a mapped driver or operating system module, SAM or any other physical memory. Memory displays show the assembly, octal, decimal and ASCII equivalent of each memory word. When displaying ID segments, DVTs and IFTs, the contents are displayed with annotations to explain the bit patterns of each word so the user need not refer to a manual to extract bit fields from the table.

ACDA also provides a WHZAT type feature to display the state of all programs in the system for rapid assessment of which programs were running at the time of the crash. A list of system module entry points may also be displayed to determine (without a generation listing) where each module resides.

Keep in mind that ACDA does not uncover the offending code that led to the failure. What ACDA does do is: 1) detect the error condition 'after it has occurred' 2) allow the user to key on a failure, e.g., a modified table location or a list that is corrupted, so that a logic analyzer or troubleshooting traps may be used to catch the corruption in action during a subsequent system failure 3) rapidly assess if anything seems amiss in the system and 4) document patterns of failures to determine if multiple crashes have the same scenario, e.g., a corrupted SAM location.

The following represents a typical display when ACDA is first executed. The sizes of various system resources are shown, for example the size of SAM. If these or any of the analysis routines, e.g., Analyzing Memory Descriptors, uncover an error, a message is displayed and the user may then use various ACDA commands to determine the nature of the problem.

```
RTE-A Crash Dump Analyzer 01/19/87
Your Snap Filename is /DUMPS/SNAP.SNP
Your Crashed System Filename is /DUMPS/SYS.DUMP
Your SYSTEM (crash dump) file is for an:
  RTE-A System - rev code 2540 $OPSY = -53
The system physical memory size dumped was 3072 k words
There are 32763 words of SAM ($SAM#) starting at page 58 ($SAMP)
The Driver/OS partition starts at logical address 34000
  and consumes 3 logical pages
There are 105 ID Segments ($ID#)
The ID Segments start at addr 63220 ($IDA) and 46 words long ($IDSZ)
The maximum LU number is 131 ($LUT#)
There are 79 DVTs ($DVT#)
The DVTs start at addr 42000 ($DVTA) and are 25 words long ($DVSZ)
There are 11 IFTs ($IFT#)
The IFTs start at addr 60650 ($IFTA) and are 9 words long ($IFSZ)
There are 100 class numbers ($CLTA,I)
Analyzing Memory Descriptors
Analyzing System Available Memory
  SAM analysis error encountered !!!
Analyzing Program Lists
Command (/e to terminate)?
```

Notice that an error in analyzing SAM has been detected. The user would then do an Analyze command which will highlight the exact failure as part of displaying various data structures and lists. The following is part of a sample display of the Analyze command:

Analyzing Memory Descriptors
 First partition page (octal) =173

Traversing MD Adjacency list from \$MEM

	wd0	wd1	wd2	wd3	wd4	wd5	wd6
		start	id/ prev	stat	prio/ next	prev adj	next adj
Addr	#pgs	page	free	stat	free	ptr	ptr
75417	24	173	67402	100000	62	0	74607
.
75676	3	5775	175104	100000	75624	75660	0

Traversing Free MD list from \$FREM

	wd0	wd1	wd2	wd3	wd4	wd5	wd6
		start	bt15=1 prev	stat	next	prev adj	next adj
Addr	#pgs	page	free	stat	free	ptr	ptr
74742	21	651	174616	0	75577	74562	74751
.
74616	10	551	175570	100000	74742	75320	74677

Analyzing System Available Memory

Traversing SAM free block list from SAM address 1

Address	Length	Next Block
3	4	41
41	13	437
.	.	.
53725	1804	57610
57610	8310	77777

Traversing SAM runstrings from SAM address 2

	WD1	WD2	WD3	WD4...
SAM	next	ID seg		
Address	link	addr	#chars	string
31	0	66332	10	RU,PROG

Scanning IO request blocks off DVT(2)s and Spool Nodes off DVT(25)s
 Scanning DVT 79

Scanning DVT 1
SAM analysis error encountered !!!

Scanning Session ID Tables for UDSPs off word 19
Scanning Session Table 1

Scanning Session Table 9

Scanning Class Table (word 1) for class completion blocks
Scanning Class Table for Class # 1

Scanning Class Table for Class # 100

SAM is allocated as follows:

Addr	Size	Allocated by/for
3	4	Free Memory Block
7	26	Run String for ID# 36
41	19	Free Memory Block
56	16	IO Request off DVT# 1

SAM overlap detected !!!!

76	167	Class Completion of Class Number 71
.	.	.
.	.	.

Analyzing Program Lists

\$TM (Time Suspend) List traversal: No ID segments in the list

\$CL (Class IO Suspend) List traversal:

The ID segments in the list are -

ID#	5
ID#	38

Notice the flow of activity: the Memory Descriptors are scanned while displaying their contents and significance; next, the data structures which use SAM, namely free SAM blocks, runstrings, IO requests, spool nodes, session blocks and class table are analyzed to create an image of SAM to determine its integrity. In this case scanning DVT 1 indicates a failure produced in SAM. The subsequent SAM allocation display shows that the IO block off of DVT 1 overlaps a free memory block which starts at address 41 and continues to address 64. This discovery would 'point' the system expert towards surveillance of DVT 1 and its associated driver.

The system expert may now want to look at the DVT to determine its contents (state). ACDA, on command, will display the DVT, with annotation, and its parameter and extension areas. The following is a sample:

Description of DVT 1

wd	addr	contents	description
1	42000	177777	DVT not linked off any list
2	42001	0	program priority queuing of IO requests No Requests queued off this DVT
3	42002	42002	No DVT in the Node list is busy Circular node list points to DVT 1
.	.	.	.
21	42024	2000	There are(is) 2 driver parameters There are(is) 0 extension words
22	42025	0	Address of extension area
23	42026	0	Driver is not partitioned
24	42027	0	No IO requests queued Default Language number in use is 0
25	42030	0	There are no spool nodes off this DVT

DVT Parameter Area

addr	contents	decimal	ascii
42031	1	1	
42032	2	2	

Perhaps the system expert wants to explore further by displaying the contents of an IFT. This would be displayed as follows:

Description of IFT 1

wd	addr	contents	description
1	60650	177777	Timeout list linkage
2	60651	0	Timeout clock
3	60652	100000	FIFO queuing of DVTs No DVT queued off this IFT
4	60653	40000	Interface driver entry address
5	60654	42000	This IFT is currently associated with DVT 1
6	60655	24050	Status: Interface is available Interface Type is 50 (Parallel) Interface Select Code is 50
.	.	.	.
9	60660	0	Map set NOT been allocated for the interface Map Set Allocated to the IF driver is 0

IFT Extension Area

addr	contents	decimal	ascii
60661	0	0	
.	.	.	.
60671	0	0	

The system expert may want to determine the state of a program. A WHZAT type display may be seen as follows:

ID#	Address	Occupant	Session	State
1	63220	D.RTR/	0 SYSTEM	dormant
19	64714	SETUP/	32 SESSIN	IO suspended
23	65204	PROG1/	0 SYSTEM	class suspended on CL# 76
58	70316	CI /	38 SEESOT	wait suspended

Then a further display of an ID segment contents:

Description of ID Segment # 1			
wd	addr	contents	description
1	63220	66176	list linkage
7	63226	1	program priority
8	63227	2007	primary entry point
9	63230	22357	point of suspension
10	63231	177627	A-reg at point of suspension
11	63232	63221	B-reg at point of suspension
16	63237	100000	program is in memory run-string is NOT in memory break is NOT pending on this program program does NOT access system common ID segment NOT deallocated when prog terminates program has NOT locked DS resources program is NOT memory locked by an EXEC 22 program status is dormant
46	63275	0	Going privileged nesting counter

ACDA has a fair amount of capabilities not discussed here but one last one is noteworthy - the ability to list memory logically. For example, if the user wants to access SAM, the S 246 3 command would display SAM location 246 for three locations as follows:

addr	contents	macro	decimal	ascii
246	20455	XOR 455	8493	!-
247	20526	XOR 526	8534	!V
250	41422	ADA 1422	17170	C

A mapped driver, operating system module or a user map may also be displayed logically while EMA or any physical memory not normally mapped (such as a PTE) may be viewed physically.

Conclusion

ACDA has been widely used to pinpoint and categorize system failures by Hewlett Packard's DSD/AMSO technical marketing, R&D labs and field. In approximately one year of existence, ACDA has saved an extraordinary amount of hours in analyzing systems; has resolved many system problems and has determined that many system 'hangs' were programming errors, e.g., resource contention lockups. Users of ACDA have found it easy to use while learning about system internals. As mentioned before, the next step is to create a tool to diagnose the reason for a system failure. This would nearly automate the troubleshooting of crashed systems.

Implementing High-Level Control Structures Using MACRO/1000 Macros

Nick Seidenman
McDonnell Douglas Payment Systems Company
2092 Gaither Road
Rockville, Maryland 20850

ABSTRACT

The intent of this paper is to show how control structures such as IF-THEN-ELSE and WHILE loops are translated from high-level languages like C and FORTRAN to machine language. To demonstrate these translations, MACRO/1000 macros will be constructed which provide several control structures for HP-1000 assembly language programs.

1. INTRODUCTION

(This section may be skipped if the reader is already familiar with macro processing.)

It is safe to say that the great majority of programs written for the HP-1000 are written in Ftn7x (HP FORTRAN-77 with extensions). A significant, albeit smaller share, however, are written in assembly language. Assembly languages in general allow the programmer to have complete control over the instructions used to accomplish a given task. This advantage is cited by many as the primary reason for choosing assembly over a high-level language. Greater control implies less "fat" due to unneeded code which might have otherwise been included by a compiler.

On the other hand, the number of lines of assembly source code will almost always exceed the amount of higher-level source code, making assembly harder to read and harder still to maintain. Since assembly instructions are simpler in nature than, say, FORTRAN instructions, it takes more of them to accomplish the same task. In many cases, groups of instructions in assembly are repeated with minor changes to suit the particular instantiation. Such repetition can be found in the .ENTR calling sequence (the standardized calling sequence used to pass arguments and a return address to a subroutine).

For example, the subroutine call

```
CALL SUB1 (ARG1, ARG2, 3)
```

in FORTRAN would become

```
JSB SUB1 ; Jump Subroutine Instruction.  
DEF *+4 ; Ret Addr (Here + 4 words).  
DEF ARG1 ; Address of ARG1.  
DEF ARG2 ; Address of ARG2.  
DEF %D1 ; Address of constant.
```

Such repetition opens the door for mistakes which would rarely, if ever, occur in a high-level language. Note that in the CALL statement above no return address is supplied; the compiler computes the symbolic return address and emits the proper source statements. Miscalculating the return address is not an uncommon mistake when programming in assembly. Furthermore, if another argument were added to the CALL statement, the compiler would still compute the correct return address. If written in assembly, however, the programmer would need to remember to change the return address by hand. Forgetting to do so is also a common error. With these disadvantages in mind, the concept of macro processing was introduced to assemblers.

Simply stated, a macro is a character string, which when processed by a macro processor will be replaced with another string. The idea is to use a relatively short string (which we will henceforth call a label) as the macro, which will be replaced by a longer string. Alternatively, a macro may be used to represent a string which will appear many times in a body of text (such as source code or a document), but may be subject to change at a later time. By changing the definition of the macro, the string can effectively be changed throughout the text. We call the replacement of a macro label with its defined string a macro expansion or, more simply, an expansion.

To further illustrate the use of macros, suppose we want to use the same value each time we declare a storage area in memory. We could just use this value in each BSS (static storage allocation) pseudo-op. But what if we later decide that we need more storage in each of these areas? We could go through the code with a text editor, changing each occurrence of the BSS value. This works fine as long as we change every occurrence. This is not only time-consuming, but it is also very easy to miss one.

Suppose, instead, we define a macro with the label ARRAYSIZE and set its value to "100". Then in each BSS statement we declare the array to be ARRAYSIZE words long. The macro processor will expand all occurrences of ARRAYSIZE to 100, performing all the tedious work for us! Now, if we need to change the size of the arrays, we merely change the defined value of ARRAYSIZE - once. We are guaranteed that all of the array declarations will be the correct size when the code is processed by the macro processor.

A macro expansion may result in the generation of several lines of text. Moreover, macros may take arguments that will be included in the expanded text. As an example, consider the case in which we define a macro to replace the .ENTR calling sequence described above with a single line CALL macro. The code written by the programmer might look something like this:

```
CALL SUB1, ARG1, ARG2, =D3
```

(Note the constant definition, =D3.) The macro would be defined to expand in such a way that its first argument would be used in the JSB instruction, and the remaining arguments would be used in the DEF pseudo-ops as shown above. To reiterate, the expanded macro would appear as follows:


```
JSB  SUB1
DEF  **4
DEF  ARG1
DEF  ARG2
DEF  =D3
```

(The assembler will take care of allocating and initializing the data area for the constant.)

There are several advantages to using a macro this way. First, the repetition is eliminated since the CALL macro may be invoked many times. Second, fewer lines of code need be written by the programmer, thus saving valuable development time. Finally, the details of how SUB1 is invoked are hidden from the programmer. If this code were to be rewritten for a different computer, only the macro would need to be changed (assuming of course that a macro-assembler existed on the target machine). This directly contradicts the conventional wisdom that all assembly programs are non-transportable!

Before moving on to the next section it should be pointed out that the syntax conventions used thus far are not by any means generalized. Whereas the arguments supplied to the hitherto fictitious CALL macro are delimited by commas, other macro processors use syntax which is quite different. The C macro preprocessor, for example, uses notation which allows a macro with arguments to look just like any other function (the functor is followed by the arguments enclosed in parentheses). The syntax used is, however, very similar to that used by MACRO/1000.

(This next section may be skipped if the reader is already familiar with MACRO/1000 macros.)

2. MACRO/1000 Macros: The Basics

Before we can begin developing the high-level control structures promised above, we must first become acquainted with the elements of the MACRO/1000 language itself. There are basically two types of macros in MACRO/1000; Assembly Time Variables or ATV's, and procedure macros.

2.1 Assembly Time Variables (ATV)

ATV's are used to symbolically define constants which will be used during the assembly pass(es). ATV's consist of a label and a value to which the label will refer. Labels can be up to 16 characters in length must begin with an ampersand (&) character. Note that the label "&" is illegal. There are a number of predefined ATV's called system ATV's which begin with "&.". These can be used to provide global information to user macros and will be discussed in greater detail below.

ATV's may be either local or global. The differences between local and global visibility will be better understood once we have looked at procedure macros. For now we need to be aware of this only because of the different pseudo-ops used to declare ATV's. Local ATV's are declared with the ILOCAL or CLOCAL pseudo-ops; and global ATV's are declared with the IGLOBAL or CGLOBAL pseudo-ops.

ATV's come in two flavors; integer and character. The distinction between the two is made when they are first declared. To declare an integer ATV, the IGLOBAL or ILOCAL pseudo-op is used. For example, the integer ATV "errCount" is defined by the statement

```
&errCount IGLOBAL 0
```

All subsequent references to &errCount will expand to the current integer value of &errCount. Note that the syntax is exactly the same when declaring a local ATV.

The preceding example demonstrated the definition of a scalar or single-value ATV. ATV's may also be defined as vectors or arrays of values. Thus

```
&Cstack[20] IGLOBAL 0,0,0,0,0,0,0,0,0,0,0,0,[10]0
```

declares an array ATV with twenty elements, and initializes the array elements to zero. This example also demonstrates the two methods by which ATV's may be initialized. Note the use of square braces toward the right side of the declaration. This notation ([10]0) will initialize the next 10 elements of the array to zero.

Character ATV's are similarly defined using the CLOCAL and CGLOBAL pseudo-ops. All character strings are treated as arrays of characters, thus the declaration of a character ATV must be that of an array. The following example defines a character ATV 20 characters in length and initializes it to the string "a character ATV".

```
&identity[1,20] CLOCAL 'a character ATV'
```

The thing to note in this example is the second argument in the label field. The first argument indicates how many ATV's are declared. The second states the length of each element in the array. The above example, then, defines a Character ATV called &identity with one element twenty characters in length.

Some final points before we move on to procedure macros; integer ATV declarations may also have a size argument but the only value which makes sense is 1. Character ATV declarations may specify 0 for the number of elements. This is the same as specifying 1. Specifying 0 for the number of integer ATV elements will be rejected by the MACRO/1000 macro processor.

2.2 Procedure Macros

Procedure macros assign labels and pass arguments to source code fragments. The procedure macros are then replaced in the source code by these fragments before being processed by the assembler. Procedure macros consist of a name statement and a body. The name statement declares a label field name, the name by which the macro is referenced, and an optional list of arguments to be passed to the macro. The body consists of the macros, pseudo-ops, machine instructions and ATV's which will be used when the macro is referenced.

In an earlier section we introduced the CALL macro. We shall now look at how such a macro is defined. It will be assumed, for the sake of simplicity, that the CALL macro always takes three arguments. We begin by telling MACRO/1000 that we wish to declare a macro. This is done using the MACRO pseudo-op in the opcode field of the source code. The next line is the name statement. This will be defined with formal arguments &label for the label field, &arg1, &arg2, &arg3 for the arguments, &sub for the name of the subroutine to JSB to, and CALL for the name field. The resulting name statement looks like this:

```
MACRO
&label    CALL &sub,&arg1,&arg2,&arg3
```

We now build the body of the macro. First comes the JSB instruction followed by the return address.

```
&label    JSB  &sub
          DEF  **4
```

When the macro is invoked, the &label and &sub ATV's will be replaced by the actual values supplied. We next fill in the DEF's for the arguments.

```
DEF  &arg1
DEF  &arg2
DEF  &arg3
```

The macro definition is completed with the ENDMAC pseudo-op. The complete macro definition looks like this:

```
MACRO
&label    CALL &sub,&arg1,&arg2,&arg3
          DEF  **4
          DEF  &arg1
          DEF  &arg2
          DEF  &arg3
          ENDMAC
```

Notice that no ATV declarations are necessary for the formal macro arguments. These are managed by the macro processor. When the macro is invoked in a source statement, the actual arguments supplied in the source line will replace the formal arguments represented by the ATV's.

2.3 CONDITIONAL ASSEMBLY

One will notice right away that this macro is somewhat inflexible in that it always takes three arguments. It would be more useful if the macro could accept any number of arguments and produce the appropriate source statements. The number of DEF's to be generated will be based upon a certain condition, namely the number of arguments present in the macro invocation at assembly-time. This introduces the concept of conditional assembly. MACRO/1000 provides several pseudo-ops for conditional assembly which can cause inclusion or exclusion of source lines, or repetition of source lines.

The first conditional assembly pseudo-op we will use is called AIF. AIF and its companion AELSEIF take arguments which comprise relational expressions. The terms in each of these expressions is evaluated for a zero or non-zero value. If the value is zero, the relation is false. Otherwise, the relation is true. Below is a brief example of how the AIF pseudo-op works.

```

      AIF  4-4
          LDA  =D0
          JMP  NEXT
&LAST  AELSEIF &LAST=' '
          LDA  =D1
          CSET 'ONE'
      AELSE
          STA  NEXT
      AENDIF
```

The preceding example demonstrates several ways in which assembly may be altered by the AIF structure. Note that the results may be emission of code and/or change or declaration of macros. We will now use the AIF structure in the CALL macro such that it will accept a variable number of arguments.

Before we can change our CALL macro, we need to have a way to determine the number of DEF statements and the return address before we begin expanding the macro into source statements. We can do this by using one of the aforementioned system ATVs called &PCOUNT. Upon invocation of a macro, &PCOUNT is set to the number of arguments actually passed to the macro.

The new CALL macro definition begins much the same as the old. The only difference is that five formal arguments are declared. This will be the maximum number of arguments that will ever be accepted by the macro.

```

      MACRO
&label CALL &sub, &arg1, &arg2, &arg3, &arg4, &arg5
&label JSB &sub
```

The next thing we need to do is determine the return address, or at least the symbolic expression for the return address. We do this by using the `&.PCOUNT` system ATV. We know that the return address will be equal to the address of the word following the JSB instruction (where we will store the return address) plus the number of arguments in the call plus one. Thus, the next line in the macro becomes

```
DEF  '*+'&.PCOUNT
```

Next we use an AIF block of the form

```
AIF  &.PCOUNT>{n}  
    DEF  &arg{n+1}  
AENDIF
```

where `n` is a number from 0 through 4, for each formal argument declared in the name statement. The macro body for this portion of the macro looks like this:

```
AIF  &.PCOUNT>1  
    DEF  &arg1  
AENDIF  
AIF  &.PCOUNT>2  
    DEF  &arg2  
AENDIF  
AIF  &.PCOUNT>3  
    DEF  &arg3  
AENDIF  
AIF  &.PCOUNT>4  
    DEF  &arg4  
AENDIF  
AIF  &.PCOUNT>5  
    DEF  &arg5  
AENDIF
```

We finish the definition with the ENDMAC pseudo-op. The complete CALL macro definition looks like this:

```
MACRO
&label    CALL &sub, &arg1, &arg2, &arg3, &arg4, &arg5
&label    JSB  &sub
          DEF  '*+'&.PCOUNT
          AIF  &.PCOUNT>0
              DEF  &arg1
          AENDIF
          AIF  &.PCOUNT>1
              DEF  &arg2
          AENDIF
          AIF  &.PCOUNT>2
              DEF  &arg3
          AENDIF
          AIF  &.PCOUNT>3
              DEF  &arg4
          AENDIF
          AIF  &.PCOUNT>4
              DEF  &arg5
          AENDIF
ENDMAC
```

Now if we invoke the macro with anywhere from zero to five arguments, the proper code will be generated for the given number of arguments. Take, for example, the statement

```
STEP1     CALL EXEC, READ, INLU, INBUF, =D-80
```

This will expand into the following code:

```
STEP1     JSB  EXEC
          DEF  *+5
          DEF  READ
          DEF  INLU
          DEF  INBUF
          DEF  =D-80
```

If we were to decide that the macro should handle more than five arguments, we would simply add the appropriate number of formal arguments to the name statement, and add the same number of AIF blocks before the ENDMAC statement.

Let's look at one more example. Since we have a CALL macro, why not have a SUBROUTINE macro. This macro will generate the .ENTR overhead code required of subroutines that conform to the .ENTR calling convention. This means that the SUBROUTINE macro will have to generate a labelled NOP for each of the actual arguments supplied in the invocation. Once again we will take advantage of the system ATV &.PCOUNT. Here is the SUBROUTINE macro definition.

```

MACRO
SUBROUTINE      &subName, &p1, &p2, &p3, &p4, &p5
&p1             AIF  &.PCOUNT>0
                NOP
                AENDIF
&p2             AIF  &.PCOUNT>1
                NOP
                AENDIF
&p3             AIF  &.PCOUNT>2
                NOP
                AENDIF
&p4             AIF  &.PCOUNT>3
                NOP
                AENDIF
&p5             AIF  &.PCOUNT>4
                NOP
                AENDIF
&subName       NOP
                EXT  .ENTR
                JSB  .ENTR
                DEF  '*-'&.PCOUNT'-1'
                ENDMAC

```

Now we can use the SUBROUTINE macro to begin the body of subroutines callable with the CALL macro. Try the above macro with the invocation

```

SUBROUTINE      ERMSG, ERCOD, ERSTR.

```

This should expand into the following code fragment:

```

ERCOD          NOP
ERSTR          NOP
ERMSG          NOP
                JSB  .ENTR
                DEF  *-3-1

```

3. THE FINITE STATE MACHINE

The macro processor can be thought of as a machine that can occupy any one of a finite number of states. At the most basic level, there are only two states: we shall call these P for pass-through, and T for translate. In the P state the machine accepts input and emits it as output, unchanged. When in the T state, however, input undergoes transformation of one kind or another. Instructions understood by the machine while in T state allow us to describe these transformations. These instructions constitute a language which we will call T_0 . Our goal is to construct a language, T_1 , in which we define the desired control structures using T_0 as our basis.

The elements of T_0 and T_1 can be grouped into three categories: initiation, continuation, and completion. Initiation elements cause transition from one state into another. They also cause information about the current state to be saved before the transition to the new state is made. Continuation elements effect the transformations made while in the current state. Completion elements perform the transformations required prior to exiting the current state. They are also responsible for restoring the information about the previous state which was saved by an initiating element. A complete set of elements will consist of one and only one initiating element, one and only one completion element, and an unspecified number of continuation elements. Such a set comprises an *activation* of T. Just as transitions may be made between states, transitions are likewise made between activations. By treating transitions between activations the same way we treat state transitions, we can nest elements of T_1 . Since the terms *activation* and *state* are interchangeable, we will refer to activations as derived states. Whereas we previously spoke of transitions between T and P, we can now be more specific and say that a transition is made between states within T, and between these derived states and P.

It does not take a great stretch of the imagination to see that, in the context of our discussion, T_0 is the set of assembler pseudo-ops which control the macro pre-pass of MACRO/1000. T_1 , then, is the language the constituents of which are the topic of this paper.

3.1 SUPPORTING DATA STRUCTURES

In order to be able to make transitions through a state to other states, and then backtrack the way we came, we must have a way of saving the information pertinent to the state we are leaving, and restoring this information when we return. This is analogous to leaving a trail of bread crumbs in order to find one's way back out of a maze. The central data structure used in these types of operations is a Last-In-First-Out (LIFO) stack or, simply, a stack. In reentrant subroutine calls a stack is used to save the current context or activation of a subroutine, and to pass arguments to the called subroutine. In finite state machines, the stack is used to save information about the current state of the machine. As we shall see below, our IF-ELSEIF-ELSE-ENDIF structure is a complete set, and will behave as a single activation even though it is implemented with several separate macros. This is accomplished by saving the context of the current control structure on a stack that we will implement with IGLOBAL and CGLOBAL macro ATV arrays (stacks) and scalars (stack pointers).

A (usually) scalar datum, called the stack pointer, is used to indicate the location of the top of the stack. This is the next location at which data may be stored. There are two operations applied to stacks, with which most programmers are familiar. The push operation will store data on the top of the stack and update the stack pointer. The pop operation takes the topmost data off of the stack and updates the stack pointer. Since the stack is generally considered to be a linear arrangement of contiguous storage cells, the stack pointer updates are usually nothing more than increment and decrement operations.

The update operations should obey a few rules. First, if the stack pointer is incremented before putting new data on the stack, it should be decremented after popping data. The convention used will also depend on whether the stack grows from low memory to high, or vice versa. The stacks implemented below grow from higher locations to lower ones. A predecrement - postincrement convention is used for the stack pointer updates. When a stack is popped, the stack pointer is first decremented, then data are taken from the location(s) indicated. A push operation will store the data where directed by the stack pointer, followed by an increment of the stack pointer. By following this convention, we always know that the stack pointer points to the next available location to which data may be stored.

Now that we have a way to save information about a control structure for use by the constituent macros, we turn to the problem of deciding exactly what information we need to save. The first datum we need to save is the point at which we will exit the structure. For a structure like IF-ELSEIF-ELSE-ENDIF we also need to save the location of the beginning of the code for next condition to check (i.e. the next ELSEIF or ELSE). The following example illustrates the use of these data.

```
IF condition1
    action1
ELSEIF condition2
    action2
ELSE
    action3
ENDIF
```

When the IF macro is invoked, code will be generated which will evaluate *condition1*. If *condition1* evaluates to a true condition, the *action1* code will be executed. The macro must also generate code to jump around the *action1* code in the event *condition1* proves false. The simplest way of doing this is to have the macros generate their own unique labels. Of the four macros in this structure, only the IF and ENDIF macros know deterministically what control macros came before and will come after them. Since the IF macro is always used first, it is in the unique position of deciding what the label for the ENDIF macro will be. The IF, ELSEIF, and ELSE macros will also be able to decide what the label for the next macro in the structure will be. Note that this decision can be made regardless of the fact that at the time of their invocation, the macros will have no knowledge of which other macros in the structure will follow.

The ability to generate these labels depends on two things. First, we need to have a way to pass the generated label to the macro which will use it. The stack works very nicely for this. We also need to have a way to generate unique labels. There are two ways to approach this problem. We could declare a global integer ATV which will be incremented each time we use it. If, however, we forgot to increment it, we would run the risk of generating non-unique labels. A second method is to use another system ATV called "&Q". &Q will guarantee a unique number between 1 and 32767 each time it is used. The macro processor takes responsibility for updating &Q's value, taking this bit of tedium away from the programmer. This latter solution is, in fact, the one which shall be used.

At this point we have determined at least two pieces of information which need to be saved on the stack; a label which will be used to reference the end of the control structure, and a label which will be used to reference the next macro in the control structure other than the structure terminator macro (e.g., ENDIF). These data will be called &endPoint and &stLevel, respectively.

A third datum used to describe the current structure type is included for the purpose of forcing macro usage to adhere to structured programming principals. To see how this works, consider the code fragment

```
IF condition1
    WHILE condition2
        (code body)
ELSEIF condition2
    (code body)
ENDWHILE
ENDIF
```

We do not want our macros to allow this kind of code which is at best ambiguous, and at worst sloppy. By having the structure terminator macro check to see which structure is being terminated we can disallow code like the example above. This information will be found in the ATV called &stType. To facilitate searches in the stack for the start of a frame, &stType will always be negative. The reason for such searches will become apparent when we discuss the SWITCH structure, below.

It has not yet been decided whether the three data will be integer or character in type. It is highly desirable to make them all of one type or the other since mixing types would require us to maintain and coordinate two separate stacks. In an earlier paragraph it was stated that the uniqueness of the labels would be guaranteed by using the &Q system ATV. Since this is integer, and since it is the only part of the labels which needs to change, the ATV's used for control structure context will also be integer. The labels themselves will take on the forms

L.&stLevel

for intermediate macros such as ELSEIF and ELSE, and

L.&endPoint.Z

for terminator macros like ENDIF.

The context described by `&endPoint`, `&stLevel`, and `&stType` are collectively called a *frame* in the stack. Another, more general term sometimes used is *activation record*. At any given time, for each frame in the stack the stack pointer (called `&RSP`) will point to the location containing `&stType`. `&RSP+1` will be the location of the current `&stLevel` value, and `&RSP+2` will indicate the current `&endPoint` value. `&RSP` is always incremented by three. If not, the stack would become "out of sync".

Since one of the reasons for using macros is to avoid needless repetition of source statements on the part of the programmer, two macros are defined which will do all of the pushing and popping of the stack. These are called `CM PUSH` and `CM POP`, respectively. `CM POP` takes no arguments; it merely restores context to its state prior to the last `CM PUSH` macro call. `CM PUSH` takes a single argument which it uses to set the new `&stType` value after pushing the current `&stType` on the stack.

4. IF-ELSEIF-ELSE-ENDIF

Now that the groundwork has been done we can examine the mechanics of the control structures themselves. The `IF-ELSEIF-ELSE-ENDIF` structure was chosen for this paper because of its relative simplicity. The general form of both the `IF` and `ELSEIF` macros is

```
IF    operand1,relop[,operand2].
```

The only difference between the `IF` and `ELSEIF` macros is the name, and the fact that the `ELSEIF` macro will be referenced by an `&stLevel` label.

Operand1 and *operand2* are labels which point to the values to be compared. Legal operands include labels to `DEF` pseudo-ops, and constant declarations such as `=D10`. One could also specify a number such as `33` as an operand. This would, however, reference memory location `33` and, since this location is in the base page, such a reference would cause a memory protect (MP) violation. The same would happen if a label that was `EQU'd` (as opposed to `DEF'd`) to some value were used.

Relop can be one of a set of relational operators. Being able to test for conditions of equality ("`=`" or `EQ`), inequality ("`<>`" or `NE`) are two requisites which immediately come to mind. Table 1 shows the complete list of relational operators which are supported.

Table 1. IF-ELSEIF-ELSE-ENDIF Relational Operators

=,==,EQ	equality
<>,! =,NE	inequality
>,GT	greater than
<,LT	less than
>=,GE	greater than or equal to
<=,LE	less than or equal to
!,NOT	logical complement
AND	logical conjunction (true if op1 and op2 are both non-zero)
,OR	logical or

The logical complement operator (! or NOT) takes only one argument. This is why *operand2* is shown as optional above.

In order to avoid duplicating definitions in the IF and ELSEIF macros, that part of the macro which generates the code for a given test will be found in a separate macro called EVALUATE. IF and ELSEIF will simply decide how many arguments there are and set up the required context information. They then both invoke EVALUATE to do the rest of the work.

The IF macro itself is actually very short. It first invokes CMPUSH passing the string 'IF'. Next, the global ATV &cmError is checked to see if an error occurred during the CMPUSH call, such as a stack overflow (no room left on stack). Assuming no error occurred, IF then checks to see if a label was supplied in the IF invocation. If so, an EQU is generated with the supplied label. Finally, EVALUATE is called with the arguments originally passed to IF. If an error was generated during the CMPUSH call, the error count is incremented and no code is generated. The entire IF macro is listed below.

```

MACRO
&label    IF    &op1,&relOp,&op2
           CMPUSH    'IF'
           AIF    :NOT:&cmError
               AIF    &label <> ''
&label    EQU    *
           AENDIF
           EVALUATE    &op1,&relOp,&op2
AELSE
&cmError  ISET  0
AENDIF
ENDMAC
    
```

Note the use of the macro operator ":NOT:". An expression in a conditional assembly structure such as AIF will be evaluated and tested to see if it is zero or non-zero. If it is zero, the condition is false. By applying the :NOT: operator, the expression will evaluate to the logical opposite. In the first AIF statement, the ATV &cmError is tested to see if it is :NOT:0.

There are several assembly-time operators that are available to the MACRO/1000 programmer. Among these are the :AND: operator and the :OR: operator which will be used in the EVALUATE macro. Other operators include :EQ:, :NE:, :GT:, :LT:, :GE:, and :LE:. Understand that these operators are meaningful only to the macro assembler as it processes the control structures used in conditional assembly, such as AIF, and AWHILE. In terms of the languages discussed above, all of these are elements of T₀.

As was said earlier, The ELSEIF macro differs in only one respect (other than the name) from the IF macro. The ELSEIF will, in addition to a user supplied label, generate an EQU for a label created using the current value of &stLevel. This is done because EVALUATE will generate statements to test the condition supplied to IF or another (earlier) ELSEIF. But it will also generate code that will cause a jump to the next location to execute should the run-time condition fail. Consider the following example:

```

IF      testval,==,=D0
        LDA  nextval
ELSEIF  testval,==,=D1
        LDA  preval
        :
```

This will generate the following code fragment:

```

        LDA  testval      <
        CPA  =D0          < Generated by
        RSS                < IF macro
        JMP  L.1          <
        LDA  nextval
L.1     JMP  L.2.Z        <
        EQU  *            <
        LDA  testval      < Generated by
        CPA  =D1          < ELSEIF macro
        RSS                <
        JMP  L.3          <
        LDA  preval
        :
```

The IF macro in the example has set the value of &stLevel to 1. ELSEIF picks up this value and uses it to generate the L.1 label referenced by the JMP L.1 statement produced by the IF macro.

The ELSE macro takes no arguments. It simply generates another &stLevel label that can be referenced by the last IF or ELSEIF along with any label the programmer may have supplied. The ENDIF macro can generate up to three labels. Two of them, the &stLevel label and the &endPoint label, are always produced. The third is an optional, programmer-specified label. ENDIF also pops the context stack using CMPOP so that the context prior to the invocation of the IF structure can be restored. One of the things that is checked when the stack is popped is to see if the &stType ATV is set to the correct structure type. If it is not, an error message is generated. This check is provided in order to alert the programmer to structure violations such as the one shown earlier.

4.1 IF Example

The following is a complete example of all the IF-block macros and the code generated thereby.

```

                IF    TABLESIZE, EQ, =D-1
                LDA  =STABLE
                INA
                JMP  @A
JMPTBL        ELSEIF TABLESIZE, EQ, ZERO
                JMP  NOTABLE
                ELSEIF TABLESIZE, LT, MAXTABLE
                LDA  =STABLE
                ADA  JMPINDEX
                JMP  @A
                ELSE
                JMP  JMPTBL
                ENDIF
```

generates (assuming &stLevel is initially set to 2 and &endPoint is set to 1)

```

                LDA  TABLESIZE
                CPA  =D-1
                RSS
                JMP  L.2
                INA
                JMP  @A
                JMP  L.1.Z
JMPTBL        EQU  *
L.2           EQU  *
                LDA  TABLESIZE
                CPA  ZERO
                RSS
                JMP  L.3
                JMP  NOTABLE
                JMP  L.1.Z
L.3           EQU  *
                LDA  TABLESIZE
                CPA  MAXTABLE
                JMP  L.4
                CMA, INA
                ADA  MAXTABLE
                SSA, RSS
                JMP  L.4
                LDA  =STABLE
                ADA  JMPINDEX
L.4           EQU  *
                JMP  JMPTBL
                JMP  L.5
L.5           EQU  *
L.1.Z        EQU  *
```

5. SWITCH-CASE-DEFAULT-ENDSWITCH

The IF-block control structure is relatively simple in that only the continuation and completion labels need to be preserved across macro invocations. The SWITCH-block construct requires this information along with several additional data. The SWITCH-block implemented here were patterned after the switch control structure found in the C programming language. The switch statement in C takes a single argument or expression which evaluates to an object of type int (integer). Each subsequent case statement compares the switch argument with its own argument. If the two arguments are equal, execution continues at that point until either a break statement is encountered, or the end of the switch structure is reached. One peculiarity of the C switch statement is that several case statements may be executed as long as no intervening break statements are encountered. For example

```
switch (c)
{
    case 'a':
        printf("alpha");
    case 'b':
        printf("bet");
        break;
    case 'c':
        printf("soup");
}
```

will print "alphabet" if the variable c is equal to the ASCII value for lowercase a. On the other hand, if c is equal to the character 'b', only the string "bet" will be printed. This is because the break statement will cause an exit to be taken from the switch block. When c is set to 'c', the string "soup" is printed and the switch block is exited. The macros used to emulate this control structure in MACRO/1000 will have exactly the same functionality. The only difference will be one of semantics. Whereas the C switch uses "{}" pairs to begin and end a switch block, the MACRO/1000 macros will begin a SWITCH block with the SWITCH macro and end it with the ENDSWITCH (or ENDSW) macro. Thus, SWITCH is an initiating element, CASE, BREAK and DEFAULT are continuation elements, and ENDSWITCH is the completion element. Note that the colon (:) will not be used to terminate CASE macros.

SWITCH does the usual initiation work, along with generating an "EQU *" for a user-supplied label, if one is provided. It then emits a "JMP L.&endPoint" line which will cause execution to be transferred to the beginning of the jump table. The definition for SWITCH is shown below.

```
MACRO
SWITCH      &x
CMPUSH     'SWITCH' ; Save context.
AIF :NOT:&cmError ; Problem?
&CSP      ISET &CSP-1 ; Nope.
&RSP      ISET &RSP-2
```

```

                                AIF  &RSP<=0   ; Rstack Ovf?
&RSP                            ISET  &RSP+2
&.ERROR                          ISET  &.ERROR+1
                                AELSEIF  &CSP<=0   ; Cstack Ovf?
&CSP                            ISET  &CSP+1
&RSP                            ISET  &RSP+2
&.ERROR                          ISET  &.ERROR+1
                                AELSE
&Cstack[&CSP]                   CSET  &x
&Rstack[&RSP]                   ISET  &caseCount
&Rstack[&RSP+1]                 ISET  &default
&caseCount                      ISET  0
&default                        ISET  0
                                JMP   'L.'&endPoint'.Z+1'
                                AENDIF
                                AELSE
&cmError                        ISET  0
                                AENDIF
                                ENDMAC

```

Aside from the error checking instructions, one will notice that an additional stack, &Cstack, has been introduced. This stack is used to save the test (&x) and CASE values until later, when invocation of the ENDSWITCH macro will cause a jump table to be generated. We will discuss the jump table further when we come to the ENDSWITCH macro. Furthermore, the IGLOBAL &default is pushed on the &Rstack and then set to zero. The purpose of this ATV is to let the SWITCH structure know whether or not a DEFAULT macro has been invoked *within this activation*. If a DEFAULT macro has been invoked, no more CASE macros will be allowed.

The CASE macro emits zero words of machine instructions. The only code emitted is an "EQU *" for a user-supplied label (if present), and an "EQU *" for the &stLevel label. The rest of the CASE macro definition handles stack errors, and saves the CASE value on the &Cstack. Here is the CASE macro:

```

                                MACRO
&label                          CASE &test
&label                          AIF  &label<>' '
                                EQU  *
                                AENDIF
&CSP                            ISET  &CSP-1           ; Push Context
&RSP                            ISET  &RSP-1
                                AIF  &CSP<=0           ; Cstack Ovf?
&CSP                            ISET  &CSP+1
&.ERROR                          ISET  &.ERROR+1
                                AELSEIF  &RSP<=0       ; Rstack Ovf?
&CSP                            ISET  &CSP+1
&RSP                            ISET  &RSP+1
&.ERROR                          ISET  &.ERROR+1
                                AELSEIF  &default ; CASE aftr DEFAULT?
&.ERROR                          ISET  &.ERROR+1
&CSP                            ISET  &CSP+1

```



```

&RSP                ISET &RSP+1
                    AELSE                ; None of the above.
&Cstack[&CSP]      CSET &test
&Rstack[&RSP]      ISET &stLevel
&caseCount          ISET &caseCount+1
'L.'&stLevel        EQU *
&stLevel            ISET &.Q
                    AENDIF
                    ENDMAC

```

Notice the use of the system macro &.Q. Also note that since no executable code is generated, it is possible for execution to "fall" right through the CASE label into the succeeding code. The last AELSEIF handles the case wherein a DEFAULT macro has been invoked earlier in this same activation.

The BREAK macro will cause code to be generated that will transfer execution to the end of the current control structure. This includes SWITCH and several other control structures not described in this paper (e.g., WHILE and FOR loops). It simply searches backward through the &Rstack until it finds an &stType indicator. These are not hard to find since they are always negative and other stack objects are always non-negative.

```

                    MACRO
                    BREAK
&outPoint           ILOCAL    &RSP
                    AWHILE    (&outPoint<&MAXR):AND: \
                        (&Rstack[&outPoint]>0)
&outPoint           ISET    &outPoint+1
                    AENDWHILE
&outPoint           AIF    (&outPoint<&MAXR):AND: \
                        (&Rstack[&outPoint]<0)
&outPoint           ISET    &outPoint+2
&outPoint           ISET    &Rstack[&outPoint]
                    JMP    'L.'&outPoint'.Z'
                    AELSE
                    JMP    'L.'&endPoint'.Z'
                    AENDIF
                    ENDMAC

```

The BREAK macro introduces a new T_0 element (i.e., MACRO/1000 pseudo-op): the AWHILE instruction. It is while executing this loop that the BREAK macro searches for the beginning of the current control structure. It is noteworthy that MACRO/1000 has a REPEAT instruction which will loop a specified number of times.

The DEFAULT macro is very similar to the CASE macro and, thus will not be shown here. There are two differences between DEFAULT and CASE: DEFAULT sets the &default ATV to 1 to indicate the occurrence of a DEFAULT macro within this activation, and since the DEFAULT macro takes no argument, no case value is pushed onto the &Cstack.

The real work for this structure is done by the ENDSWITCH macro, the primary job of which is to build the jump table. Like the other macros, ENDSWITCH emits an "EQU *" for a user-supplied label. It then emits

```
L.&stLevel      EQU  *
```

to satisfy any earlier reference which may have been made. Next, it emits a JMP instruction with an address equal to the location of the end of the table plus one. This is followed by

```
L.&endPoint     EQU  *
```

which was referenced by the SWITCH macro.

The final section of ENDSWITCH creates the jump table. This table consists of one two-line entry for each CASE macro invoked within this activation. These entries are of the form

```
CPA  &Cstack[&count]
JMP  L.&Rstack[&count].
```

Recall that the &Cstack contains the case values supplied as the argument for CASE macros within the current activation. A count is kept of the number of CASE invocations. This count is then used to compute the depth within the &Cstack and &Rstack to which the array index ATV &count must be set. The CASE count is also used as the repetition number for a REPEAT loop within which the actual jump table is generated. Finally, a jump instruction to the location of the (possibly nil) DEFAULT code is generated, followed by a label marking the end of the structure. Of course, ENDSWITCH's dying act, like any good completion element, is to restore the previous state of T. Here then is the definition for ENDSWITCH:

```
MACRO
&label          ENDSWITCH
AIF  &label<>' ' ; User label?
&label          EQU  *
AENDIF

.*
.*  Point to beginning of CASE values in &Cstack
.*  and CASE labels in &Rstack.
.*
&LP1           ILOCAL   &CSP+&caseCount
&LP2           ILOCAL   &RSP+&caseCount
AIF  &stType<>-2   ; Type ok?
&.ERROR        ISET  &.ERROR+1
AELSE

.*
.*  Generate JMP to skip over table when finished
.*  with CASE or DEFAULT code body.
.*
'L.'&stLevel    EQU  *
```

```

&jmpTableLen      ILOCAL      (&caseCount-
                        &default+1)*2+&default
'L.'&endPoint'.Z'  JMP      '*+'&jmpTableLen
.*
.*      Load up the test value from SWITCH macro.
.*
                        LDA      &Cstack[&LP1]
.*
.*      Generate the jump table.
.*
                        AWHILE    &LP1>&CSP
&LP1              ISET      &LP1-1
&LP2              ISET      &LP2-1
                        AIF      (&LP1=&CSP):AND:
                                (&default)
                                JMP      'L.'&Rstack[&LP2]
                        AELSE
                                CPA      &Cstack[&LP1]
                                JMP      'L.'&Rstack[&LP2]
                        AENDIF
                AENDWHILE
.*
.*      Restore last state's context.
.*
&RSP              ISET      &RSP+&caseCount
&CSP              ISET      &CSP+&caseCount+1
&caseCount        ISET      &Rstack[&RSP]
&RSP              ISET      &RSP+1
&default          ISET      &Rstack[&RSP]
&RSP              ISET      &RSP+1
                AENDIF
                CMPOP
                ENDMAC

```

(Lines beginning with "." will not appear in the emitted code.)*

Before moving on to an example, it would be worthwhile to examine an alternate method for implementing this structure. This method uses a discontinuous jump table. That is, pieces of the jump table are imbedded in the body of the code, rather than placed, in one piece, at the end of the code body. The SWITCH macro would generate a LDA for the test value. Each CASE macro would generate code which looks like this:

```

L.&stLevel        RSS
                  CPA      &caseValue
                  RSS
&stLevel         ISET      &.Q
                  JMP      L.&stLevel

```

The same "fall-through" functionality is found here as is provided by the first method. The main problem with this method, however, is that it requires four words of memory for each CASE macro. The chosen method uses only two words per CASE plus three additional words for the overall structure. For small SWITCH structures the difference is trivial. Most applications, however, typically use several CASES in a SWITCH. The second method does offer the advantage that no &Cstack is required, nor are the CASE count and additional storage cells in &Rstack. It has the further advantage that the code generated looks less convoluted than code generated using the first method. But since it was stated at the beginning of this paper that assembly is usually chosen for its ability to facilitate the generation of lean code, the first method became the method of choice.

5.1 SWITCH Example

We conclude this section with an example using the SWITCH structure. To demonstrate the recursive nature of T1 this example will contain a SWITCH within a SWITCH. Since MACRO/1000 is case insensitive, the example has been written in lower case letters.

```

switch      =D5
  case =D6
    call exec,=d1,,abc
    break
  case =D5
    switch   abc
      case df
        jmp  endit
        break
      case =d4
        call fn1,=d2,abc
        break
    endswitch
  jmp  endit
default
  lda  =D4
endswitch

```

In the expanded rendition of the preceding program segment, lines beginning with ** are included for reference only.

```

**  switch      '=D5'
      JMP      L.2.Z+1
**          case '=D6'
L.2      EQU      *
**          call 'exec', '=d1',, 'abc'
      JSB      exec
      DEF      *+3+1
      DEF      =d1
      DEC      0

```

```

DEF abc
** break
JMP L.2.Z
** case '=D5'
L.3 EQU *
** switch 'abc'
JMP L.8.Z+1
** case 'cdf'
L.8 EQU *
jmp endit
** break
JMP L.8.Z
** case '=d4'
L.9 EQU *
** call 'exec', '=d2', 'abc'
JSB exec
DEF **+8+1
DEF =d2
DEF abc
** break
JMP L.8.Z
** endswitch
L.11 EQU *
L.8.Z JMP **+6
LDA abc
CPA cdf
JMP L.8
CPA =d4
JMP L.9
jmp endit
** default
L.6 EQU *
lda =D4
** endswitch
L.16 EQU *
L.2.Z JMP **+7
LDA =D5
CPA =D6
JMP L.2
CPA =D5
JMP L.3
JMP L.6

```

6. CONCLUSION

Although this paper is somewhat lengthy, we have only begun to scratch the surface of the theory and practice of compiler design and construction. Compilers, for instance, would not ordinarily emit assembly language source code. Instead, a compiler would typically produce some sort of binary relocatable or absolute code. Moreover, compilers are capable of parsing expressions, usually supplied in infix form, and generating code which, when executed by the target machine, evaluates the expressions.

The reader should, however, have gained a better understanding of how high-level language compilers work. Such an understanding will ultimately lead to better programs.

7. BIBLIOGRAPHY

MACRO/1000 Reference Manual Copyright (c) 1986, Hewlett/Packard Company

Brian Kernighan, Dennis Ritchie; *The C Programming Language*
Copyright (c) 1978 Bell Telephone Laboratories, Incorporated

Alfred V. Aho, Jeffrey D. Ullman; *Principles of Compiler Design*
Copyright (c) 1977 by Bell Telephone Laboratories, Incorporated

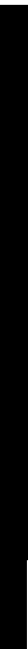
Using the HPCRT Library
Alan Tibbetts
Hewlett-Packard Co.
1266 Kifer Road
Sunnyvale CA

With the introduction of the Compatible Serial Drivers at Rev. 4.11, HP also introduced a new utility library called HPCRT.LIB. Although the primary purpose of this library is to provide routines needed by the new drivers, it also serves as a convenient gathering place for an assortment of routines that are generally useful when developing interactive programs, especially those that use the features of Hewlett-Packard CRT's.

The library can be divided into four classes:

- routines specific to the compatibility drivers
- routines that are useful for CRT I/O
- a mini-formatter especially suited to systems programming
- miscellaneous routines

The talk will also describe the use of the mini-formatter and the process of adding a new format.



Some Advanced Software Techniques

Paul Schumann

E-Systems, Inc.

P. O. Box 1056 CBN 101

Greenville, TX 75401

Introduction

Over the many years I have worked with the various revisions of RTE, I have developed a small collection of tools and techniques now employed in my own programming. This paper is intended to discuss these techniques, and to describe some uses of the tools which are either already in or will soon be added to the Contributed Software Library:

- 1) IMRel: How Disassembly can add Advanced Capabilities to your code
- 2) Making it Smaller: A Formatter that uses the Resident System Routines
- 3) Keeping it Small: A Segmentation Alternative to MLS or CDS
- 4) Making it Faster: Data-structure Exploitation with Manual EMA-mapping
- 5) Avoiding the "Watered Effect" with "Data-Hiding" and "Strong-Typing"

IMRel: How Disassembly can add Advanced Capabilities to your code

IMRel (Inverse Macroassembler for Relocatable files) is a two-pass disassembler for relocatable files; since I recently extended it to handle absolute (type-7) files it also somewhat misnamed. During the first pass it looks for all memory references and builds a symbol-table in EMA; during the second pass it decodes all instructions, providing pseudo-labels where appropriate and using entry-point, external, and debug labels if they are available (which seems like a very good reason for always compiling or assembling with the debug option on). IMRel's predecessor at E-Systems was a program called RELIX, a descendant of RELIA, which was part of the CSL many years ago. RELIX became unusable for two reasons:

- 1) CI file-space was added to the system and,
- 2) Debug records were added to relocatable files.

I decided to build the "ultimate" disassembler with one goal in mind: "If it is in the relocatable code, I want to tell you about it." I would like to think I succeeded. I am introducing it in this way because IMRel (along with KERMIT) will serve a variety of purposes in the remainder of this paper.

Just before the first version IMRel was completed, one of my co-workers approached me with a problem: some FTN7X code was required on an RTE-4B system and some of the FTN7X intrinsic routines for bit manipulation were missing from the system library. After extracting the appropriate routines from the RTE-6 system, and disassembling them, my co-worker went away happy. This was IMRel's first use (even in an incomplete

condition) and the time I took to build IMRel was immediately justified. As a side note, if you need a program to perform module extraction from a library, ask your SE for a copy of RELIB, the relocatable-library manager. It is a part of the SE Service Kit (SSK), which I understand may soon be added to the CSL

IMRel has proven useful in two different pursuits. As the title of this section implies, if a given system routine provides a service you like and would like to add it to your code, or if it perhaps has some small side-effect you don't like, IMRel can provide you with a MACRO source to that routine, to use as you see fit. (NOTE - it should come as no great surprise that you cannot expect support from Hewlett-Packard for any code you might modify in this manner, so BE CAREFUL!) The other usefulness of IMRel is as a tool to gain a greater understanding of any code in the system; it has helped me to find and fix some system bugs.

KERMIT has been a major beneficiary of IMRel's services. Here are several of its "advanced capabilities" which are a result of a disassembly of some system routines:

- 1) Disassembly of a routine from HPCRT.LIB, a new library which supports the 12040 "D" multiplexer showed me how to identify an LU using the new firmware. HPCrtSSRCDriver (HPCrt Special Status Read Compatible Driver) is a routine which tells its caller whether the given LU will accept a special status read; if it does, the caller can readily determine which interface- or device-drivers are being employed. Since KERMIT must be able to run on systems which may lack that library, I incorporated a part of this routine into KERMIT directly.
- 2) As some of you may already know, KERMIT-CX, a derivative of my KERMIT, comes free with CONNECT, a product of Don Wright and ICT. Don had to make some changes to my KERMIT in order to integrate it into his product, and in examining his changes, I found that I had never even considered what happens as the KERMIT server logs its session off, when ENQ/ACK protocol was in effect as KERMIT started. In the case of my KERMIT, since it tries to restore the mux configuration when it shuts down, log-off messages would need to time out before the session would actually go away. KERMIT-CX got around the problem by leaving ENQ/ACK off. After disassembling CLGOF in RTE-A, and with a little help from my Hewlett-Packard SE in the form of a listing of LOGOF for RTE-6, I was able to solve the problem. KERMIT now performs a completely "silent" log-off under either RTE-6 or RTE-A, and it doesn't need to leave a "dirty" mux configuration.

In the area of finding and fixing bugs, KERMIT has been a beneficiary once again. A major change to KERMIT as of revision 1.99 was transportability. The original reason for KERMIT's lack of transportability was the manner in which it determined whether an LU was on a mux port. Since I was

performing this operation differently now, as a part of determining whether the (A-series) mux was a "D" revision, I thought I would have no problems. I was surprised when KERMIT still had transportability problems under RTE-6. After a great deal of effort, I determined that a system routine, WhoLockedLu, was causing the problem. I call this routine inside of the SetLine routine to find out what program is using a given LU if KERMIT cannot get an LU lock. WhoLockedLu looks in the Device Reference Table word 3 for a given LU to get the resource-number of the lock, which is then used as an index into the resource-number table to get the ID-segment number of the locker, which then leads to the name of the locking program. Disassembly of WhoLockedLu showed a non-transportable reference into the system in order to find the location of the resource-number table

```
$alias /rntab/ = '$RNTB', NoAllocate
common /rntab/rntab
integer*2 rntab, AddressOf
<other code>
RnTabAd = AddressOf(rntab)
```

The corrected code, which you will find in K6SUBS, is

```
$alias /rntab/ = '$$RTB', NoAllocate
$alias xla = '.XLA', Direct
common /rntab/rntab
integer*2 rntab, xla
<other code>
RnTabAd = xla(rntab)
```

The actual problem was this: LINK will not set the "transportable" bit in a program unless [not a complete list!] it is extended background and makes all of its system references through the entry-points inside the "VCTR" module (which you must relocate first when you generate a system. \$RNTB is not in the VCTR module, but \$\$RTB is. I reported this to the Phone-In Consulting people, expecting it to be handled as an enhancement request, but I'm pleased to report that they are handling it as a bug.

Making it Smaller: A Formatter that uses the Resident System Routines

In the May/June (1987) issue of *TC Interface*, Bill Hassell, the "HP 1000 Guru," wrote a very interesting article called "Q-SUBS." In it, he enlightens us on a set of formatting routines which have been available in the CSL for four years, were incorporated into %DECAR at RTE revision 4.0, and which will soon be documented in the *Relocatable Library Reference Manual*. The entire article is very interesting, and I highly recommend it, but the introduction to Q-SUBS is of major interest here; it is a brief history of "The Formatter" and why so many of us tend to avoid it. I have received permission from Mr. Hassell to use parts of his article in the discussion which follows.

The ANSI standard for a Fortran formatter requires that it be able to accept and perform a run-time format, that is, formatting based on information which becomes defined as a program runs, rather than the "usual" formatting instructions which were defined when the program was written and compiled. A formatter which complies with the standard must be able to perform any kind of formatting all of the time, which requires typically 2.5K words; a program which prints the word "test" on the user's terminal and has a compiled size of 25 words will require 5 pages (including the base page) to run. Note that in addition to building the ASCII representation of the data, the Formatter is also responsible for the system I/O calls; this will also be required of any limited-function formatter replacement.

KERMIT was not going to be a segmented program, and I knew as the job began that I would probably not have space for "the formatter;" I decided, like so many before me, to build my own set of formatter routines, but with a difference. My formatter would exploit the system routines I was already directly or indirectly using within KERMIT:

- 1) Since KERMIT is primarily concerned with file-transfers over a terminal connection, the system I/O calls (EXEC-style) were already available,
- 2) KERMIT was already set to use some fairly small string routines (.SST and .SSTC) for building and parsing packets, so they were available anyway, and
- 3) the user would be able to perform masked file searches; these routines perform conversions using IntToDecimal and DecimalToInt.

The first version of "tpFM" (terminal print-line formatter) appeared in KERMIT consisting of three relatively small routines. The capabilities of this formatter include character-strings and integer*2 or integer*4 numbers. Number formatting permits left-justified numbers, and blank- or zero-filled right-justified numbers. In IMRel, additional capabilities were required, so the "fmXx" routines were born:

fmReset: clears the print-buffer
fmTab: pads spaces to the specified column
fmStr: appends character strings to the print buffer
fmI4: formats integer*2 or integer*4 numbers in decimal or octal
rJust: performs zero- and blank-fill right-justification
(fmH4) formats integer*2 or integer*4 numbers in hexadecimal

I had anticipated the need for hex formatting in IMRel, but I never used it; fmH4 can be found in a commented-out form, but it has been tested, so if you need it, enjoy! The Fortran source for the largest of these routines (rJust) fits nicely on a single page at six lines per inch, including comments and a labeled-common area, they are obviously fairly small.

Both mini-formatters exhibit a calling hierarchy which I will illustrate using the IMRel formatter. The number formatters convert their data to ASCII

using `DintToDecimal` or `DintToOctal` (except hex formatting), and then call `rJust`. `rJust` performs any justification requested or fills the resulting string with stars if the number is larger than the field would permit. What is important is that the result of the number-formatting operation is a string, which is appended to the print buffer using `fmStr`, just like any other "normal" call to `fmStr`. The `fmXx` routines build strings into the print-line buffer, and separate calls within `IMRel` copy the entire buffer to a device to for listing and the major portion of the same print-line buffer to a file in order to build the source file.

The `tpFm` routines in `KERMIT` work in a slightly different manner. All calls require a text string as the first parameter; if that string ends with an underscore, the line will be held for more formatting calls. A call which has a string ending with anything else will still result in the requested formatting operation, and then the line will be output to the user's terminal. Once terminal output is performed, the print-line buffer is automatically cleared for re-use. This sequence of operations was quite sufficient for the relatively simple formatting requirements of `KERMIT`.

Keeping it Small: A Segmentation Alternative to MLS or CDS

In the previous section, I noted that `KERMIT` was not originally a segmented program. In addition to a few bug-fixes, I wanted to add a "run" command, and I anticipated some future improvements as well. I was already out of space, so what could I do? I required compatibility with both `RTE-6` and `RTE-A` in any solution I used, and I wanted the minimum number of code differences between the versions for each system. I thought about the similarities between `RTE-A`'s `CDS` and `RTE-6`'s `MLS`, but since you cannot use `DEBUG/1000` on `MLS` code, and since at that time, `CDS` and `DEBUG` were not behaving well together, I determined that these were not applicable. My only solution seemed to be "traditional" manual segmentation, and the existing organization of `KERMIT` didn't seem to lend itself well to that.

The "traditional" method of manual segmentation assumes that you have a series of sequential processes to perform, few or no large loops, and localized branching. In other words, if the program became too large, you could cut it into a few relatively independent pieces, with a small main program to tie them together. The main program starts the first processing segment, the first segment would call in the second segment when needed, the second segment would replace itself with the third, and so on. The segmenting method found in the `Adventure` program is similar to this; the game is initialized in the first segment, and then play processing shifts constantly between the second and third segments. Even `Adventure` is somewhat unusual, however, because the second and third segments together form a very large loop; the connection between the two segments is fairly complicated as a result.

It was clear that KERMIT would not lend itself well to this sort of treatment. The theme of KERMIT is a central control (the command-processor module or the server module) and a number of subroutines which perform the needed operations. The only logical divisions seemed to be:

- 1) Command-processing, and all of the command-parsing routines
- 2) File-masking, which brought in a lot of system-library routines
- 3) Packet-I/O, with the packet-building and -parsing code.

What I needed was a method which would allow me to build overlays of subroutines. The main would need to know which subroutines were in which segments, but it would still have all of its calls in place.

Fortunately, Link came to the rescue. Unlike LOADR, Link "remembers" where a given module appears in a relocatable file, such as after a type-5 (segment) program. Even if the main needs to use a given routine, if it appears positionally after a segment module, that routine will relocate in the segment, and the main will link to the routine's address; of course, the main must insure that the appropriate segment has been loaded before it calls a segment-resident subroutine. One possible problem occurs when subroutines call other subroutines: you must insure that there are no cross-segment calls, because the return addresses will get lost in the overlaid segment, even if you arrange for the overlaid segment to be called back in. Therefore, some routines could not be made segment-resident; they had to be in the non-overlaid ("main") code. My first cut at this type of segmentation was fairly successful at accomplishing my goals; everything worked, but the linkage to the file-masking code was unnecessarily complicated. The 1.99 revision of KERMIT has removed this complexity.

For those who may still be using a revision of RTE for which there is no Link, this segmentation method is still available. A long time ago, LOADR required that all external references from the main had to be satisfied during the relocation of the main or the first segment, but this hasn't been the case for many years. LOADR can also support subroutines in segments, but the loading mechanism is more complicated than with Link. Given a main, main subroutines, and three segments, each with subroutines, the following loading sequence is effective:

- 1) Relocate file #1 consisting of the main, its subroutines, and the segment-1 program. As LOADR encounters the segment-1 program, it will attempt to satisfy all undefined externals from the main and its subroutines from the system or user-defined libraries, and then it will relocate the first segment.
- 2) Relocate file #2 consisting of the subroutines belonging to segment-1 and the segment-2 program. When LOADR encounters the segment-2 program, any undefined externals from the first segment will be satisfied from libraries as usual, and then the second segment will begin relocation.
- 3) The relocation of file #3 proceeds as in the second step; the third file consists of the subroutines to be loaded in the second segment,

followed by the segment-3 program. Any undefined externals from the second segment will be satisfied from user or system libraries.

- 4) The last file to be relocated consists of the subroutines to be located in the third segment. Any remaining external references from the third segment would be satisfied by the system and user-defined libraries as the end command was given, and the load would be complete.

Making it Faster: Data-structure Exploitation with Manual EMA-mapping

During the 1986 Interex conference in Detroit, Bill Gibbons gave us a talk on "system" programming in Fortran. He made a very brief mention about Extended Memory Arrays (EMAs) and mapping, and how a careful programmer could arrange to reduce the overhead associated with EMAs to nearly nothing. I had a performance problem in IMRel which I wanted to solve, and this appeared to be the path to the solution.

First, though, for those who may be newer at this, a little refresher. EMAs and the working-set portion of VMAs are implemented by setting aside two or more pages of the user's data space as a "window" into the physical memory of the EMA or the VMA working-set. As the user references data in the V/EMA, hardware sets this window, or "mapping-segment" (MSEG) so that the addressed data is "visible" (mapped), plus the page following the addressed data. Consider the following fragment:

```
do i=1,128                                !Move 128 elements
    LclVal(i) = EMAVal(i)                  !from EMA to local
End Do
```

Except for the first reference, remapping of the MSEG should not be required, regardless of the actual types of the arrays involved (integer, real, complex, or whatever), but the compiler will generate code so that each reference to the array EMAVal will remap the MSEG! If we know where the MSEG is, and if we know which part of the EMA is mapped at any given time, we can dramatically reduce the overhead by removing the extra remapping operations.

So how do we use this information? IMRel defines a 12-word symbol table entry as follows:

Words	Contents
1-8	Symbol-name (up to 16 characters)
9-10	Symbol-address (offset within a given relocatable space)
11	Symbol-identifier (gives symbol's relocatable space and origin)
12	Symbol-flags (gives symbol type and other information)

In order to simplify the access method as much as possible, I elected to map three pages of symbol-table space at any given time (3 pages = 256 symbols); this also made the conversion of a symbol-number to its EMA location quite simple, using $a = \text{ibits}(n, 8, 8) * 3;$ for a given symbol-

number "n", "a" in the above yields the first page-number of the 3-page space containing symbol-number n. IMRel has three routines which manage the entire symbol-table:

- 1) **MapIt** converts a symbol-number to an EMA page-number and offset within the MSEG (as above), and maps the appropriate EMA pages via a call to MMAP:

```
call MMAP (<start-page>, <number of pages>)
```

Note that MMAP always maps an extra page,

- 2) **PutSym** copies a 12-word symbol-table record into the EMA, and
- 3) **GetSym** copies a 12-word symbol-table record from the EMA.

```
$ALIAS /MSEG/ = 29696                                !See note 1
Subroutine PutSym(n, rec)
*,Put symbol into EMA

common /MSEG/ MSEG                                  !See note 1
integer*2 MSEG(0:3071)                              !See note 1
integer*2 n, rec(*)                                  !See note 2

integer*2 a, MapIt                                   !See note 3

a = MapIt(n)                                         !See note 3

call MoveWords(rec, MSEG(a), SymSiz) !See note 4
return
end
```

Notes:

- 1) The MSEG must be exactly three pages. If it is smaller, we would have to handle page-crossings in two different places in a symbol; if it is larger, the origin of the MSEG will be somewhere else, and the MMAP call (inside of MapIt) wouldn't map enough pages. Since MapIt returns an offset within the MSEG, we must index the MSEG starting at zero.
- 2) These lines limit the size of the symbol-table to 32767 symbols (I don't allow negative symbol-numbers); the definition of the local array as an array is not strictly necessary.
- 3) The call to MapIt converts the symbol-number "n" into information for MMAP (if needed) and an offset into the MSEG, returned as MapIt's value.
- 4) MoveWords can copy the data directly from the "local" (never-mapped) space to the EMA (via a "locally-mapped" MSEG array). The only required difference between the above and the GetSym routine is that GetSym reverses the positions of the first two parameters to MoveWords!

The EMA overhead actually decreases over the traditional “automatic mapping” method as the amount of contiguous data to be manually mapped increases. It is the access to IMRel’s symbol-table which I wanted to improve, and manually mapping the EMA reduced to 33% the amount of time required to decode a given file! If I should need to increase the size of a symbol-table entry at some later time, I would probably go to 16 words (even if it wastes some space) so that the conversion of a symbol-number to an EMA offset is still quick and easy.

Avoiding the “Waterbed Effect” with “Data-Hiding” and “Strong-Typing”

We at E-Systems have given a name to that property of some programs for which a bug-fix in one spot invariably results in another bug in some other spot; sometimes the new bug is even worse than the one just fixed. We call it the “waterbed effect” after the property of that piece of furniture to rise in some place other than where you press on it. With rare exception, programs exhibiting this behavior have:

- 1) Very large modules and very few subroutines
- 2) Implicit typing, or “implicit integer a-z” typing
- 3) No labeled commons, or one very large labeled (or unlabeled) common

By now, most of us already recognize the need for modular design of our programs, so I won’t dwell on the first point. Suffice it to say that a well-tested subroutine is not likely to misbehave. But beyond modularity, there are a few methods I will describe which can help to reduce the waterbed effect in a program.

It has been said that Niklaus Wirth, the father of the Pascal language, never had to program for a living, else he would have arranged for variables to be useable without declaring them first, as in the implicit typing rules of Fortran. I am a poor typist, but even the best ones occasionally have typographical errors, and “strong-typing” rules as found in Pascal provide the nearest thing to a spelling-checker that I have found so far. By adding **implicit none** to the top of each and every module, Fortran becomes a relatively strongly-typed language too; the only things you don’t need to declare are the intrinsic functions. With **implicit none**, Fortran warns me about the untyped variables, which could be typographical errors, a missing include statement, a missing declaration, or some other error. Since I also never use **dimension** in my code, preferring to explicitly type and dimension a variable all in one step, e.g.,

```
integer*2 array(250), i, j, k, parms(5)
```

Fortran also warns me about any attempts to redefine a variable I have already declared in a labeled common. This type of error can be frustrating to

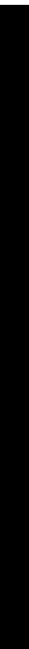
try to find at run-time; having Fortran find them for me as I compile makes the job much easier by removing a whole class of possible mistakes.

In Kernigan and Plauger's *Software Tools*, you can read about the concept of "data-hiding." Stated simply, "A piece of software cannot easily change the values of variables to which it has no access." Ignoring the lack of memory protection between pieces of the same program from each other under RTE, and assuming that all array indices stay within their bounds, the concept holds up well in practice. One must first segregate groups of variables into functional areas, like main controls, file controls, communications, debugging aids, formatting, and so on, as labeled commons. By setting up each labeled-common area as its own include-file, you can "hide" the unneeded information from a given routine (as opposed to placing all of the labeled-common areas into a single include file). Then, by only including the functional groups needed by a given routine, you can be more assured that a file-handling routine with no need for communications will not be able to accidentally alter a communications parameter, and vice-versa. Some of the beneficial side-effects of smaller, separately-includable labeled-common areas include

- 1) Ease of documentation: redundant comments (you do comment your include-files, don't you?) can be eliminated by only listing the includes in the **block data** routine.
- 2) Automatic maintenance of storage order: if you need to change the common, you only need to re-compile the program once you are done.
- 3) Easier alteration of a variable's data-type: you are not allowed to equivalence an integer array and a string to each other in a subroutine if either is passed as a formal parameter, but if an array is in a common, you could even equivalence it to two different character formats, if desired.
- 4) Faster compiles: the time needed to compile the routine is reduced, since the compiler needs to keep up with fewer symbols. Large commons clutter up the symbol-table with things that would never be used.
- 5) Easier to debug: a large number of symbols can prevent DEBUG/1000 from debugging your program due to a symbol-table overflow; by only including the variables a routine needs, the likelihood of this condition is dramatically reduced.

I present the following as an example of the utility of data-hiding efforts. In May of 1987, I suddenly found myself with a need for an absolute disassembler. Looking in the CSL index, then in Plus/1000 (its predecessor), and then in LOCUS (now we're really going back...) I could not find what I needed. Even though it would take (I thought) some time, my only choice seemed to be the extension of IMRel's capabilities to include type-7 files; the effort was actually complete in one hour!





IMAGE/1000: SECRETS HP NEVER TOLD YOU

A. Marvin McInnis
Consultant
5250 W. 94th Terrace, #114
Prairie Village, KS 66207

INTRODUCTION

IMAGE/1000 (92069x) is an excellent product which has become something of a stepchild since the introduction of IMAGE/1000-II (92081x), but the "old" IMAGE is still the better choice for many applications.

This presentation is directed to system managers, analysts, and programmers who, like the author, do not have access to IMAGE source code but want to get the best possible performance from the product. We will present at least a dozen techniques, developed over ten years, which allow users to surmount some of the apparent limitations of IMAGE and to significantly improve its performance in real-world applications. We will emphasize the benefits which can be achieved by adopting a requester/server structure for IMAGE programs. We will also discuss possible future extensions to these techniques, as well as the obstacles to successfully implementing them.

Most of the techniques presented here are supported by HP, but some are not; at least two performance improvements can be realized with no programming whatsoever! While many of these techniques were originally developed under RTE-4B and RTE-6, this presentation will presume an RTE-A environment. Utility programs to support some of the techniques described are available on the 1987 HP1000 Users Conference swap tape or from the author.

DATA BASE FUNDAMENTALS

At the most fundamental level, a data base can be defined as a technique or structure for storing data to facilitate the retrieval of that data. Period. In practice, a computer data base system usually includes software to create a data base, to store data in it and retrieve data from it, and to perform routine maintenance operations.

There are four primary benefits which should accrue from implementing a data base on a computer system:

- 1) Rapid random-access retrieval of data.
- 2) Flexible access to data.
- 3) Improved data integrity.

IMAGE/1000: Secrets HP Never Told You

4) Standardization of the programming environment.

Note that economy of storage and small program size are NOT among the benefits of data base. Indeed, it has been suggested that computer manufacturers write data base software in order to sell more discs! (Does anyone remember the reason Intel originally developed the microprocessor? It was because they were in the semiconductor MEMORY business!)

IMAGE/1000 FUNDAMENTALS

The original HP IMAGE/1000 (92063x ... IMAGE/1000-0?), was introduced late in 1976. A modified network-model data base, IMAGE pretty well satisfied our four criteria and was a real step forward for HP, despite severe limitations under RTE-2 and RTE-3. This product was superseded by the "old" IMAGE/1000 and is no longer sold by HP.

The "old" IMAGE/1000 (92069x) evolved from the original and was introduced at about the same time as RTE-4B. The enhanced file system under RTE-4B, using double word integers, at last allowed really large IMAGE data bases, although RTE-L, RTE-XL, and RTE-A.1 retained the earlier limits. Speed, data integrity, and the programmatic interface were all improved as well. While "old" IMAGE is still listed by HP as an "active" software product, it suffers from rather poor support and is for all practical purposes a "mature" product.

Finally, IMAGE/1000-II (92081x) was introduced in 1983, shortly after the introduction of the hierarchical file structure of RTE-A. While still evolutionary, IMAGE-II was a totally new product (with a price to match). Significant differences of IMAGE-II were its use of a central data base server program and extensive recovery facilities for enhanced data integrity.

Unfortunately, IMAGE-II has been plagued with "features" and performance problems which apparently still persist four years after its introduction. But despite these problems, IMAGE-II is by far the better choice if you are less concerned about performance than you are about data integrity facilities and HP support.

In this presentation we will limit ourselves to the "old" IMAGE/1000. The "old" IMAGE is still an excellent product, and there is a lot we can do to overcome its limitations and shortcomings.

AN IMAGE WISH LIST

In the limited time and space available, we are going to present a handful of successful techniques we have used with a number of large and very active IMAGE data bases. Let's start with with an IMAGE wish list.

IMAGE/1000: Secrets HP Never Told You

First, we would like to be able to overcome some of the IMAGE limits, especially under RTE-A:

- We need to be able to accommodate any number of users at a time, certainly more than 8!
- We would like to be able to create and maintain data bases under RTE-A as large as under RTE-6.

Second, we need to improve the security of IMAGE:

- We need to be able to perform more secure DBPUTs, DBUPDs, and DBDELS in a multi-user environment.
- We need to be able to maintain the structural integrity of the data base under all conditions .

Third, we would like to explore ways to improve the performance of IMAGE. We would like to obtain maximum improvement possible, with a minimum of programming and complexity. Some techniques we might examine are:

- Obtaining useful performance data
- Optimizing IMAGE's use of the FMGR file system
- Using multiple disc drives and interfaces
- Minimizing disc overhead
- Performing faster serial reads
- Performing faster chained reads

BUILDING A LARGE DATA BASE UNDER RTE-A

According to the HP literature, you cannot build large IMAGE data bases under RTE-A. The published limits are:

- Total data base size	800 Mb
- Maximum data set size	4 Mb
- Maximum data entries per data set	32,767

Besides being inconsistent, these limits would come as a surprise to some of my clients, one of whom has an RTE-A IMAGE data base of 348 Mb, with a single data set larger than 200 megabytes, several data sets with more than 200,000 entries each (one has 870,000 entries), and more than 40 simultaneous users! Putting it as politely as possible,

IMAGE/1000: Secrets HP Never Told You

these limits are pure bull dust! HP just never got around to finishing FMGR, its libraries, and its utilities when RTE-A superseded RTE-A.1.

The IMAGE library routines and most of the FMGR library routines work just fine. The only problems are FMGR itself, and the ECREA library routine. DBDS, the schema processor, can't build a large data base because ECREA will not create an extended file of specified size.

There are two solutions:

- 1) You can run DBDS on an RTE-6 system and then use FC to move the empty data sets to your RTE-A system. Be sure to specify the "L" option for the copy so that FC will do its own FMGR directory access rather than calling the ECREA routine.
- 2) Create the IMAGE root file with DBDS, using the options ROOT, NOSET, and TABLE. Then run our utility MDBDS (on the swap tape) for each data set in the data base, using the data set specifications contained in the DBDS list output.

MDBDS manages to create the large FMGR files by calling ECREA with SIZE(1) = -1 and then truncating the file to the desired size with ECLOS (which works correctly). Here is a sample code fragment:

```
SIZE(1) = -1
CALL ECREA(DCB,ERR,NAME,SIZE,2,SECU,CRN,DCBS,IRTN)
ITRUN = (IRTN/ 2) - SET_SIZE
CALL ECLOS(DCB,ERR,ITRUN)
```

After creating the data sets, MDBDS also initializes them just like DBDS does. But since MDBDS uses large data control blocks it is faster. Believe it or not, this works every time! There are also other reasons to use MDBDS, which we will cover later.

Whichever method you use, you will soon discover that the RTE-A FMGR doesn't know about extended files. D.RTR works correctly, so the FMGR "PK" command will work, but the "CO" command will corrupt your data sets. Whenever you move extended FMGR files on an RTE-A system, you MUST use the FC utility, using its "L" option. Curiously, the "DL" command works correctly, but you will get an ominous FMG-001 error if you use the "LI" command on an extended file; trust me, you can just ignore it! Our utility program DBCHK (on the swap tape) will allow you to examine records of extended files.

WARNING: Large data sets created this way are not supported by HP, although we think they should be. We have, however, been using these techniques successfully since the introduction of RTE-A in 1983.

IMAGE/1000: Secrets HP Never Told You

DATA INTEGRITY ISSUES WITH IMAGE

The integrity of a network model data base (including IMAGE) can be breached in two basic ways: 1) corruption of the structure of the data base, and 2) corruption of data record contents. Note that data base integrity issues in IMAGE are always concerned with writes to the data base (the DBPUTs, DBUPDs, and DBDELs), since those are the operations which modify its data and/or structure.

Corruption of the data base structure is always a very serious event and, when this kind of damage is detected, all data base activity should be stopped immediately until it has been repaired. The primary symptoms of a corrupt data base structure are broken chains: either 1) chains which are incomplete (links are missing) or 2) two chains which contain the same link. In IMAGE, broken chains are usually the result of either a system halt or a program abort.

There is little you can do to protect IMAGE from a system halt, but here are a few suggestions:

- Adopt a good backup plan and follow it religiously!
- Locate the system console where it cannot be rebooted by ordinary users.
- Locate the CPU and disc in a low-traffic area which is kept clean and well ventilated; a separate room is ideal. Cables should be carefully routed and tied. Clean filters once a month.
- Invest in a good quality uninterruptible power supply (UPS) for the CPU, discs, and CRT terminals. If you do not power your terminals from the UPS, and especially if you use block mode, you should always design your programs survive a multiple CRT lobotomy.
- Keep your CE (and anyone else with a screwdriver) away from the system unless there is a known hardware problem and the data base is closed. If you are on a maintenance contract with HP, you should insist that your CE know how to safely shut down YOUR system.

The DBCOP and RECOV modules of IMAGE provide fair protection of the data base structure after a program abort, but only if you remember to run RECOV! Of course, you can avoid aborts in critical areas of code if your programs have been carefully designed and coded, include extensive error checking, and have been thoroughly debugged in an actual production environment.

You can achieve additional protection against program aborts or system

IMAGE/1000: Secrets HP Never Told You

halts by performing only "soft" deletes from user programs. This requires that you define a single word (or two character) flag field in each manual master or detail data set. Then, rather than doing DBDELS in your programs, perform DBUPDs to set the flag field to a unique "delete" value. The data base structure will be protected, since the DBUPD does not modify the chain structure. Actually deleting the flagged records can be accomplished by a batch process as part of your backup procedure or at some other non-critical time.

PROTECTING DATA RECORD INTEGRITY

If IMAGE has an Achilles' Heel, it is the difficulty of maintaining the integrity of data record contents when performing DBPUTs, DBUPDs, and DBDELS in a multi-user environment. And the culprit here is not IMAGE itself, but the FMGR file system's data control block (DCB) buffering scheme.

The essence of the problem is that FMGR's DCB buffering does not provide a reliable way to force physical reads from disc. When your program calls DBGET, the record is moved to your buffer from the DCB; there is no way to determine whether or not a physical disc read was performed.

For example, a record can be modified by a program, and the change posted to the disc, but another program's DCBs may not reflect the update. Consider the following scenario:

- 1) Program A and Program B read the same record from the data base and each modifies its own copy.
- 2) Program A locks the data base, re-reads the record, checks it for changes since the first read, calls DBUPD, and unlocks the data base. Note that DBUPD does post the DCB contents to disc.
- 3) Program B now locks the data base and re-reads the record, but this read is from the DCB (not the disc!) and it returns the original data. Program B now checks the record, calls DBUPD, and unlocks the data base.
- 4) The data base on disc now reflects Program B's update. Program A's update has been lost. Even worse, if Program A re-reads the updated record it will get its own update data and not what is now on disc!

The "official" workaround is to call DBCLOS for each of the data sets which will be affected by the DBPUT or DBUPD before performing the operation:

- 1) Read the record (into program buffer A) and modify it (into program buffer B).

IMAGE/1000: Secrets HP Never Told You

- 2) Lock the data base and call DBCLOS for each data set which will be affected.
- 3) Re-read the record (into program buffer C), which will re-open the data set (and its DCB) and cause a physical dtsc read.
- 4) Check the record for any changes from the original read.
- 5) Call DBPUT or DBUPD.
- 6) Unlock the data base.

Of course, all of the data set closes and re-opens will vastly increase the time required for a simple DBPUT or DBUPD. And even this doesn't guarantee data record integrity; we have observed mysterious cases of duplicate DBPUTs and incorrect DBUPDs even with this technique.

A BETTER SOLUTION

More than ten years ago, HP encountered a similar problem when the FMGR was added to RTE: maintaining the integrity of the disc directory structure in a multi-program environment. The solution, which is still in use in RTE-A today, was to separate the directory manager (D.RTR) from the rest of the FMGR file system.

The FMGR file system and the program D.RTR are an excellent example of a "requester/server" software structure. User programs (and, usually, RTE processes) never access the disc directories themselves, but through "calls" to D.RTR. The user programs (and RTE) are "requesters" and D.RTR is a "server."

There are several advantages to this structure:

- A single server can service any number of requesters, through request queues maintained by RTE.
- The server automatically guarantees data consistency among all requester processes.
- The server can be very carefully coded for maximum efficiency and extensive error handling.
- The requester/server relationship provides a safe and uniform programming environment for users.
- Critical code exists only in the server, rather than being duplicated in each program, thus enhancing security and conserving valuable user program space.

IMAGE/1000: Secrets HP Never Told You

- Critical data structures (disc directories, for example) are isolated from direct access by poorly coded user programs.

There are, of course, some disadvantages to the requester/server structure. Probably the biggest disadvantage is that all requester calls are queued through a single server, which becomes the small end of the funnel. A secondary problem with the technique is that a server abort affects all of its requesters, which can have severe consequences. But overall, for most applications the advantages far outweigh the disadvantages.

AN IMAGE SERVER

We developed and began using a requester/server structure for our IMAGE programs in 1979, primarily to overcome two problems:

- 1) The 8-user limit imposed by the FMGR file system.
- 2) The problem of data record integrity with DBPUT, DBUPD, and DBDEL.

HP actually showed us the way in an IMAGE sales brochure from the late 1970s entitled (we think) "IMAGE/1000 Performance Brief." Unfortunately, our copy of this seminal brochure has been lost and HP couldn't find one for us to include in this presentation. The program structure described in HP's brochure and adapted by us consists of a single data base server (which we named MSDBA) and any number of user programs. Communication between the user programs and MSDBA uses Class I/O and Resource Number locking.

Our program MSDBA, which is coded for maximum efficiency and includes extensive error handling in addition to IMAGE's, performs five basic functions:

- 1) At startup: initialize and open the data base. Place "public" Class and Resource Numbers in a system common area.
- 2) Perform integrity checks on the data base and report its status.
- 3) Receive IMAGE transactions from requester programs, perform the desired operation, and return the results.
- 4) Maintain data base activity statistics for integrity checking and performance evaluation.
- 5) At shutdown: carefully close the data base, release Class and Resource Numbers, and flag system common.

IMAGE/1000: Secrets HP Never Told You

MSDBA is compact (19 pages, including buffer space) and fast, and includes most of the tricks and techniques described in this presentation. MSDBA consists of a main segment and two overlays: the first overlay contains the IMAGE DBOPN and DBCLS calls, as well as the open/close integrity checking and error reporting code, and the second overlay contains the code for interprocess communications, handling IMAGE requests, error checking, and generation of activity statistics. With this partitioning, overlay loads are required only at startup and shutdown, and we don't waste space by keeping DBOPN and DBCLS in memory all the time.

Since all DBPUTs and DBUPDs to the data base are through MSDBA, and since MSDBA has only one set of DCBs, we avoid the overhead of repeatedly closing data sets and there is never a problem maintaining the integrity of data record contents. Note that we only do "soft" deletes, as described earlier.

Even though MSDBA is normally the only program accessing the data base, we still open the data base with IMODE = 1. We have measured less than 10% overhead on writes to the data base using the shared mode rather than an exclusive open, and the shared mode allows simultaneous access by QUERY (be careful now!) whenever necessary.

IMAGE REQUESTER LINKAGE

Requester programs are coded to call our own linkage routines rather than calling IMAGE library routines directly. These linkage routines are quite compact (less than one page total), yielding MUCH smaller programs than with the IMAGE routines. As stated earlier, coordination and communication between requester programs and MSDBA is via Class I/O and Resource Number locking:

- 1) The requester program gets the "public" Class and Resource Numbers from system common, and also obtains a "private" class number from RTE for MSDBA's reply.
- 2) The requester linkage routine formats the request buffer and then attempts to lock the Resource Number (global lock, queued, with wait).
- 3) When the RN lock is granted, the requester does a Class Write/Read of the request buffer using the "public" Class Number, and immediately issues a Class Get request on its own "private" Class Number to await MSDBA's reply.
- 4) MSDBA, which has been suspended by a Class Get on the "public" Class Number, receives the request buffer and immediately unlocks the Resource Number. (The RN coordination ensures that only one request buffer can be in SAM at any given time.)

- 5) MSDBA performs the requested operation on the data base (using standard IMAGE calls), does a Class Write/Read of the results using the requesters "private" Class Number, and immediately issues another Class Get against the "public" Class Number.
- 6) The Class Get previously issued by the requester against its "private" Class Number completes, and the linkage routine passes the data from MSDBA back to the requester program.

As you can see, requests to MSDBA are efficiently overlapped even though only one request buffer at a time is present in SAM, and since most IMAGE calls involve disc access, there is plenty of CPU time available to manage the Class I/O request overhead.

IMAGE REQUESTER/SERVER RESULTS

You really have to try this requester/server technique to realize just how powerful and efficient it is! By far the most significant achievement is that any number of user programs can access the data base simultaneously, while maintaining the same data base integrity as if there were only a single program. And as a bonus, programs are much smaller and somewhat easier to maintain.

It is also worth noting that everything we have described so far about our IMAGE requester/server is legal and fully supported by HP. If you don't try any of our other IMAGE tricks, try this one!

We have been using this technique on various real-world systems, under RTE-4, RTE-6, RTE-A.1, and RTE-A, since 1979 with excellent results. Our client with the 348 Mb RTE-A IMAGE data base on an A600 system has been running our IMAGE server since 1982. This is a very active production system (the client has NO technical staff), with typically forty CRT terminals accessing the data base at any given time. In the five year life of this system we have experienced a total of ONE broken chain incident, caused by a supervisor who panicked and pressed the "BREAK" key before shutting down the data base. (Our IMAGE server's own integrity checks caught the error on restart, and we were able to carefully patch the broken chains in about two hours. Yes, they DID have a good backup, but a reload on this system would have taken more than ten days!)

Programs using a simple IMAGE requester/server may be somewhat faster or slower than programs coded with direct IMAGE calls. The bottom line is that you probably won't see much performance difference using a requester/server structure. While a small number of programs may run slightly faster, DCB and/or disc directory thrashing may slow the server when a large number of requester programs are active. Balanced against that, there may be less swapping overhead since

IMAGE/1000: Secrets HP Never Told You

requesters are smaller than equivalent "plain IMAGE" programs. And, of course, without the server you couldn't have more than eight programs accessing the data base anyway!

We have even experimented with multiple servers, based on techniques used successfully on Tandem Computers systems. While multiple servers should be faster, even with just one CPU, the performance improvements we observed weren't worth the other problems we created (those pesky DCBs again!).

But stick with us! We have developed a number of other techniques to speed your IMAGE application, and they are particularly effective in a requester/server environment.

INCREASING IMAGE PERFORMANCE

As noted earlier, rapid random retrieval of data is one of the primary reasons to use a data base. For the record, we really aren't too concerned with the speed of writes to the data base. Data published by Tandem Computers, covering a wide range of computer systems and applications, indicates that in a data base there are typically 20 reads performed for each write. Thus, when we talk about increasing the performance of an IMAGE data base, we need to expend most of our efforts to decrease the read (DBFND, DBGET) times.

As with most data base systems, physical access to the data base on disc is the most important factor in determining IMAGE performance. While a faster CPU will help a little, speeding up IMAGE is primarily a matter of reducing disc access times.

When we examine the speed of HP discs (and almost all other discs) we find that seek time (moving the heads from track to track) is the most significant variable. The potential for improvement is significant: we once encountered an application in which we more than DOUBLED the throughput by simply rearranging the physical allocation of disc space. If we can reduce head motion we can speed up IMAGE, it's that simple!

Reduced head motion is one of the primary reasons that IMAGE can be faster than IMAGE-II. The older FMGR file system allows you to control where your data sets physically reside on disc, and data sets created with DBDS or MDBDS are guaranteed to occupy contiguous disc tracks. With the newer hierarchical file system it is difficult to control the physical location of files, and IMAGE-II data sets will typically be fragmented and scattered all over the disc.

As you will see, most of our proven techniques for improving IMAGE performance either reduce the frequency of physical disc accesses or attempt to minimize disc head motion:

- Placing the data base on its own disc drive and/or interface.

IMAGE/1000: Secrets HP Never Told You

- Reducing directory thrashing by increasing the number of data control blocks (DCBs) IMAGE builds in your program.
- Intelligently selecting the physical location of data sets on the disc.
- Performing serial reads outside of IMAGE, using large DCBs.
- Performing smarter chained reads.

USING A SEPARATE DISC DRIVE FOR YOUR DATA BASE

In our experience, allocating a separate disc drive and interface to the IMAGE data base will result in substantial performance improvements.

Placing your IMAGE data base on a separate disc drive will almost always yield a good speed improvement. Not only is head motion reduced because it is no longer shared with other system activities, but seeks on your IMAGE disc can occur concurrently with seeks on the other system disc(s).

The MAC discs and controllers can transfer data to and/or from two disc drives simultaneously, at slightly reduced transfer rates. But while more than one CS/80 disc drive can be connected to a single HPIB disc interface card on A-Series systems, the interface can handle only one disc data transfer at a time. Adding a separate HPIB interface for your IMAGE disc drive will allow full-speed transfers concurrent with I/O to other discs.

These improvements are significant enough that we almost always recommend both a separate disc drive and interface for your data base on busy RTE-A systems. Your users will notice the difference!

ELIMINATING DIRECTORY THRASHING

We have always had problems with directory thrashing in IMAGE, and they became worse when we began using our server.

When a program opens an IMAGE data base, the DBOPN routine builds a number of data control blocks in free memory space at the end of your program. DBOPN builds these DCBs with a 256 word (2 block) buffer, and the number of DCBs created is equal to the largest path count in the data base plus one, up to a maximum of sixteen DCBs per data base. Since IMAGE uses the standard FMGR file system routines, access to a data set will always involve one of these DCBs. Unfortunately, if the number of DCBs built by DBOPN is less than the number of data sets in your data base, the DCBs will have to be "recycled" each time your program needs to access a data set not currently open.

This arrangement would probably be satisfactory if all programs were like QUERY, but production IMAGE programs seldom are. The result is a lot of disc directory thrashing as data sets are repeatedly opened, closed, and opened again. Throughout the history of IMAGE, HP has employed several algorithms for recycling DCBs but the basic problem remains: data set opens and closes are high overhead functions in RTE (lots of calls to D.RTR and disc seeks to the directory), and they slow down your IMAGE application. And worst of all, the overhead grows geometrically as data base activity increases.

Clearly, what we need is more DCBs (ideally one DCB per data set), rather than a better recycling algorithm. Repeated requests to HP over several years for 1) a fix, 2) a workaround, 3) an enhancement, and finally 4) access to IMAGE source code to fix it ourselves were to no avail.

Finally the solution dawned: we just trick IMAGE into building more DCBs. AND IT IS SO SIMPLE!!! Even better, there are at least two ways to do it, and one of them is fully supported by HP and requires no programming at all!

Trick #1: Use dummy data set declarations in the schema -

Since IMAGE builds DCBs based on largest path count, all we need to do is to modify our schema and add dummy data sets which will result in a path count equal to the number of DCBs we want minus one! As an IMAGE data base can have up to 50 data sets and most real world data bases have far fewer than that, there should be plenty of spares to use.

Here is a sample schema addition to obtain (n + 1) DCBs:

```
ITEMS: DCBFIX, X2 (1,1);

SETS:  NAME:      DCBKEY::13, A;
       ENTRY:    DCBFIX(n);
       CAPACITY: 1;

       NAME:      DCB1::13, D;
       ENTRY:    DCBFIX (DCBKEY);
       CAPACITY: 1;

       NAME:      DCB2::13, D;
       ENTRY:    DCBFIX (DCBKEY);
       CAPACITY: 1;
       .
       .
       .
       NAME:      DCBn::13, D;
       ENTRY:    DCBFIX (DCBKEY);
       CAPACITY: 1;
```

Remember that you can get up to 16 DCBs this way, no more. Be sure to place your dummy ITEM and SET declarations AFTER all other "real" declarations, so that the existing ITEM and SET numbers won't be changed. This trick does the job, takes up minimal space, and is HP supported. And if you aren't concerned about what HP thinks, you don't even have to build the "trick" data sets as long as your programs never reference them.

Trick #2: Patch the root file -

WARNING: This trick is not HP supported.

After discovering trick #1, an examination of the IMAGE root file disclosed an even simpler way to get more DCBs. There it is in word 4 of the root file: the number of DCBs for DBOPN to build!

All you have to do is patch word 4 of the root file to the number of DCBs you want. (I wonder what happens if you use try a number greater than 16 ... we've never tried it.) You can use CMMA or write your own 6-line FORTRAN program, but please shut down your data base and back up the root file first!!!

Besides being easier, patching the root file is really a better solution than trick #1 since it doesn't create any extraneous data sets and can be performed on an existing data base without having to do a reload. We have used this technique successfully on several very busy data bases and have observed a measurable improvement in the speed of our servers each time, with no problems whatsoever.

Of course, with either method you should size up your IMAGE programs (or your server), using LINK's "SZ" command, to increase the free space available for the DCBs. It's neat to do a "DL" of the data base cartridge and see all of the data sets open at once. Try it yourself. We think you'll like it. And we think HP should support it.

PHYSICAL ARRANGEMENT OF YOUR DATA SETS

DBDS creates your IMAGE data sets in the order in which they are declared in your schema, which is logical enough. But the default order is almost certainly not the best arrangement from the standpoint of minimizing disc head motion.

Think about your application. You probably have a few primary data sets which are accessed very frequently, and a number of secondary data sets which are accessed only occasionally (e.g., comment or annotation records). If one of your secondary data sets lies between two very active sets, you are wasting valuable time every time the disc heads seek across your secondary set.

As an example, our client's 348 Mb RTE-A data base includes a 223 Mb secondary data set containing compressed text data. For a long time we

IMAGE/1000: Secrets HP Never Told You

left this huge data set where DBDS would have created it, in the physical middle of the data base with some very active data sets on either side. It doesn't take a genius to realize how much disc time we were wasting. But simply rearranging the data sets resulted in a big improvement.

This is one reason our utility MDBDS (on the swap tape) is written to build only one data set at a time: to allow you to choose the arrangement of your data sets. But how can you determine the best arrangement? There are two ways.

The simplest and easiest method is TLAR ("that looks about right"), based on your understanding of the data base and the nature of the data base transactions in your application. The general rules are:

- The location of large data sets is more critical than small ones.
- Locate what you expect to be the most frequently accessed data sets near each other.
- Try to locate detail data sets near their related master sets containing the most frequently used keys.
- Locate large data sets away from the more active areas.
- Locate less active data sets (especially large ones) at the "edges" of the disc cartridge.

A more sophisticated optimization method is to let your programs generate data set activity statistics for you. You can code each program to include a simple one dimensional table of accesses to each data set and store that data to a file when it closes the data base. A slightly more complex method is to maintain a two dimensional table of previous data set accessed vs. current data set being accessed, yielding not only set access frequency data but also "from/to" data on the frequency of seeks (head motion!) between data sets. Of course, this technique becomes much more practical if you are using a single data base server.

Large, very active data bases will benefit most from disc optimization, but you will see improvements in almost any application. Try it; it's worth the effort!

PERFORMING YOUR OWN SERIAL READS

IMAGE serial reads (DBGETs with IMODE = 2 or 3) are among the worst system hogs known to HP users. If you have a large data base, doing your own serial reads will be much faster than IMAGE and will inflict less performance degradation on other users of the system.

IMAGE/1000: Secrets HP Never Told You

Although IMAGE serial reads tend to be both disc and CPU intensive, the use of 256-word DCB buffers is the primary problem. 256 words is a good DCB buffer size for chained or directed reads, which tend to be random, but it is way too small for most serial reads.

Fortunately, since IMAGE data sets are implemented as FMGR type 2 files, we can safely perform our own serial reads on them using FMGR calls. And if we use large DCB buffers when we open the data sets, we can perform our serial reads with far fewer physical disc accesses than IMAGE requires.

The general procedure is:

- 1) Open the data set file (using a large DCB buffer), save the first 16 words of the DCB, close the file, and restore the first 16 words to the DCB.
- 2) Lower our priority to be polite.
- 3) Read the file until we find a record of interest.
- 4) Reset our priority to its original value.
- 5) Re-read the record of interest using DBGET.

The data returned by our own reads will consist of the IMAGE media record concatenated with the full data record. The media record is documented in the IMAGE manual, and you can get the sizes of the media records in your data base from your DBDS listing. EQUIVALENCE statements in your FORTRAN programs will allow you to painlessly reference any desired data item in the record.

Here is a rather generalized code example (no error checking!):

```
* Open file, save DCB, close file, and restore DCB
  IMODE = 1
  CALL OPEN(IDCBS,IERR,IDBSET,IMODE,ISECU,ICR,IDCBS)
  CALL MoveWords(IDCBS,IBUF,16)
  CALL ECLOS(IDCBS)
  CALL MoveWords(IBUF,IDCB,16)
* Set starting point of read
  NUM = ISTART
* Lower our priority
  PRI = GETPR()           ! get program priority
  100 CALL SETPR(PRI + 20) ! set lower priority
* Perform our own serial read until qualified record found
  200 CALL EREAD(IDCBS,IERR,IBUF,IL,LEN,NUM)
      IF(LEN.EQ.-1) THEN   ! end of file
        CALL SETPR(PRI)   ! (reset priority)
        GO TO xxxxx
```

```

ELSE IF (IBUF(1) .EQ. 0) THEN ! empty record
    NUM = NUM + 1
    GO TO 200
ELSE IF (record test) THEN ! qualified record
    CALL SETPR(PRI) ! (reset priority)
    GO TO 300
ELSE ! not qualified
    NUM = NUM + 1
    GO TO 200
ENDIF
* Re-read selected record with IMAGE
300 IMODE = 4
LIST = 2H@
IARG = NUM
CALL DBGET (IBASE, IDBSET, IMODE, ISTAT, LIST, IBUF, IARG)
IF (record test) THEN ! record we want
    GO TO yyy
ELSE ! don't want this one
    NUM = NUM + 1 ! continue the search
    GO TO 100
ENDIF

```

The old "save the DCB" trick which we used here is definitely not supported by HP, but we have been using it successfully since 1976! And it is completely safe as long as it is used only to read a data set. (This powerful technique can also be used in a lot of other ways, not all of them as safe as this.) The main reason we use it here is to avoid using valuable disc directory entries.

GETPR and SETPR are our own routines to get and set our program's priority. Since SETPR calls the RTE library routine IXPUT, it may or may not be supported depending on whether or not IXPUT is documented in this year's manuals.

A natural question is "How big should the DCB buffer be?" In general, based on our own testing on real-world systems, disc throughput will generally increase logarithmically with the size of the DCB buffer. And as the size of the DCB buffer increases, a program will slowly metamorphosize from a disc hog to a CPU hog! (That's why we drop the program's priority while doing our own serial read.) We recommend a DCB buffer size of 8, 12, 16, or 24 blocks (1024 to 3072 words). If you have more buffer space available, try it. Remember that the actual buffer size you use should be 1) a multiple of 128 word blocks, and 2) an even divisor of the file size; any extra will be wasted. As a corollary, avoid file sizes which are a prime number of blocks or, in the case of IMAGE data sets, two times a prime.

Finally, after finding a record with your own serial read, always perform a directed DBGET (IMODE = 4) to re-read it. This will return you to a pure (HP supported) IMAGE environment and ensure that IMAGE's internal structures point to the record you think they do!

IMAGE/1000: Secrets HP Never Told You

If you frequently perform IMAGE serial reads, and particularly if you have a large data base, try these techniques. You WILL see an improvement in performance, and your users will notice it too!

PERFORMING MORE EFFICIENT CHAINED READS

The chained record structure of IMAGE is a central characteristic of the network data base model. While the chained data structure can be very powerful, there is relatively little a user can do to improve the speed of chained reads since they tend to result in semi-random disc seeks. The techniques used to speed up serial reads won't work, but here are a few other suggestions which can speed up IMAGE chained reads for you:

Use backward chained reads -

When you perform a DBPUT, the new record is inserted at the foot of the related chains unless a chain is defined as sorted. A normal (forward) chained read (DBGET with IMODE = 5) begins at the head of the chain and works toward the foot, while a backward chained read (IMODE = 6) begins at the foot. Since most applications reference recent data much more frequently than older data, the backward chained read will tend to be faster at finding the records you want to retrieve.

Always use the shortest chain -

If you have more than one key item value, always do a DBFND on each key item and then perform your chained read using the shortest chain path. It sounds simple, but this can produce good performance improvements with large data bases, although the programming can get a little messy at times.

Use compound keys -

As your data base grows, the chains for non-unique keys will tend to get longer and longer, and the searches slower and slower. Compound keys can be obtained by concatenating key data values with some simple variable which changes slowly. Using these compound keys instead of just the data value will tend to produce several short chains instead of one very long one. For example, if your data base spans several years and it is reasonable to search it a year at a time, a compound key consisting of a key data value concatenated with a two digit year will yield more reasonable chain lengths and considerably faster searches.

Rearrange data sets -

Arrange data sets so that the most frequently used master data set is located on disc adjacent to its related detail set. If a master data set has two related detail sets, consider putting it between them. This can

result in a slight speed improvement, especially when chains are short.

POSSIBLE FUTURE PERFORMANCE IMPROVEMENTS

We hope we have described at least one technique in this presentation which you will find useful. But there is still considerable room for future IMAGE improvements. Below are a few ideas to ponder; I'd like to see YOUR enhancement presented at next year's conference!

Disc cache -

The most attractive potential performance enhancement we have looked at but have not tested is the use of a disc cache. And the route HP has taken with IMAGE-II, a separate data base cache, would be even better. (I can't imagine what a pig IMAGE-II would be without the cache!)

A disc cache can provide substantial improvement in speed just where we need it: in systems where there are lots of disc reads. A disc cache can exist either in software (a CPU resident cache) or in the disc controller hardware/firmware. Typically, a disc controller resident cache will be far less flexible than a software cache, although it will have the advantage of minimal CPU overhead.

Note that the cache option for the 7933/35 discs isn't any good for IMAGE applications unless you spend all of your time doing serial reads (or batch processing on a 3000 system). The cache option for the 7936/37 discs may do a much better job for IMAGE applications.

Some things to look for in a software cache for IMAGE are:

- User-selectable cache size.
- True write-through operation, to protect data integrity.
- Capability for IMAGE-only operation rather than for all disc accesses, or the capability to maintain separate caches for each disc volume. This would make the best use of the CS/80 disc controllers and minimize the contention between IMAGE and non-IMAGE disc accesses.
- A replacement algorithm based on "popularity" as well as "aging" criteria. If replacement is based on aging alone, a single serial read can flush the entire cache.

We don't know whether HP will ever offer a disc cache which will be useful with IMAGE, but we hope that we will soon see a good quality software-based cache offered by a good quality third-party vendor.

Has anybody tried this already? We'd like to hear from you.

Multiple servers -

We briefly mentioned earlier that we had tried multiple servers with only limited success. We still think that two or three synchronized servers for a single data base could yield a worthwhile improvement in IMAGE speed, since their disc waits and CPU usage would tend to become interleaved.

We have yet to overcome the same old problem of DCB buffering and data record integrity. The multiple servers we have tested worked well enough except for this one problem, but we discontinued testing because we were doing it on a live system and didn't want to damage the client's data base.

We still think this idea is worth pursuing. Are there any volunteers out there?

Requester/server communications without Class I/O -

We have always wondered how much overhead we incur using Class I/O for communication between our requesters and our server. Certainly, letting HP sell our clients an A900 would end our worrying about such things ... or would it?

The "Generic Software Bus" described by Frank Smith, John Campbell, and Brian Unger in their article in the March/April 1987 issue of Interex's "TC Interface" could be just the kind of solution we're talking about, but we haven't had time to look into it. How about somebody else giving it a try?

ACKNOWLEDGEMENTS

I would like to acknowledge the contribution of my former partner, Judith Skinner, who originally developed our IMAGE server design and collaborated on some of the other techniques I have described in this presentation. Besides being a prolific and innovative software designer, Ms. Skinner is one of the unsung heroes of the early years of the HP1000 International Users Group, prior to the formation of Interex. We all owe her our thanks.

Finally, I would like to thank my clients, whose IMAGE applications were the reasons that most of the techniques described here were developed. It is their loyal patronage over more than a decade that has made this presentation possible.

A DATABASE MANAGEMENT SUBSYSTEM FOR IMAGE/1000

Paul F. Gerwitz
Patrick C. Klier
Management Services Division
Eastman Kodak Company
1669 Lake Avenue
Rochester, NY 14650

INTRODUCTION

Hewlett-Packard's IMAGE/1000 product has been successfully utilized in many manufacturing applications in the Eastman Kodak Company. During the implementation of one large manufacturing application, IMAGE/1000 was extended to provide needed additional functionality and efficiencies through the development of a software package called the Data Base Manager (DBM).

The manufacturing application consisted of a network of 35 HP/1000 E-series systems which provided process monitoring and control, data collection, quality and product tracking functions. The major software components were RTE-6/vm, IMAGE/1000 and DS/1000-IV. The application design requirements included:

1. Data collected via sensors, programmable controllers or by operator input, must be stored on the local system for some fixed length of time.
2. Collected data will be summarized and forwarded to other systems in the network.
3. Data must be accessible from anywhere in the network with reasonable response time and be transparent to the user.
4. Critical data must be backed up without stopping the application.
5. The database related application programs should be designed to execute on any system in the network.
6. The database subsystem must not adversely affect the overall performance of the systems.

This paper will present the functions and structure of the DBM, how it is integrated into an application, detailed discussion of DBM Internals, and performance characteristics.

DBM FUNCTIONS

The DBM subsystem includes the following features:

1. **Transparent Access**

Application programs access the subsystem by calling IMAGE intrinsics. Programs are then loaded with either IMAGE libraries or DBM libraries. These libraries result in less code being appended to the program. In addition, the location of the database being accessed is unknown to the application, since the Database open uses a logical database name.

2. **Remote Database Access**

Remote access is also transparent to the user and is implemented using the DS/1000-IV Program-to-Program capability. Additional error processing and message sequence numbering capability is built-in to improve the reliability of the communications.

3. **Request Queuing**

Request queuing allows requests from remote nodes to be maintained in a queue until they can be processed by the local DBM subsystem. This prevents requests from being rejected when they cannot be immediately processed.

4. **Automatic Request Timeout**

The timeout capability provides remote request timeouts when an applications remote request has been sent, but no error or status information was returned within a given time window. This prevents application programs from waiting indefinitely for a response from a remote system or when some error condition has occurred on the remote node.

5. **Automatic Database Backup**

The backup capability provides functions to improve the recoverability of data in the event of a system or program failure. Backup is implemented by maintaining an exact duplicate of the primary database, either on the local or a remote system, and is accomplished in parallel with updates to the primary database.

6. **Remote Database Locking**

Remote database locking under DBM is more reliable than the equivalent capability under IMAGE. It provides both the intrinsic level of locking as well as a 'cooperative locking' capability for programs that require more strict control of a given database.

7. **Database Copy and Compare Utilities**

These utilities are primarily used to maintain the integrity of backup databases, but may be used outside of the DBM subsystem as well.

DBM SOFTWARE STRUCTURE

The Data Base Manager subsystem consists of user interface subroutine libraries and subsystem programs. Below are some definitions and a brief description of the programs and libraries.

Local Node: The computer system on which a database application program resides and executes.

Remote Node: Any computer system logically connected to the local node by communication facilities.

Originating Node: The node from which a database request is initiated.

Primary Database: The database that may be accessed by applications executing locally or remotely.

Secondary Database (backup database): A database that is an exact copy of a primary database. The secondary may also be accessed in read-only mode by application programs on the node local to the secondary.

DBM The DBM program processes all requests for databases under its control on its local node. Any number of DBM programs may reside on a single node. Each DBMxx program (cloned copy) controls up to 4 physical databases, which may be any combination of local primaries and read-only secondaries. DBM opens each database in shared read/write mode, thus allowing non-DBM applications access to the database (i.e., read-only report writing applications).

DBMC The Database Manager Controller manages a set of tables that contain the current state of the DBM subsystem on its node as well as resource and class numbers for application programs to access databases under DBM control.

DMOPR The DBM Operator Program provides the interactive access to the DBM subsystem. Its functions include initialization, modification, display of current subsystem status, recovery from program aborts and shutdown functions on the local node.

RDBM The Remote Database Manager is responsible for sending requests for databases located on remote nodes. It will communicate the request to the remote node, using DS/1000-IV software where the request can be processed by the DBMxx clone that is responsible for the database. There may be many RDBMx clones on a local node, controlling access for up to 4 remote databases each.

DMSL The Database Manager Slave program is the general DS/1000 slave or receiver program for the DBM subsystem. All requests from remote

nodes are processed through DMSL and consist of either 1) Database access requests from RDBM's, 2) Return status information from a DBM at another node to an application on the local node, 3) Backup requests from remote nodes, 4) Backup error status messages from remote nodes in the event of a backup failure or 5) initial handling of lock requests from remote nodes. DMSL routes each request to the appropriate processing program on its local system or in the case of remote lock requests begins the processing internally.

DBREM The program provides auxiliary functions to the DBM subsystem on local node. Any remote access to a local database will have status or actual data to return to the requesting application. DBREM functions as a DS/1000-IV master for sending of this data or status information, communicating with a corresponding DMSL program on the originating node.

DMQU The Data Management Queuing program will queue class I/O requests to a disc track to minimize the usage of SAM on a system. It is most useful when a DBMxx program cannot process remote requests as fast as they are received or when System Available Memory (SAM) is temporarily unavailable for requests. It is also used to queue data buffers being returned to an originating node by DBREM.

DMBK The DBM Backup program is used to send backup requests sent by the DBM that controls the primary database, passing these requests to the remote node where the backup database resides. The controlling DBM at the remote node then processes and completes the request to the backup database. It also has a queue of held backup requests in case the secondary node is down.

DMTO DMTO implements the remote timeout capability. It also monitors the local and remote cooperative locks that have been issued by application programs.

UPDBM The UPDBM program provides the timing intervals to DMTO. DMTO waits to be scheduled by UPDBM at the predetermined intervals set at subsystem initialization.

DMCMP The DBM Compare program compares two databases for equivalency. The program can process databases on remote nodes and can be executed from any node in the network.

DMCPY The DBM Copy program is used to copy all files of a database to those of a database with an equivalent schema. The source and destination databases may reside on the same node or different nodes, and the program can be executed on any node in the network.

DMTM The DBM Track Manager manages a pseudo 'Track Pool' on RTE-A systems for use by DBMC and DMQU. The program uses a file created on the

scratch FMGR cartridge and does direct EXEC read and writes to the tracks allocated to the file.

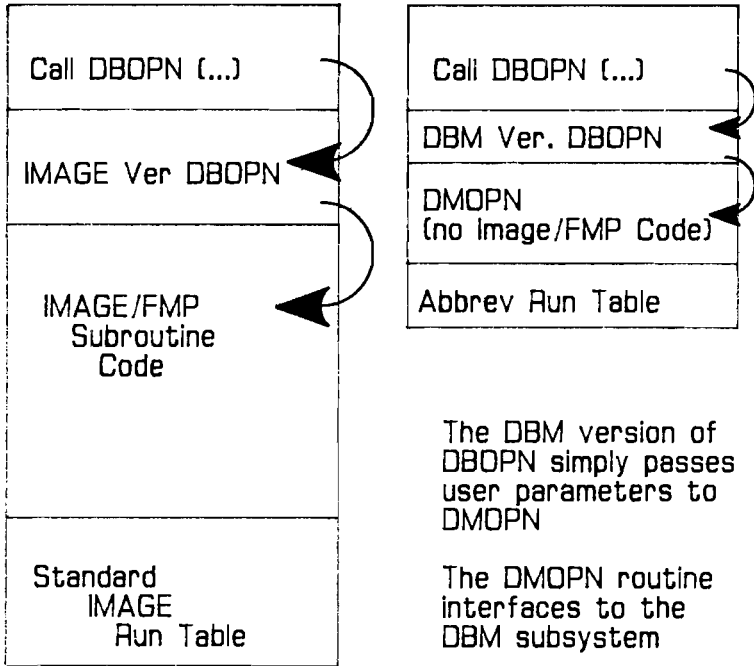
DBLIB The DB Library provides intercept routines for DMLIB. When called, the routine takes the parameters that were passed in and calls the appropriate routine from DMLIB.

DMLIB The DM Library provides the interface for an application program to access databases under DBM control. The user does not call the routines in this library directly, but uses the intercept routines in DBLIB at load time.

WRITING APPLICATION PROGRAMS

When writing application programs to access databases, IMAGE intrinsics are used. The program is then loaded with either the IMAGE libraries or the DBM libraries. This 'transparent access' capability relieves the programmer of determining the environment the program will run under. When executing under the DBM subsystem, intrinsic calls are intercepted by the DBM library subroutines appended to the program. These subroutines perform the functions of interfacing to the subsystem as well as managing the database 'run table' for the program. The run table contains the root file information (item and set names and numbers, privileges), the DCB area for each Data Set to be opened and a buffer where data is stored for reading and writing to the datasets. Application program run tables under DBM are smaller than what would be required by IMAGE. The IMAGE run tables are kept in the DBMxx clone for each database that is open. The interface between the application program and the library code, and the relative size differences, can be represented as shown in Figure 1.

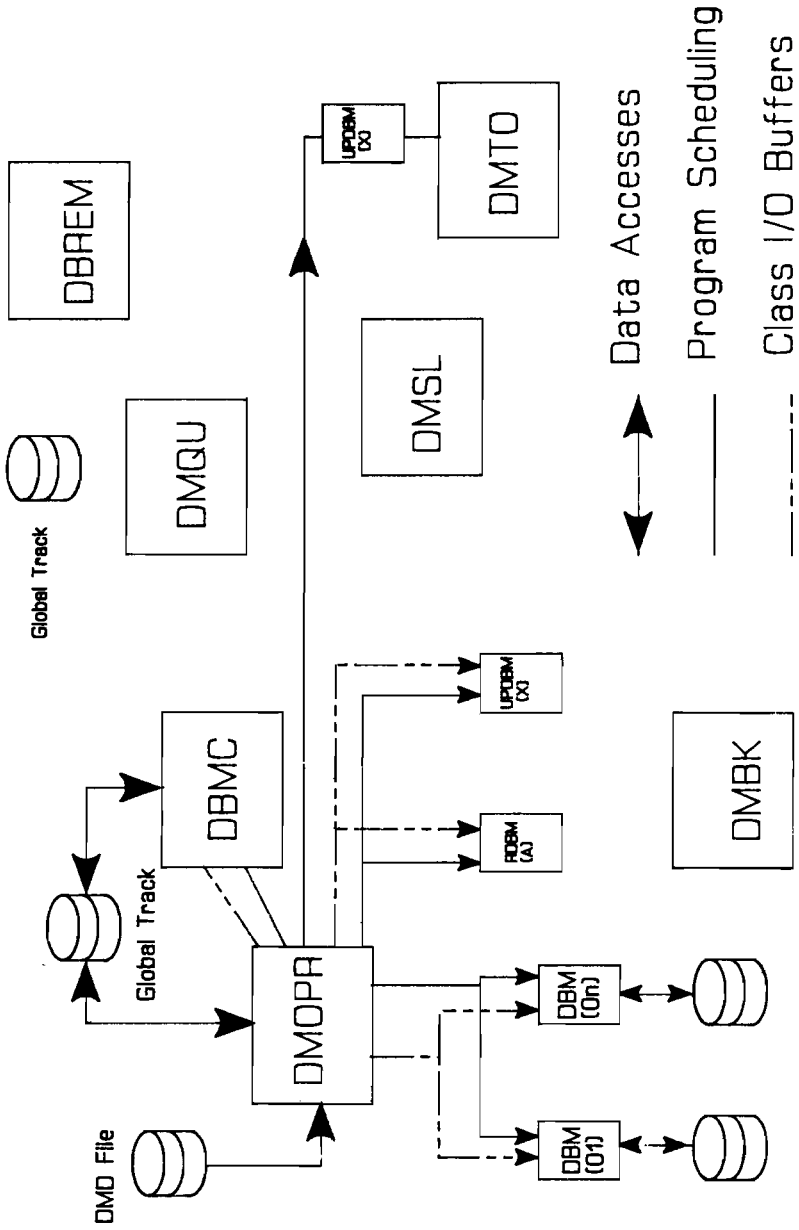
FIGURE 1



DBM SYSTEM INTEGRATION

In order to put a database under control of the DBM Subsystem, the user must create a Database Manager Definition (DMD) file. This is typically created by the system manager using EDIT. The DMD file relates a physical database name with a logical name, which will be used by the application program in the DBOPN call as well as specifying other characteristics of the subsystem for a given node. The format and several examples of the DMD file are found in Appendix A. The database must have been already created using the IMAGE DBDS utility. Initialization of the DBM subsystem is accomplished using the DMOPR program and may be done during system bootup or by a user at a terminal, see Figure 3. DMOPR begins by allocating a class number and sending it to DBMC via an EXEC schedule call. DBMC will then suspend on that class number waiting for DMOPR to complete the initialization. While DBMC is waiting on the class get, applications are prevented from opening databases. DMOPR then reads the DMD file and builds the configuration tables internally based on the DMD records. It will

FIGURE 3 - Initialization



clone the appropriate DBM and RDBM programs according to the '03' DBM/RDBM Descriptor Records, as well as allocating a class number and resource number for each. As each Database Descriptor Record (04) record is read from the DMD file, the physical database name is put in a buffer. When all these records are read, DMOPR schedules each DBM/RDBM clone, passing their class and resource numbers in the schedule call, and then sends the database names for each via class I/O. The UPDBM/DMTO Init Records are then decoded and the UPDBM program is started, passing the timing values to it in the schedule call. When all the DMD records are processed, the configuration tables are written to the disc track and a class buffer is passed to DBMC, indicating that the initialization is complete.

The DBM and RDBM clones are scheduled by DMOPR and given their class and resource numbers. The DBM clone will retrieve the database names from the class queue, open each database, and then wait for requests from applications on its class number. The RDBM clone will initialize itself and then wait for application requests sent on its class number. Any errors encountered during this process will be passed back to DMOPR via class I/O.

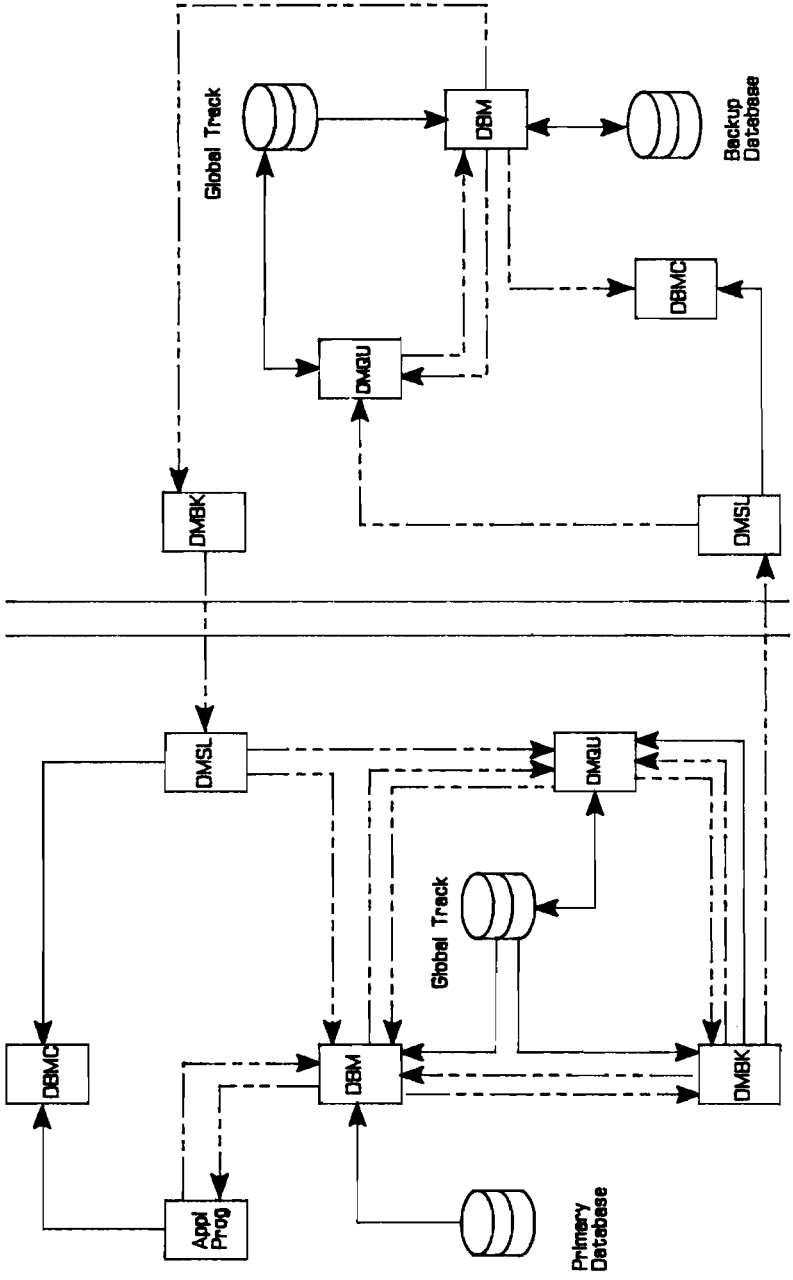
DBM SYSTEM OPERATION

LOCAL ACCESS

Local access under DBM provides the capability most often found in most systems. The process of opening a database, reading a record from a dataset and then closing the database is described below. Refer to the left side of Figure 4.

1. The user calls DBOPN to open the database. The database name is a logical name previously defined in the DMD file. The DMOPN subroutine is entered and a request is sent to DBMC to obtain the class and resource numbers of the DBM clone that is assigned to the physical database that is to be opened. DBMC will also mark the database as being opened to that application program.
2. The program attempts to lock the DBM's resource number. If the lock is successful, the open request is sent via class I/O to the DBM clone program which will process the request and return the run table information needed by the program on a class number assigned to the program. The DBM clone then unlocks the resource number so that other requests can be processed.
3. The application code retrieves the returned status information and builds the local run table in the free space of the program. The DBM routines then return the status information to the application program code.
4. The application program calls DBGET which enters DMGET. The request

FIGURE 4 - Local and Local with Backup



is validated against the run table list for the item names supplied in the call. If there is an error, the routines return immediately. If the item/set names are valid, a class buffer is built and a resource number lock is attempted to the DBM clone. When the lock is granted, the class buffer is sent to the DBM clone.

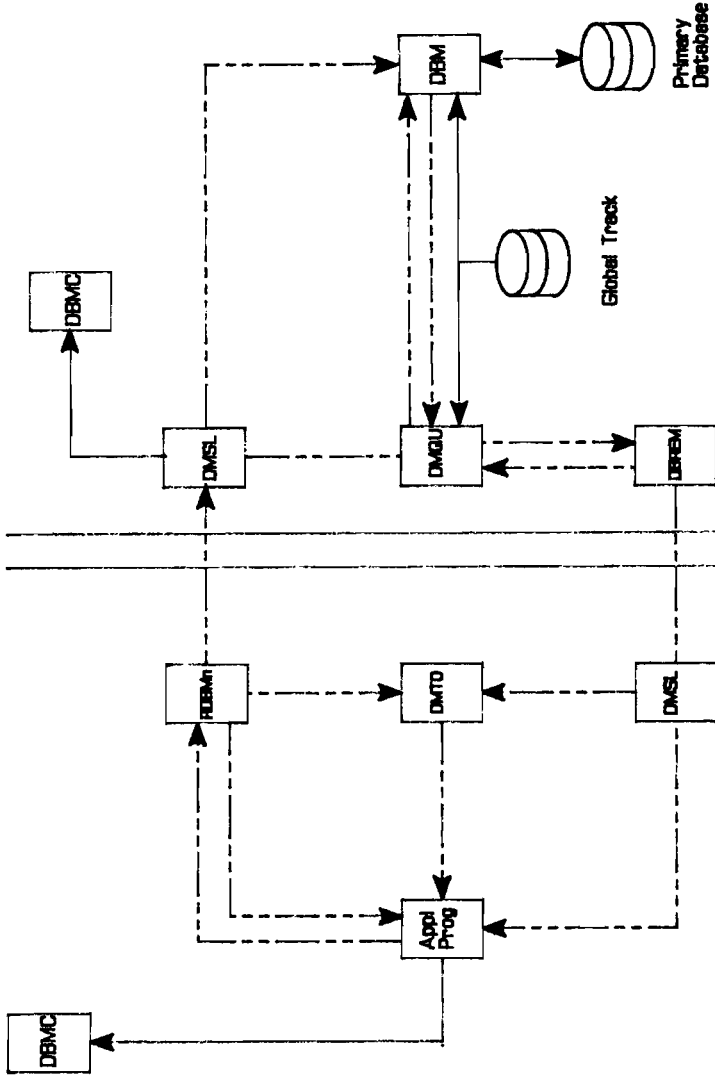
5. The DBM clone receives the request and processes it, passing the data back on the application program's class number. The data is received and passed back to the application program code.
6. The DBCLS call causes notification to DBMC to close the database and delete the entry in its tables. No additional requests are sent to the DBM clone program.

AUTOMATIC DATABASE BACKUP

One of the unique capabilities of the DBM subsystem is the automatic database backup function. Backup is defined as maintaining a mirror image copy of the primary database in another location. In most systems the backup database resides on a different node than the primary database, but could reside on another disc LU or even another physical disc on the same system. The backup function ensures that changes to the primary database (due to DBPUT, DBUPD, DBDEL) are performed on the backup database as well. The backup processing operates in parallel with the primary processing so that applications are not left waiting for a request to complete. It is also completely transparent to the application program. The processing for the backup capability is shown in Figure 4.

1. Upon the receipt of a DBPUT, DBDEL or DBUPD request from an application program, the DBM program forwards the request to the DMBK program to be sent to the node where the backup database resides. The DBM then processes the request to the primary database. The DBM will then wait with its resource number locked until a status is returned from DMBK indicating whether the backup request was successfully sent to the remote node. If the request was successfully sent, a class buffer is returned to DBM so that it may complete its processing. If an error did occur, such as a DS error or a Program-to-Program Reject, the request is forwarded to DMQU to be stored, and a completion status is returned to the DBM program so that it can complete the request to the application. Every 30 seconds DMBK will check the status of the remote node and reprocess the backup request stored in the disk queue until it is successful or until the queue is full. If the queue does become full, all backup requests to that node are flushed and DBMC is notified that the backup is down.
2. On the backup node, the request is received by DMSL and sent to the backup DBM via the DMQU program. When the request is forwarded to the DBM clone that controls the backup database, the request sequence number is checked, and if the request was received out of sequence,

FIGURE 5 - Remote Access



the backup is set down. If the sequence number is correct, the request is processed and the status is saved locally; no status is returned to the primary node. If the sequence error occurs, it will send back a response to the primary DBM upon receipt of the next backup request.

REMOTE ACCESS

Access to remote databases is similar to that of local request, except that the database is not physically located on the same node as the application. This access is transparent to the application program through the use of a logical database name. The processing of remote requests is described below, see Figure 5.

1. The open request causes a schedule of DBMC to obtain the class and resource numbers for the RDBM clone associated with the logical database specified by the application program. The request is then sent to the RDBM after locking its resource number. Subsequent requests are sent directly to the RDBM clone.
2. RDBM sends the request to the DMSL at the remote node where the physical database resides. The DMSL picks up the request and forwards it directly to the DBM clone or queues it DMQU if the DBM is busy.
3. The request is processed by the DBM program and the results are sent via DMQU to the DBREM program, which communicates them to the DMSL at the originating node.
4. The DMSL at the originating node forwards the results of the request back to the application program via its assigned class number.

When processing remote requests, the DMTO program is used to monitor the progress of each request and will timeout requests that take too long. This timeout capability prevents applications from waiting indefinitely for the completion of requests. When the request is sent to the remote node, DMTO is notified and puts an entry into its table. When the request is completed and the status is returned, the local DMSL will notify DMTO that the remote request is complete and to remove the table entry. If a request does not complete within the timeout window, the application is sent a timeout error. When the reply buffer finally arrives, an error is logged, and the data discarded. The timeout values are initially provided at initialization through the DMD file, but can be adjusted by rescheduling UPDBM.

DATABASE LOCKING

The locking and unlocking of databases by remote application programs presents some serious concerns which must be addressed in the network environment. Since all locks apply to the entire database, all other

application programs are prevented from modifying a database whenever an application program has a true lock on that database. If the application should abort before unlocking the database (or 'forget' to unlock it), all updates will be locked out until the problem is corrected by manual intervention. Thus, a problem at one node in a network may adversely affect many other nodes in the network.

True remote locking also affects the response time of programs performing database updates, since a lock request may have to wait for a remote program to unlock the database. Any program priority considerations at the remote node and DS overhead will increase the wait time. If numerous programs in the network are competing for updates, response times may degrade quickly. Programs which must execute within a predefined time window (such as those involved with machine control or data acquisition) may encounter serious timing problems.

The standard HP IMAGE interface always implements true locking for all local and remote accesses. This is necessary, since each program physically accesses the data sets, and dataset chain conflicts would occur if multiple programs attempted to add or delete records at the same time. Since the DBM subsystem restricts all updates to a database to a single DBMxx program, (which has a permanent 'true' lock on the database), this conflict can never occur, and true application locks are not required to maintain integrity of the data set chains.

For database access under DBM subsystem control, database locking is implemented by locking a resource number associated with the logical database name. Each node which accesses a given database will have its own local resource number for controlling both locking and unlocking. This restricts update access to no more than one program per node, but programs at different nodes cannot lock each other out using the standard lock and unlock calls, (programs on the same node ALWAYS cooperate, but programs on separate nodes do not). Since local resource number locks are used, the lock will be released if a program aborts and other programs at that node may now be granted the lock. This method prevents problems at one node from permanently 'locking out' other programs within the network. It does, however, allow certain undesirable conflicts to occur. Remote cooperative locking, as described below, was implemented to resolve those conflicts.

COOPERATIVE LOCKING

True application locking could be achieved under the DBM subsystem if all programs implemented cooperative locking. This case is highly undesirable because of the additional system overhead incurred and delays resulting from waiting for remote database unlocking. The concept of cooperative locking where needed allows potential conflicts to be avoided while minimizing overhead costs.

Cooperative locking means that cooperating programs may actually lock each other out of accessing a specified database, even though the programs reside on separate nodes. Programs from other nodes which are not cooperating may continue to perform updates to the database as is currently possible.

True application locking of a database accessed via the DBM subsystem is seldom if ever necessary. Many modifications which may be made to a database will never conflict with other modifications. Cooperative locking should be used whenever possible conflicts may occur. The following are examples of database modifications and descriptions of when cooperative locking is or is not needed:

1. DBLCK-DBPUT-DBUNL

Since this sequence allocates a new record from the free record chain, no prior information is needed and no conflicts between programs can arise. This sequence should NEVER need cooperative locking.

2. DBLCK-DBGET-DBDEL-DBUNL

DBGET-DBLCK-DBDEL-DBUNL

This sequence for a detail data set does not need a lock IF this program is the only program in the network which may delete this record (The second form of this sequence is improper unless this condition is true.) This is because a record of a detail data set will never be relocated. For a manual master data set, however, a record may move if it is a synonym entry and the associated primary entry is deleted. This sequence for a manual master data set should therefore use cooperative locking unless this program is the ONLY program which may delete any entries from that data set.

3. DBLCK-DBGET-DBUPD-DBUNL

Cooperative locking is needed for this case UNLESS the data being updated will never be updated by another program in the network (although updates by other programs to fields not specified for this call are acceptable) and the record will not be deleted during the execution of this sequence. Design considerations sometimes result in these conditions being true. For those situations where conflicts may occur, cooperative locking should be used to eliminate any conflicts.

IMPLEMENTATION OF COOPERATIVE LOCKING

The implementation of remote cooperative locking is achieved by using the DMTO program to keep track of the locks for its node. When a request is received from a remote node that includes a lock, DMSL notifies DMTO, which records information in a table entry and attempts to obtain the lock, returning the status of the lock to DMSL. If the lock was successful, the request is forwarded to the DBM program managing the database and

processed. If the lock is not successful, DMSL sends the request to the DMQU with a special 'HOLD' status until the lock can be obtained.

As DMTO scans the tables, it also checks for locks that are taking longer than expected. It will check with the DBMC program at the originating node to be sure the program is still running. If a lock is kept too long, it will release the lock automatically and the remote application program will be accessing the database in an unlocked fashion. When a program sends an unlock request, the entry is deleted from the DMTO tables and other lock requests are processed.

UTILITY FUNCTIONS

The DBM has several utility functions to aid in using the backup capability, for comparing two databases and copying one database to another. DMCMP can be run in the background to compare two equivalent databases, and the databases remain accessible during the compare. Databases are equivalent if they are identical except for the root file and dataset names, security codes and cartridge reference numbers. The item lists, dataset record formats and dataset capacities must be identical. The utility can be executed on any node, even if the databases are not local. If the comparison shows the databases to be different, the DMCPY program can be used to copy the primary database to the secondary to restore the equivalency. The databases involved in the copy must be closed, but the remainder of the subsystem continues to run. The copy operation takes from 10 to 30 minutes depending on the size and fullness of the database. It does not copy the IMAGE records individually, but opens the dataset as a Type 1 file and copies in 512-word blocks.

PERFORMANCE

When analyzing the performance of the DBM subsystem, the overriding consideration was to ensure that the subsystem did not adversely affect the performance of a given system. The subsystem was not designed to compete with IMAGE, but to provide added functionality without sacrificing reasonable system and network performance. That goal was met, with favorable results.

The performance measurements were made using a file driven program acquired from HP serving as a pseudo application program. This program executes a test procedure (defined by the user) which is contained in an ASCII file. Each command in the file specifies a given IMAGE operation (OPEN, CLOSE, GET, PUT, etc.) and the item names and data values for a given database. A test database was defined with a mix of master and detail datasets and item relationships that are typical of the application databases found in the network. The program was executed against this database, and timing and error status information was obtained. The initial testing proved valuable by providing status information about the subsystem's behavior, which in turn allowed some further tuning and

refinement of the code

When the timing data was analyzed, the DBM did not appear to significantly affect the overall performance of the system. In fact, for some operations (DBOPN and DBLCK) the DBM performed them faster (see Figure 6). The largest difference, about 3 to 1, occurs when opening a database. This can be accounted for, since IMAGE is physically opening the dataset files and incurring high overhead due to the file system. Since the DBM clone programs have already opened the database files at initialization, much of this overhead is eliminated for application programs using the DBM. The difference in timing for DBLCK is also a function of the file system, as IMAGE locks a database by locking the root file, while the DBM simply locks a resource number for that database. The DBINF calls are not included in the analysis because they are processed within the application subroutine code and do not interact with the operating system.

FIGURE 6

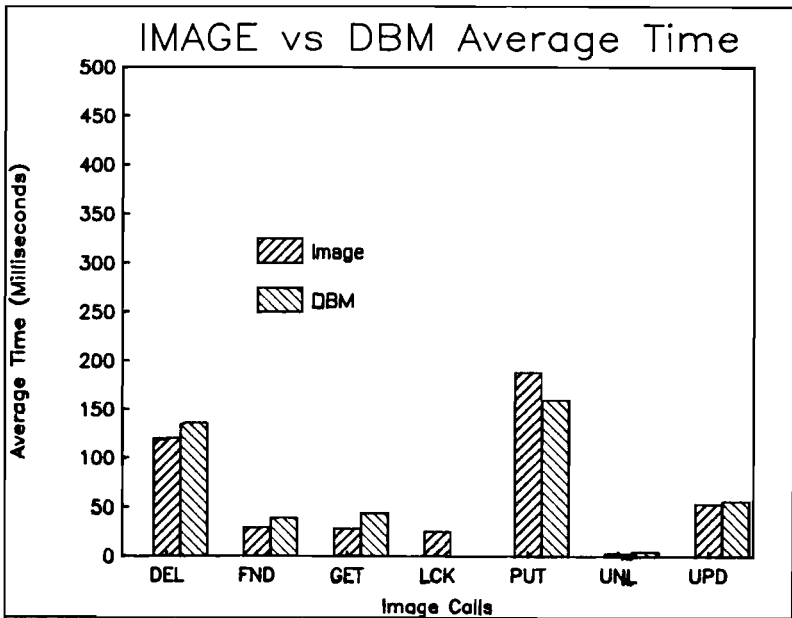
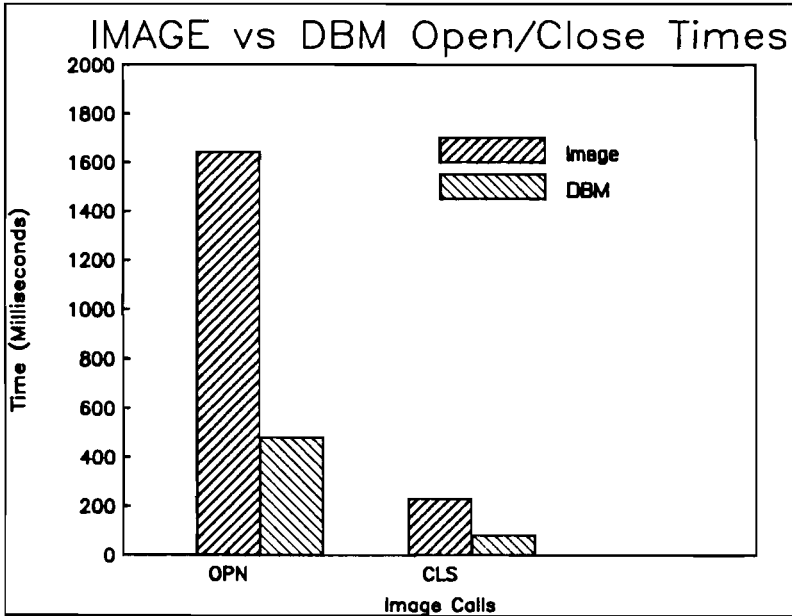


FIGURE 6 Continued



Performance comparisons between DBM and Remote IMAGE have not yet been made. We feel that the performance differences between Remote IMAGE and DBM will not be significant, since both are implemented in a similar fashion. If any differences exist, the DBM would take slightly longer to process requests, due to the remote queuing and multiple programs involved in handling the requests.

An additional consideration in the area of performance is the amount of library code that is needed in applications programs. Recall that application programs loaded with DBM require far less subroutine code and also a smaller run table. Typical savings range from 5 to 10 pages in code space and 1 to 3 pages for run table, depending on the database being accessed. These savings add up to making the job of segmentation and implementation of programs simpler and more reliable.

THE FUTURE OF DBM

Currently the subsystem is implemented in two networks. The subsystem will execute under RTE-6/vm with complete functionality. Applications programs can execute on RTE-A as long as the database resides on an RTE-6/vm system. Several additional changes must be made to enable the entire subsystem to execute on RTE-A, namely the Disc Queuing Program (DMQU).

There was also some consideration given to porting the subsystem to run under IMAGE-II. This was ruled out due to the difficulty in integrating DBM functionality with that offered in IMAGE-II. The central monitor, DBMON, replicates some of the functionality provided by the DBM program. There is concern that the logging capabilities could significantly affect performance as well. The DBM could continue to be a viable subsystem, if IMAGE/1000 was enhanced to provide new file system support and other relevant enhancements such as dataset locking.

CONCLUSION

The Data Base Manager subsystem provides several extensions to IMAGE/1000 that improve the efficiency and reliability of manufacturing applications. These include:

1. Transparent access to applications programs by providing interface libraries and logical database names.
2. Less appended code for application programs thus reducing the memory requirements when executing the programs.
3. A more robust remote access capability transparent to the application program including timeout processing for remote requests.
4. Queuing of remote requests to minimize errors being returned to application programs, thus using the system resources more effectively.
5. Automatic database backup for all update-type accesses to the primary database.
6. Database compare and copy utilities to ensure reliability of the backup databases.
7. Database locking capabilities that exceed those offered through IMAGE.
8. Configuration and control of the subsystem that are independent of the application programs running under DBM.
9. No significant impact on performance on systems where the DBM is running.

ACKNOWLEDGEMENTS

The authors would like to acknowledge and thank Don Richards and Paul R. Gerwitz, Sr. for the many hours and countless effort that they expended to design and implement the DBM subsystem. Their dedication to excellence during the entire life of the DBM project is born out in the quality of the software and the huge impact it has had on the success of the applications where it is used.

APPENDIX A

Database Manager Definition (DMD) File

TYPE Record Type Description

01 Title and ID record.

user text

user text is appropriate comments or title information.

02 Comment record.

user text

03 DBM/RDBM Program Descriptor record.

nn status partition size

nn is the DBM/RDBM clone Program ID. For DBM's this is a two-character numeric to get DBM01, DBM02, etc. For RDBM's use a one-character letter to get RDBMA, RDBMB, etc.

status is a numeric status/option code. Must be '01' to clone and initialize the RDBM or DBM program.

partition is a numeric partition number to which this DBM or RDBM program is to be assigned. If zero, no assignment is made and the program will use any partitions which are big enough.

size is the program size for this DBMxx or RDBMx. If zero, the size of the DBM or RDBM will be used. The size determines the space available for IMAGE RUN tables and application request buffers.

04 Database Descriptor record.

ldbname dbstat pnode pdbname mn bnode bdbname

ldbname is the logical database name (6 char).

dbstat is a numeric database open status.

- 1 Open as primary database (no backup)
- 2 Open as backup database (read-only local)
- 17 Open as primary database (backup enabled)

pnode is the node number for the primary database.
(8224 [2 blanks] or -1 if local)

pdbname is the name of the primary database (20 char max). For

local databases this is the local IMAGE/1000 namr including security code and cartridge reference. In the case of a remote database, this is the logical database name and there must be a corresponding DMD entry at the remote node. NOTE: If this is a backup database for a primary database at this node or another node, the database namr is still specified as a 'primary' database to the DBM which has it open. It may be accessed by application programs in a 'read-only' state.

bnode is the node number for the backup database. NOTE: This parameter and those that follow are not specified for databases accessed via RDBM's or for backup ('read-only') databases. (-1 if local node).

bdbnamr is the namr of the backup database (20 char max).

05 Equivalent Database Descriptor record.

ldbname dbstat edbname nn

edbname is the name of the logical database (6 char) to which the specified logical database (ldbname) is to be equivalenced. All references to this database (ldbname) will be directed to the equivalent database (edbname). All application program open entries will be listed under the logical database name specified by the program.

06 UPDBM and DMTO initialization record.

scanfreq dslim oplim locklim

scanfreq is the time in seconds between scans of the DMTO internal table. Default is 5.

dslim is the time in seconds after which DMTO tests to see if the DS link is still up. Default is 45.

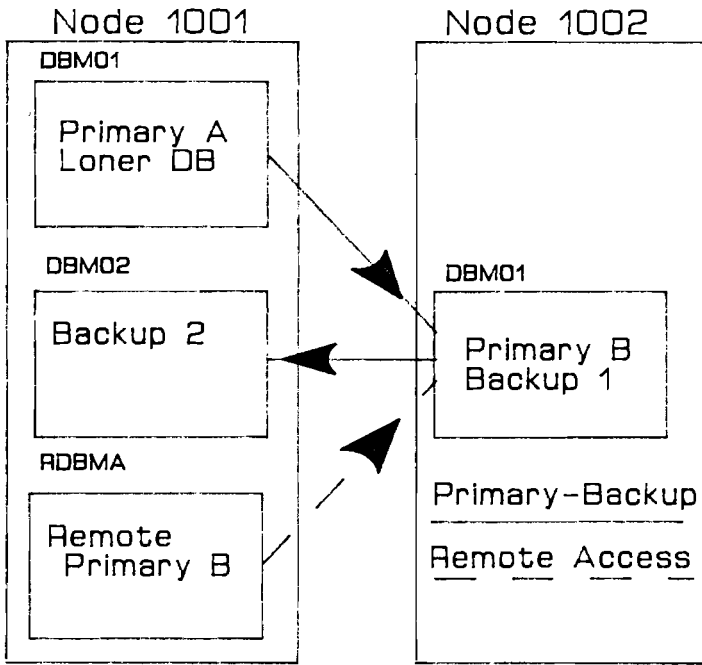
oplim is the time in seconds after which a remote request is "timed out." Default is 300.

locklim is the time in seconds for a cooperative lock check (schedules DBMC at node having DB lock to check if application is still active). Default is 15.

EXAMPLE

Referring to figure 2, consider the following example. The database structure can be viewed as follows with the corresponding DMD file entries. This example shows several different kinds of entries both, remote and backup.

FIGURE 2



DMD FILE

```
01 DATABASE DEFINITION FILE NODE 1001 <870901.1558>
01 *****
02 SPECIFY DBM/RDBM PROGRAMS
02 -----
03 01 01 13 19
03 02 01 14 18
03 A 01 15 00
02 SPECIFY DATA BASES
02 -----
04 >PRIMA 1 8224 >PRIMA:DB:DB; 01 1002 >BACK1:DB:DB
04 >LONER 1 8224 >LONER:DB:DB; 01 -1 >LONER
04 >BACK2 2 8224 >BACK2:DB:DB; 02
04 >REMOT 1 1002 >PRIME:DB:DB; A
02 TIMING VALUES
02 -----
06 10 60 300 60
```

```
01 DATABASE DEFINITION FILE FOR NODE 1002 <870901.1558>
01 *****
02 SPECIFY DBM/RDBM PROGRAMS
02 -----
03 01 01 00 00
02 SPECIFY DATA BASES
02 -----
04 >PRIMB 1 8224 >PRIMB:DB:DB; 01 1001 >BACK2:DB:DB
04 >BACK1 2 8224 >BACK1:DB:DB; 01
05 >EQVDB 1 >PRIMB 01
02 TIMING VALUES
02 -----
06 10 60 300 60
```

William D. Drotning

Thermophysical Properties Division 1824
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185

ABSTRACT

An examination of data analysis programs used on our Hewlett-Packard [2] 1000 system showed that many programs used a large fraction of code and programming effort to accomplish common tasks, such as file access and creation, data retrieval and storage, and the like. A clear need existed for a comprehensive, well-structured environment which would consolidate many previously-separated functions in a consistent, robust, easy-to-use format. Program DATA was written to accomplish a wide variety of tasks related to analysis of data stored in File Manager files on Hewlett-Packard HP 1000 minicomputer systems. As a data analysis tool, DATA provides a number of mathematical analysis and data manipulation functions. Flexible file structures, on both input and output, are allowed. An editor is included which allows DATA to serve as a binary file editor for data; search routines based on numeric, rather than string, values are incorporated. Also included are graphical plotting of data and functions, cursor selection of specific data regions, and other graphics features. DATA also serves as a shell operating system, with direct interface to the RTE operating system. Commonly-used file management functions are incorporated or enhanced for operator convenience. DATA is both command- and softkey-driven, with on-line help facilities and extensive error-checking. Finally, DATA serves as a common framework for future expansion with additional libraries or user-defined functions.

INTRODUCTION

In our laboratory, a number of instruments for measuring various thermophysical properties of materials are connected to a central HP 1000 minicomputer system. Typically, data are collected from the instruments in the multitasking environment and stored in disc files for subsequent analysis. These acquired and stored data may consist of a single vector of time-sequential dependent variables, such as a time-varying voltage signal, or multiple vectors of data from multiple sources. Depending on the device and the acquisition program, individual data records are acquired and stored in a variety of forms, from raw data (typically voltage) all the way to fully-analyzed data, such as specific heat versus temperature. A significant portion of the effort in the laboratory is spent on data reduction, analysis, and presentation.

Over time, the staff have written software as required to perform the various data reduction and analysis functions. In a typical reduction or analysis program, a data file is opened, records are read into the program, some analysis or reduction scheme operates on the data, and

frequently, an analyzed or reduced data file is created for storage to disc. As a result of this evolution, a large number of very similar programs have been developed, and program maintenance has become an increasing difficulty. Even finding the right program on the system became a problem, and depending on the programmer, each analysis program operated a bit differently. Yet, a substantial amount of the code in these programs is duplicated. A clear need existed for a comprehensive, well-structured environment which would consolidate many of these previously-separated functions into a consistent, robust, easy-to-use, single program format. In addition to reducing the number of programs to maintain, support and use, this universal approach has the potential of solving a headache of data file maintenance. With separate programs, the output of one analysis was frequently used as the input to another program, and the large number of temporary or intermediate data files created to pipe data from one application to the next can grow very rapidly. With a single, unified program, one could eliminate the need to store all the temporary data, saving an analyzed file only after all operations have been completed.

Program DATA was designed to accomplish a wide variety of tasks related to analysis of scientific data stored in files on Hewlett-Packard HP 1000 minicomputer systems. As a data analysis tool, DATA provides a number of often-used mathematical analysis and data manipulation functions. Flexible file structures, on both input and output, are allowed. An editor is included which allows DATA to serve as a binary file editor for data; search routines based on numeric, rather than string, values are incorporated. Graphical plotting of data and functions, cursor selection of specific data regions, and other graphics features are also included. DATA also serves as a shell operating system, with direct interface to the RTE operating system. Commonly-used file management functions are incorporated or enhanced for operator convenience. DATA is both command and menu (softkey) driven, and on-line help facilities are included. Finally, DATA serves as a common framework for future expansion, such as incorporation of the IMSL library or additional user-defined functions.

PRINCIPLE OF OPERATION

Data manipulation and processing is done through the use of four large internal arrays. In general, these arrays are used as input x and y arrays (where x and y refer to independent and dependent variables, respectively), and output x and y arrays; however, these arrays may also be used as interim working arrays in some applications. Once an input file is specified, the input and output arrays are filled with data from the input file, according to the current specifications for the input file structure and format. The input file is closed, and subsequent operations on the data occur on the arrays, not on the original input file, which remains intact. Unless otherwise indicated, all data processing, editing, and manipulation operations are done to the output arrays. To save the data currently stored in the output arrays, the operator must specify an output file, optionally define output header records, and actively write the array data into the file. The operator may also specify logging of the screen output in order to save a record of a particular analysis.

PROGRAM DESIGN

Program DATA was written as a main and several segments. Large internal data arrays are stored in EMA. For the majority of functions used in our laboratory, data records in files consist of an independent and a dependent variable, so that two data vectors can describe a file. When DATA is initiated, the current terminal softkey definitions are saved and subsequently written back to the terminal when a DATA session is finished. The relatively small main program is used primarily to process input commands. Each input command is stored in a circular list which provides a historical record of the command entries. Following case folding and parsing, the command is interpreted, and the appropriate segment is called to accomplish the specified function. The command processor distinguishes numeric from alphabetic commands; numeric commands are used to designate a particular data record, much like in EDIT/1000, while the latter define functions in DATA. Alphabetic commands consist of up to four characters, and, depending on the function, additional parameters.

Once a function is selected and the appropriate segment is loaded, a variety of input techniques were used in DATA. Rather than rigidly adhering to a fixed technique, we have used different techniques which we have found to be convenient and appropriate to the particular application. In some cases, blockmode entry of protected screen forms is used. A few applications prompt the user for input through a sequential series of questions, but wherever appropriate, user entry is done through defined softkeys. This is a favored approach in many cases, since it allows the user to see (rather than remember) the available options, and eliminates the tedious task of answering a lot of default questions. Softkeys work especially well for toggling logical switches and allowing the user to control the decision sequence.

DATA provides an on-line help facility in a "standard HP" manner: two question marks give an entire list of available commands, while two question marks followed by a specific command gives selective information for that command. A separately-maintained ASCII file contains the help information.

A number of "attributes" are maintained by DATA which may be viewed or modified by the user. These values are maintained in common blocks, and essentially define logical switches which control program flow through DATA, or define tolerances used by DATA, such as an equality tolerance for comparison of real values.

ENVIRONMENT

DATA is written in FTN4X as a main, several segments, and a library of subprograms. Source code is approximately 9000 lines, and the loaded program requires 26 pages, plus EMA. The current configuration of DATA allows its use on RTE systems which employ FMGR file and directory structures. Interactive terminals require an interface through a type 05 driver, and must be in the 262x or 264x families, capable of programmable screen, format, and softkey functions. Graphics terminals must be of the 2623A or 2627A types to use DATA's graphics functions.

DATA is used extensively on HP150 Touchscreen personal computers, either directly linked at 19.2 kbaud to the HP 1000 while emulating a 2623A terminal, or operated as personal computers and ported to the HP 1000 through the Reflection terminal emulation software [3]. EMA is employed for improved performance during array manipulations. Graphics/1000-II DGL software is used for the graphics routines.

DATA FORMAT

The format of input and output data files is flexible, and is controlled by user-definition through a screen form, shown in Fig. 1. Data in disc files may be accessed in either ASCII or binary form. For ASCII output, the user may specify a specific FORTRAN-legal format.

```

File Format                                     (DIM = 1500)

Input File
Ascii/Binary:           Data format: 
Number of header records (40A2): 
Number of records between header and data: 

Output File
Ascii/Binary:           Data format: 
ASCII format: (  )
Number of header records (40A2): 

```

	Data Record
	Format Key
	1 - (i)
	2 - (y)
	3 - (i,y)
	4 - (x,y)

Fig. 1. Screen form for data file format definition, command 'FORM'.

FUNCTIONAL GROUPS

The commands within DATA may be divided into the following functional groups:

DATA Configuration. This set of commands is used to set or interrogate the current configuration of the program. Included here are commands to alter the input or output file structure, declare input and output files, and display current file characteristics. A list of operator commands which have been entered is maintained for viewing. A number of DATA switches may be queried or set through the use of attributes.

Help. On-line help for each command, and a comprehensive list of currently available commands, is available interactively in DATA.

Graphics. A graphics system is available for CRT plotting of data and functions, as well as fits to the data, lines, and text labels. The graphics commands are softkey-selectable, and many of the commands allow interactive cursor control for input. Figs. 2 and 3 show the available softkey selections in the graphics section. Although scaling is done automatically, many scaling and tic mark choices may be changed by the user. Labels and lines may be added to provide additional information on hardcopy output. The graphics output is not meant to replace high quality presentation graphics, but only to provide fast CRT plots with some capabilities for line drawing and annotation.

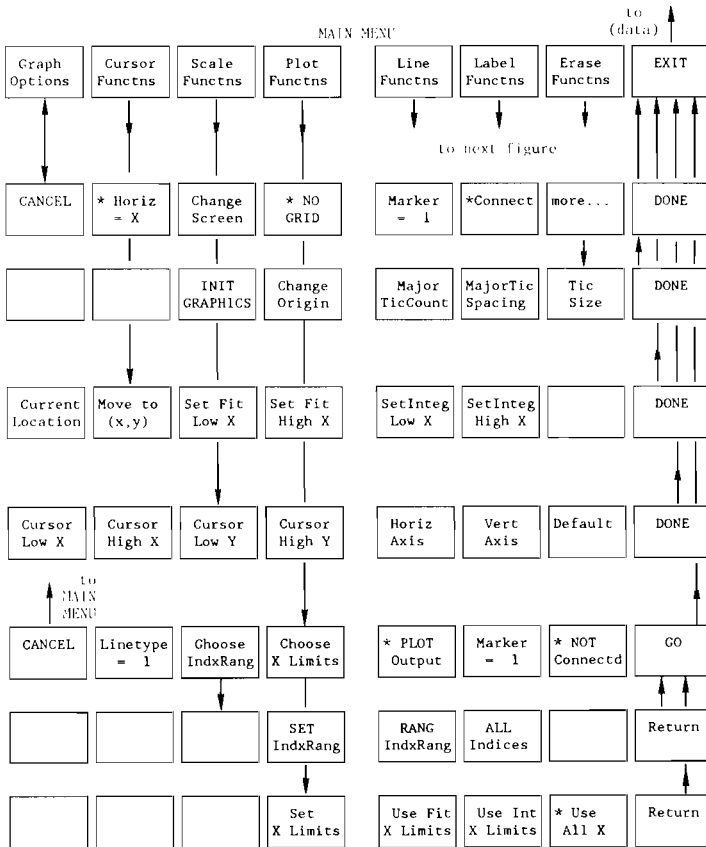


Fig. 2. Graphics softkey selection tree

COMMANDS BY FUNCTIONAL GROUP

Commands are entered following the DATA prompt "(data)"; each command must be followed by a carriage return. Lower or upper case may be used. Commands consist of up to four ASCII characters. (The use of integers or a simple carriage return are exceptions.) Some commands require additional parameters, separated by commas from the primary command. Other commands allow optional parameters or commands, also separated by commas. Filenames use the standard FMGR namr structure, with parameters separated by colons. Some commands invoke labelled softkey functions for additional menu-based operator interaction. Improper entries are reported to the user without serious effect on the DATA session, and extensive error-reporting is used throughout.

DATA Configuration.

ARRY -- Show number of values in input and output buffers
CNFG -- Show buffer dimension and directory limitations
DATA -- Same as ARRY
EDIT -- Set edit mode
EX -- Exit DATA
FILE -- Shows current input and output file information
FORM -- File format selection and display
HEAD -- Display/change output file header.
HIST -- Show history of commands (last 22 entered)
IFIL -- Specify input file
LOG -- Set logging attribute
LU -- Display user terminal LU and EQT
OC -- Set default output cartridge
OFIL -- Specify output file
SE -- Set DATA attributes
SHOW -- Show current DATA attributes
VERS -- Software version and source timestamps

File Management.

CO -- Copy a file
DL -- FMGR directory listings
MV -- Move a file
PU -- Purge a file
RN -- Rename a file
WRIT -- Write data from output buffer to output file

System Commands.

CLX -- Extended cartridge listing
LU -- Display user's terminal LU, EQT, and LU type
RU -- Run program from RTE or FMGR
SY -- Access to RTE system commands
TI -- Display RTE system time
WH -- WHZAT

Graphics.

GR -- Graphics system (see Figs. 2 and 3)

Internal Array Information.

L -- Same as LI (with one exception)
LI -- List specified input and/or output buffer values
LIST -- Same as LI
P -- List pending line
W -- Show pending line location in data buffer

Data Processing.

AVRG -- Averages and statistics on output buffer
BACK -- Polynomial background subtraction from output
DRV3 -- Derivative (3 point Lagrangian interpolation)
FILL -- Fill data buffers
FLIP -- Flip (X,Y) output array to (Y,X)
FLM -- File mathematics
I=0 -- Replace input buffer data with output buffer data
INT -- Numerical integration (trapezoidal rule)
LINT -- Linear interpolation of buffers
MATH -- Math manipulation on data buffers
O-I -- Replace output buffer data with input buffer data
POL -- Polynomial curve fitting (least-squares)
POLY -- Polynomial curve fitting (same as POL)
SM3 -- Smoothing routine, using 3 points
SNGL -- Convert output buffers to a single-valued function
SORT -- Sort (numerical) data buffers
SRCH -- Search for numeric values in data buffers
STEP -- Find/remove a step in output buffer
SUB -- Choose subset for new output buffers
Y -- Estimate y value from x and polynomial fit

Editor.

C -- Change pending line
I -- Insert a line before pending line
K -- Kill pending line
L -- List from pending line
N -- Display number of pending line
P -- Display pending line
R -- Replace pending line
W -- Location of pending line in buffers
n -- Make line n the pending line
-n -- Make pending line backwards n lines from current
{cr} -- Make next line the pending line (forward 1)
{space} -- Insert a line after pending line
\$ -- Make last line the pending line (bottom)
^ -- Make prior line the pending line (backward 1)
- -- Make prior line the pending line (backward 1)
. -- Make next line the pending line (forward 1)
/ -- Make next line the pending line (forward 1)

Help.

?? -- List of legal commands
??.{attribute} -- Description of attributes (see SHOW)
??.{command} -- Help for command

ATTRIBUTES

Several attributes are maintained by DATA for a variety of functions. The current attribute values may be examined with SHOW, and they may be changed with the SE command:

- The ALPH (logical) attribute is used to control alphabetical sorting used by the DL commands.
- The APLT (logical) attribute is used to control automatic plotting of polynomial fits as they are generated by POLY.
- The DX attribute is a real value used to define a differential x value for the interpolation done by LINT.
- The EQ attribute is a real value used to define the tolerance allowed for "equality" tests between real values.
- ERAS is a logical attribute used in conjunction with the APLT attribute by POL to determine whether, during auto-plotting of polynomial fits, the currently-plotted fit should be erased prior to plotting of the new fit.
- ORDR is a logical attribute used by POL to control numerical ordering of the output data following a polynomial fit.
- OVER is a logical attribute used by OFIL to control whether existing files may be overwritten when an output file is specified.
- RANG consists of two integer attributes which define the index range of the output buffers to be used by a number of commands (LINT, POL, INT, SM3, DRV3, AVRG and STEP). The index corresponds to the data record number, or the array subscript.

SUMMARY

Program DATA has been used successfully by our staff to accomplish a variety of data analysis functions in a comprehensive, easy-to-use, unified format. Additional information about the program may be found in Ref. 4.

REFERENCES

1. This work performed at Sandia National Laboratories supported by the U. S. Dept. of Energy under Contract Number DE-AC04-76-DP00789.
2. Reference to a particular product or company implies neither a recommendation nor an endorsement by Sandia National Laboratories or the U. S. Department of Energy, nor a lack of suitable substitutes.
3. "Reflection" software supplied by Walker Richer & Quinn, Inc., Seattle, WA.
4. W. D. Drotning, "DATA: A Comprehensive Data Analysis Environment for HP1000 Computers," SAND87-0369, Sandia National Laboratories, Albuquerque, NM, March, 1987. Available from: National Technical Information Service, U.S. Dept. of Commerce, 5285 Port Royal Road, Springfield, VA 22161.



HP 1000/MEF to HP 9000/500 Interface

Tony Jones
Hewlett-Packard Company
2 Choke Cherry Road
Rockville, MD 20850

Introduction

In the past, data communication between an HP 1000/MEF computer and an HP 9000/500 computer required an A-Series HP 1000 which communicated via Distributed Systems (DS) to the HP 1000/MEF and via Network Services (NS) to the HP 9000. NOW, there is an interface which provides a direct connection between the HP 1000/MEF and the HP 9000.

This interface provides a high-speed data link between an HP 1000/MEF computer and up to two HP 9000/500 (HP-UX) computers over HP-IB. The hardware required consists of standard HP-IB cards on each of the computers. The software consists of two parts: a subroutine library on the HP 1000 and a receiver program on each of the HP 9000s.

The HP 1000 subroutine library provides the ability to open communications to one or both of the HP 9000s, check communication status, send buffers of data, and close communications to one or both of the HP 9000s. The receiver program allows the user to choose an HP-IB interface, store buffers in files or pass them to another program for further processing.

Some of the interface features include:

- Protocol driven for automatic error detection and retries
- ASCII or Binary data transfer
- Data transfer from an HP 1000/MEF to up to two HP 9000/500s for redundancy
- HP 9000 Remote Process Execution from an HP 1000
- HP 1000 to HP 9000 File Copy program
- Support of HP-IB extenders for increased distance

The interface can allow for more distributed on-line processing. For example, one can use the HP 1000 as a real-time front end processor for the HP 9000s or the HP 9000 can offload the HP 1000. The interface can also replace lower speed data transfer via RS-232C or the need to transfer data via magnetic tape.

This software is available for purchase from the Hewlett-Packard Baltimore Washington Area Application Project Center. For further information, contact Ed Magin, Project Manager, at (301) 258-2282.

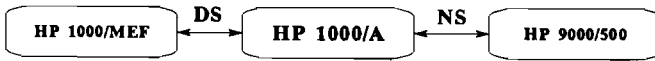
Description

The software interface makes possible high-speed data transfer between an HP 1000/MEF computer and up to two HP 9000/500 (HP-UX) computers over HP-IB. The software consists of three parts: a subroutine library on the HP 1000, a receiver program on each of the HP 9000s, and a set of utility programs which provide file transfer capability.

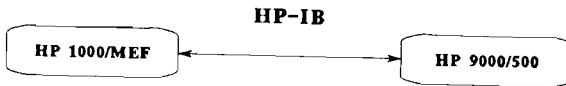
Benefits

There now exists a lower cost solution for data transfer between an HP 1000/MEF and HP 9000/500s. Previously, the recommended solution was to use an A-Series HP 1000 as a front end between the two sets of machines:

Before



Now!



The transfer rate is > 60 Kbytes/sec (with protocol).

What does it replace?

Lower speed communication via RS-232C. It is no longer necessary to use terminal emulators for data transfer or be concerned about binary data transfer. This software supports binary or ASCII transfer. Application programmers can use the HP 1000 subroutine library and put their effort on writing the application.

It also replaces data transfer by magnetic tape. It is no longer necessary to make a tape on one system and move the tape to another system.

Features include:

- HP 1000 subroutine library for integration into user applications

- Protocol driven for automatic error detection and retries
- ASCII or Binary data transfer
- Data transfer between one HP 1000/MEF and up to two HP 9000/500s for redundancy
- HP 9000 Remote Process Execution from an HP 1000
- One-way Process to Process Communication
- Can be used with HP-IB extenders for increased distance
- HP 1000 to HP 9000 File copy utility program
- Can be integrated with DS/1000-IV to provide a more complete networking solution

Figure 1 is a possible system configuration for redundancy applications. There are two sets of redundant HP 1000 systems. The HP 1000s communicate with each other via DS. Each of the HP 1000s have a single connection to both of the HP 9000s. It is the responsibility of the interface software to transfer identical data to both HP 9000s. There can also be multiple HP-IB cards in each of the computers which leads to flexible configurations. An HP 1000 application can communicate via multiple interfaces. An HP 9000 receiver program exists for each HP-IB card. Each of the HP 9000s have four HP-IB cards while each of the HP 1000s have one HP-IB. If one of the HP 9000 receiver programs aborts, then the data will still arrive at the other HP 9000.

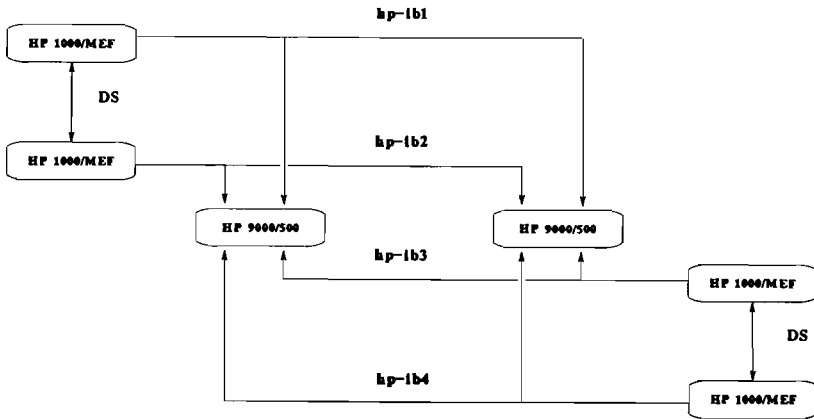


Figure 1: Sample configuration with data redundancy

Figure 2 is a simple configuration for data transfer from an HP 1000 to an HP 9000. In this mode, a user can send data, or do ASCII or binary file transfers. This method of file transfer is much faster than RS-232C connections. For example, transferring a 120K byte binary file to an HP 9000 takes 2 to 5 seconds. The same file requires at least 2 minutes at 9600 baud. If there are two HP-IB cards in the HP 9000, an application can write data to two separate files with one operation.



Figure 2: Single HP 1000 to HP 9000 connection

Start-up

HP 9000

The user can choose to have the "receive" programs started at system boot, or a general user can execute the programs. The former case is the preferred method because the interface could be looked at as a system resource and the HP 1000 can always access the link. Once the program begins proper execution, it will continue to run until the system is shut down or someone kills the process.

System file set-up to run program at system boot

This procedure describes how to set up certain system files to automatically run the "receive" program at system boot time. This method considers protecting the HP-UX Operating System from unauthorized use, especially when a general user on the HP 1000 opens a file for storing data. This method is effective for other applications.

Overall, this technique is based on the system automatically logging in a general user in order to process a ".profile" script file that contains the runstring for the "receive" program. In addition, the last line in the ".profile" script file is a command sequence to log out. The following information describes how to implement this technique:

a) Add a user on the HP-UX Operating System.

b) Create a ".profile" file in that user's home directory. The contents of the ".profile" file must consist of the command sequence for running the "receive" program. In addition, the command sequence must be preceded by the "nohup" command to prevent the "receive" program from terminating when the auto log-out occurs.

Here's a sample copy of a ".profile" file that runs two "receive" programs that redirect any error diagnostic messages to files, and then automatically logs out.

```
nohup receive /dev/hpib1 2>error1&
nohup receive /dev/hpib2 2>error2&
exec login
```

c) Write-protect the ".profile" by using the "chmod" command. For example, suppose ".profile" is contained in a user's home directory named, "/users/hp1000"; issue the following two command sequences ('#' is superuser prompt):

```
# cd /users/hp1000
# chmod 755 .profile
```

d) Modify the "/etc/rc" script file to add a command sequence that can automatically log in the user that you have set-up in step a. Make sure that this command sequence is placed within the block of code that is executed only at system boot time.

For example, suppose you have added a user whose login name is "hp1000"; add the following command sequence in the "/etc/rc" script file:

```
exec login hp1000
```

Running Program/Stopping Program

There are two alternative methods for running the "receive" process. One way is to let the system start the process at system boot time. The other method is to allow any general user to start the process by issuing a certain command sequence.

If you elect to have the "receive" process run at system boot time, then perform the procedure described in the previous section. After completing the installation procedure, the "receive" process runs automatically when the system is powered up, and when the reboot command, "/etc/stopsys -r", is run.

In this case, however, only the superuser can terminate the "receive" process. To do this, you must determine the process-id (PID) used for the "kill" command. For example:

To determine the PID, type:

```
# ps -ef
```

To terminate the "receive" process:

```
# kill <PID>
```

To allow any general user to start the "receive" program, issue the following command sequence ('\$' is user prompt):

```
$ [nohup] receive /dev/hp-ib [>output] [2>error] [&]
```

where:

- **nohup** prevents the "receive" program from terminating when you log out. For more detailed information, read Section 1 of the HP-UX Reference Manual. This argument is optional.
- **/dev/hp-ib** is the device file for the HP-IB interface. This argument is mandatory.
- **>output** - Stores data read into a file named "output". This argument is optional.
- **2>error** - Stores error diagnostic messages into a file named, "error". This argument is optional.
- **&** - Runs the "receive" process in background. This is optional.

OR

```
$ [nohup] receive /dev/hp-ib [|program] [2>error] [&]
```

where:

nohup is same as above.

receive is same as above.

/dev/hp-ib is same as above.

2>error is same as above.

|program - Sends data to another program that is running concurrently with the "receive" program.

& is same as above.

Here are some sample illustrations on how to start the "receive" program:

To run the "receive" program that accepts the filename from the HP 1000, redirects any error diagnostic messages to a file, and runs in the background, type the following command sequence:

```
$ receive /dev/hp-ib 2>error &
```

To run the "receive" program that redirects both the output data and any error diagnostic messages to separate files as a background task, type the following command sequence:

```
$ receive >output 2>error &
```

To run the "receive" program that sends the output data to another program, redirects any error diagnostic messages to a file and then to logout, type the following command sequence:

```
$ nohup receive 2>error | program &
```

If you run the "receive" program without using the "nohup" command, there are two alternative ways to terminate the program. One way is to use the "kill" command described previously. The other technique is to enter the <break> key.

Designating Output File

To store the output data into a file specified by the HP 1000, and to generate any error diagnostic messages to the terminal screen, issue the following command sequence:

```
$ receive /dev/hp-ib
```

However, if the HP 1000 system doesn't pass a filename to the "receive" program, then the output data will be generated to the terminal screen, as well as any error diagnostic messages.

To store the output data into a file specified by the HP 9000 user, and to generate any error diagnostic messages to the terminal screen, use the HP-UX I/O redirection feature. For example, issue the following command sequence:

```
$ receive /dev/hp-ib >output-filename
```

However, if the HP 1000 system passes a filename despite the fact that the HP 9000 user has already specified output redirection, then the filename passed by the HP 1000 system takes precedence.

To send the output data to another program and generate any error diagnostic messages to the terminal screen, use the HP-UX piping feature. For example, issue the following command sequence:

```
$ receive /dev/hp-ib | program
```

However, if the HP 1000 system passes a filename, then it overrides the piping to a program. As a result, that program continues to wait for input and then terminates when the "receive" program terminates.

HP 9000 Remote Process Execution from an HP 1000

An application program can execute a process on the HP 9000 by sending commands to a "shell" program. This method requires that the "receive" program be started with a pipe to a "shell":

```
$ receive /dev/hp-ib | sh
```

In order to execute a command, the application must send a data buffer containing a command terminated with a line feed

character (ASCII 10). For example, the HP 1000 application program could send a package of data and then start a process which would generate a statistical report.

HP 1000

The HP 59310B HP-IB card should be configured as the system controller with HP-IB address 0 (factory settings). The system generation for the HP-IB card must be set up as subchannel 0 for direct I/O. Refer to "The HP-IB in HP 1000 Computer Systems" for further details on configuration and installation of the HP-IB cards.

The HP 37203A or HP 37204A HP-IB extenders may be used to extend the distance between computers.

Checking the system:

The "ShowHpibLus" program will show any logical unit numbers (LU) and equipment table numbers (EQT) which are assigned to HP-IB cards with subchannel zero. One must use an LU assigned to an HP-IB card because the subroutine library uses direct addressing for HP-IB communication. If there are no valid LUs, then one must do a system generation to add them. Refer to "RTE Driver DVA37 For HP 59310B Interface Bus" in order to add LUs and the HP-IB device driver to the system. Use "ShowHpibLus" to verify the available HP-IB LUs.

Setting up the session switch table:

Set up your session switch table (SST) such that a session Lu points to one of the HP-IB card Lu's listed by ShowHpibLus. The application program will pass the session Lu to the subroutine library. One can automate the setup of the HP-IB card Lu by storing the Lu commands in the WELCOM startup file.

Setting the device timeout:

The subroutine library requires setting the device timeout before use. Use the EqT listed by "ShowHpibLus" ('CIxx' is the user prompt and xx is the session number):

```
CIxx> to <EQT> 1000           Timeout of 10 seconds
```

Calling routines

The general procedure for communicating with the HP 9000(s) is:

```
Open communications to the HP 9000(s)
Do while there are buffers to send
    Send buffer to the HP 9000(s)
End do
Close communications to the HP 9000(s)
```

Routine	Description
OpenCommunications	Connect to HP 9000
SendBuffer/SendEMABuffer	Send data
CommunicationStatus	Check last connection
CloseCommunications	Disconnect from HP 9000

Figure 3: HP 1000 interface routines

Status codes are returned to indicate whether the requested operation was successful or not. The subroutines try at least three times to complete a requested operation before reporting an error condition.

This section describes how to call the user level library routines. Variables which are underlined are returned to the user program.

OpenCommunications

This routine opens communication line(s) to the HP 9000(s). The call has the following form:

OpenCommunications (Address, Files, Status)

Type	Name	Element (Range)
Integer*2	Address(4)	[1] Number of HP 9000 addresses (1..2) [2] HP-IB card Lu (1..255) [3] HP 9000 address #1 (1..30) [4] HP 9000 address #2 (1..30)
Character*64	Files(2)	[1] file name #1 [2] file name #2

```

Integer*2      Status(2)      [1] Error for address #1
                                     [2] Error for address #2
                                     0 -> No error

```

SendBuffer

This routine sends a data buffer to the HP 9000(s).
The call has the following form:

```
SendBuffer ( Buffer, Length, Status )
```

Type	Name	Element (Range)
Integer*2	Buffer(N)	[1..3] Reserved for protocol Do not use. [4..N-1] User data area [N] Reserved for protocol
Integer*2	Length	Length of User data + 3
Integer*2	<u>Status(2)</u>	[1] Error for address #1 [2] Error for address #2 0 -> No error

Note:

The first three words of the buffer are reserved for protocol-dependent information. This is to prevent unnecessary copying of the data buffer. One can use an EQUIVALENCE statement to simplify accessing the data buffer:

```

Integer*2      Max                Maximum data buffer size
Parameter      ( Max = X )      X is dependent on space
                                     available to the program in
                                     the system environment.

```

```

Integer*2      DataBuffer(Max) ! Use this for your data
Integer*2      Buffer(Max+4)
Equivalence    ( DataBuffer(1), Buffer(4) )

```

Length varies according to the variable length data buffers.

SendEMABuffer

This routine is identical to the SendBuffer subroutine except the buffer resides in EMA. This would allow sending larger buffers if the use of EMA is desired. The call has the following form:

```
SendEMABuffer ( Buffer, Length, Status )
```

Note:

One can determine the maximum buffer length per transfer by calling the EIOSZ system routine. Refer to "RTE-6/VM Programmer's Reference Manual" for details on EMA programming.

CommunicationStatus

This routine returns whether or not communication is open to an HP 9000. The call has the following form:

```
CommunicationStatus ( Address, Status )
```

Type	Name	Element (Range)
-----	-----	-----
Integer*2	Address	HP 9000 address (1..30)
Integer*2	<u>Status</u>	= 0 -> Communication open <> 0 -> Communication not open

CloseCommunications

This routine closes communication line(s) to the HP 9000(s). The call has the following form:

```
CloseCommunications ( Address, Status )
```

Type	Name	Element (Range)
-----	-----	-----
Integer*2	Address(4)	[1] Number of HP 9000 addresses (1..2) [2] HP-IB card Lu (1..255) [3] HP 9000 address #1 (1..30) [4] HP 9000 address #2 (1..30)
Integer*2	<u>Status(2)</u>	[1] Error for address #1 [2] Error for address #2

ConfigTimeOut

This routine changes the time-out values used in sending and receiving information to and from the HP 9000(s). The call has the following form:

ConfigTimeOut (TransTime, ReceiveTime)

Type	Name	Seconds (Range)
Integer*2	TransTime	Transmit data time (0..327)
Integer*2	ReceiveTime	Receive data time (0..327)

Note:

The recommended times are 10 seconds. Requesting a time-out of 0 means that the HP 1000 will wait indefinitely for the HP 9000(s). This is not a recommended mode of operation.

ReportHPIBErr

This routine displays a descriptive error message based on an error number. The call follows the following form:

ReportHPIBErr (Error)

Type	Name	Status (Range)
integer*2	Error	Error number (0..12)

Note:

ReportHPIBErr will accept the status number returned by the subroutine library. This can be helpful during the development phase.



Error Recovery

HP 9000

Table of Error Conditions

The errors generated by the "receive" program fit into three categories. The first type is based on generating an error diagnostic message to the terminal screen followed by program termination. The second type is based on generating an error diagnostic message to the terminal screen with the program continuing to execute. The third type is based on sending a response PDU packet to the HP 1000 program with the HP 9000 program continuing to execute.

The following information is a table that shows all errors that will cause the HP 9000 program to terminate:

Error Message	Returned From
error...incorrect usage of this program correct usage is: receive {devicename}	MAIN
opening the HPiB interface failed: errno=n	MAIN
HPiB self-test FAILED: errno=n	MAIN
failed to enable EOI mode: errno=n	MAIN

The following information is a table that shows all errors generated to the terminal screen, by default. However, these errors will not cause the program to terminate:

Error Message	Returned From
error occurred while waiting to listen	MAIN
read error: errno = n	MAIN
Out-of Sync Error: Re-syncing	MAIN
{filename } is busy: errno=n	MAIN
error occurred while waiting to talk	SEND
error on sending response: errno=n	SEND

The following information shows the errors passed to the HP 1000 program:

Error Number	Returned From	Meaning
1	MAIN	Protocol Violation
2	MAIN	File Access Problem
3	MAIN	Bad Sequence Number

Note:

Refer to the manual entry for the command, "errno", contained in Section 2 of the HP-UX Reference Manual to obtain more detailed information on those errors that generate a message of the form:

errno=n

where: n is some integer value.

HP 1000

Table of Error Conditions

The subroutine library will return various status numbers based on the result of a user request. One can use the ReportHPIBErr routine to display the message corresponding to the status. The following table contains the error numbers, which routines can return them, and their meanings:

Legend of abbreviations for library subroutines:

- O -> OpenCommunications
- B -> SendBuffer/SendEMABuffer
- S -> CommunicationStatus
- C -> CloseCommunications
- All -> Any of the above

Error Number	Returned From	Meaning
0	All	Ok
1	OBC	Bad checksum after N attempts
2	OBC	Time-out on sending information
3	OBC	Time-out on receiving response
4	OBC	Unrecoverable error on HP 9000
5	BS	Communication line not open
6	B	Bad sequence number detected at HP 9000
7	O	Communication line already open
8	O	HP-IB card already in use or invalid Lu
9	OBC	Unrecoverable error on HP 1000
10	OBC	HP 9000 caused a protocol violation
11	OBC	Bad parameter passed by user
12	O	No more room for information (addresses)

The following messages may be displayed if errors are detected during HP 1000 - HP 9000 communication:

Message	Cause
HPIBACCESS: Xlux error detected	Improper system set up
HPIBACCESSE: VMAIO error detected	SendEMABuffer error

Note:

"Unrecoverable error..." means that the application program must start the general calling sequence over again.

Special Considerations

General

Other devices on HP-IB interface:

The only devices that should be on the HP-IB interface are the HP 1000 and the HP 9000(s).

HP-IB extenders:

The HP-IB extenders cause a slower data transfer than a direct connection. The data on the HP-IB interface always moves at the speed of the slowest device.

HP 9000

Multiple I/O Redirections to Same File:

If multiple copies of the "receive" program redirect the output (">") to the same file, then the data from each program will overlap in the same file. At the operating system level, the capability of locking files is not implemented. However, to bypass this limitation, the output filename must be passed from the HP 1000 since file locking capability has been implemented programmatically.

HP 1000

Timeouts:

The time-out settings control how long the HP 1000 will wait for the HP 9000(s) and the amount of time there will be between retries. One must also consider HP 9000 system loading. The time-out must be long enough such that the HP 9000(s) has time to complete the task. Never set the time-outs to zero because the HP 1000 program will hang if there is ever a problem on the HP 9000(s).

The user must set the time-out for the HP-IB card eqt before running the program if the send-time-out and receive-time-out are the same values since the subroutine library only sets the time-out if the values are different.

OpenCommunications:

Once the application program has opened communications to the HP 9000s, another request to open communications does not lose the connection. This feature is useful if an HP 1000 error was detected when trying to connect to the HP 9000s. The HP 1000 error would most likely be caused by the user aborting an application program and restarting it without restarting the receiver programs.

Buffer sizes:

It requires less protocol overhead to transfer one large buffer than it takes to transfer many small buffers.

EMA vs. Non-EMA buffers:

It takes longer to access EMA buffers vs. non-EMA buffers due to the overhead of mapping routines.

User buffer lengths larger than maximum transfer size:

The user must consider that the subroutine library requires a buffer with the first three words and last word reserved for protocol usage. If the length of the user data is larger than the maximum transfer length, then it will be necessary to make multiple calls to SendBuffer/SendEMABuffer. If the user chooses to send the buffer by passing multiple sub-buffers starting at different indices, then plan to preserve the first three locations before the indexed sub-buffer location and one location past the end of the sub-buffer.

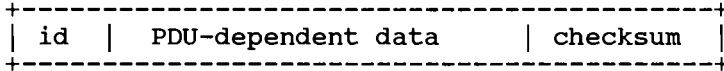
CloseCommunications and HP 1000 error:

CloseCommunications will try to close any requested (valid) HP-IB address whether or not the connection has been established. This allows one to close a connection to the HP 9000(s) if a previous program left it open.

Technical Description

Protocol Data Units (PDU)

The general method for communicating between the HP 1000 and the HP 9000(s) is via Protocol Data Units (PDU). A PDU consists of 16 bit unsigned integers. This implies that this is a word- (vs. byte-) oriented protocol. The PDUs have the following format:

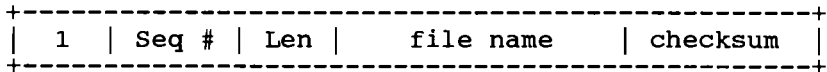


Each PDU has an identifier (id), information related to the specific PDU and a checksum. The checksum is an unsigned 16-bit sum of all 16-bit words in the PDU. The checksum calculation ignores any carries.

The HP 9000(s) acknowledges each PDU from the HP 1000 with either an ACK or a NACK PDU. The following section describes the various PDUs:

REQ

OpenCommunications uses the request (REQ) PDU to pass a file name. If the file name is blank, the data will be written to the file or pipe specified in the start-up command on the HP 9000.



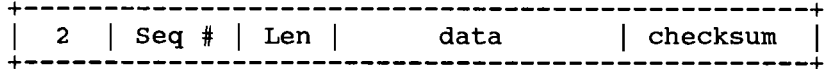
The id for a REQ PDU is a 1.

The sequence number (seq #) is used to identify a data PDU which has been sent more than once. The HP 9000(s) will accept a data packet and discard any successive data packets which have the same sequence number.

The length of the filename, in words, is stored in the Len field. A file name with an odd number of characters must be padded with a blank.

DATA

SendBuffer/SendEMABuffer uses the DATA PDU to pass data buffers.



The id for a DATA PDU is 2.

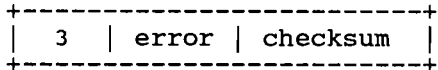
The Seq # is modulo 8 (0..7).

The Len is the length of the user data in words.

The data portion contains the user data portion of the buffer passed by SendBuffer/SendEMABuffer.

ACK

The ACK PDU is used to acknowledge the checksum of the received packet.

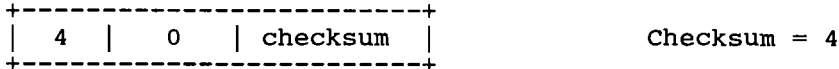


The id for the ACK PDU is 3.

The error field (one word) contains any errors which may have occurred. The HP 9000 uses this field to return the status of an operation or the detection of a perceived protocol violation.

NACK

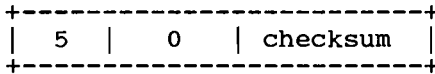
The NACK PDU is used if the PDU checksum does not match the calculated checksum.



The id for the NACK PDU is 4.

EOT

CloseCommunications uses the EOT PDU to close the file on the HP 9000.



Checksum = 5

The id for the EOT PDU is 5.

Using the Send9000 example program:

Note: The user must set the time-out for the HP-IB card eqt before running the program if the send time-out and receive time-out are the same values. The subroutine library only sets the time-out if the values are different.

Program with parameter usage:

```
CIxx> Send9000, PackCount, AskFile, STimeOut, RTimeOut, HP9KAddr
```

Parameter descriptions:

PackCount = number of buffers to send
Default: 1 buffer

AskFile <> 0 --> Ask for file name for first HP 9000
Default: testfile First HP 9000
Default: testfile2 Second HP 9000

STimeOut = Time-out for sending a PDU (seconds)
Default: 10 seconds

RTimeOut = Time-out for receiving a PDU (seconds)
Default: 10 seconds

HP9KAddr <> 0 --> Ask for: HP-IB card lu
Number of HP 9000s (2 max)
HP 9000 address 1
HP 9000 address 2
Buffer length (10000 max)

Default: 45 HP-IB card lu
1 HP 9000
28 HP 9000 address 1
26 HP 9000 address 2
10000 word buffers

There is very little error checking for user entries.

Examples:

To run the program in default mode:

Send9000

To send data to two HP 9000s:

Send9000,1,1,10,15,1

Description:

Send one buffer (automatically generated)
Ask for file name for the first HP 9000
Wait up to 10 seconds for sending a PDU
Wait up to 15 seconds for receiving a PDU
Ask for HP-IB card lu number
Ask for number of HP 9000s
For each HP 9000 (2 max.)
 Ask for HP 9000 HP-IB address
Ask for buffer size in words (10000 max.)

Refer to Appendix A for the "Send9000" program source listing.

Conclusion

The HP 1000/MEF to HP 9000/500 interface provides a lower cost solution to reliable, high-speed data transfer. Application programmers can continue to concentrate on the application and spend less time being concerned about interfacing the computers. Applications can be more distributed for on-line processing needs.

Appendix A: Sample Data/File Transfer Program

This program is made up of three parts: initialization and data transfer, ASCII file transfer, and binary file transfer.

```
program send9000
*
* (c) Hewlett-Packard Company, 1986
*****
* F Series version <870413.0948>
* This version uses direct I/O. The lu is 45 and the addresses are 28 & 26.*
* If compiled with debug (d), one will get a dot (.) for each data buffer. *
*
* Program usage: SEND9000, PackCount, AskFile, SndTimeout, RecTimeout, HP9KAddr*
*
* PackCount = number of packets
* AskFile <> 0 --> Ask for file name
* SndTimeout & RecTimeout are in seconds
* HP9KAddr <> 0 --> Ask for HP-IB lu, HP 9000 adrrs & Buff size
*
*****
** Added character buffer for commands from console
** Added file transfer. Note: file is transferred as type l (Filter it).
** Added ascii file transfer. Note: each record is terminated with a \n.
c
c This program makes use of the HPIB communication routines:
c
c OpenCommunications
c SendBuffer
c CommunicationStatus
c CloseCommunications
c ConfigTimeout
c ReportHPIBErr
c
c implicit none
c Include HP9000def.inc
c
c integer*2 address(MaxHP9000+2) ! HP 9000 addr info
c +-----+
c | number of hpiib devices |
c +-----+
c | Bus lu of HPIB card |
c +-----+
c | HPIB address for HP9000 #1 |
c +-----+
c | HPIB address for HP9000 #2 |
c +-----+
c
c character*64 files(MaxHP9000) ! file names for each HP9000
c integer*2 status(MaxHP9000) ! result codes for each HP9000
c integer*2 dlen ! length of the data buffer
*****
* Change dlen to appropriate data length *
*****
parameter ( dlen = 10000) ! Length of data
integer*2 buffer(dlen+4) ! Full buffer including protocol
```

```

integer*2   databuff(dlen)  ! True data portion of buffer

** Added charbuff and TrimLen
character*80 charbuff       ! character buffer
integer*2   TrimLen         ! TrimLen function

** Added HP 1000 filename for file transfer
character*64 HP1000File     ! HP 1000 source file
integer*2   MaxLength       ! Buffer length for read
parameter   ( MaxLength = 128*(dlen/128) ) ! 128 word blocks
logical*2   AsciiMode       ! File transfer method

integer*2   len              ! length of data transferred
integer*2   BusLu            ! HPIB card logical unit
integer*2   HP9000Count     ! Number of HP9000s
integer*2   HP9000addr(MaxHP9000) ! HPIB addresses for each HP9000
integer*2   i                ! scratch
integer*2   n                ! scratch
Integer*2   PacketCount     ! Number of buffers to send
Integer*2   Parms(5)        ! RMPAR parameters
Integer*2   TransTime       ! Send timeout in seconds
Integer*2   ReceiveTime     ! Receive timeout in seconds

Integer*2   TimeIndex       ! index for elapsed time
Integer*4   ElapsedTime     ! Time function
Integer*4   Timer(100)     ! Elapsed time at major events
Integer*4   T                ! temp value for differences
Logical     RetryNeeded     ! Flag used in retrying OpenComm...

*****
*   Line up the data so that it is placed after the protocol information *
*****
*   Equivalence ( Buffer(4), databuff )
** Added charbuff
   Equivalence ( Buffer(4), databuff, charbuff )

*****
*   Change the BusLu to the appropriate HPIB card LU.
*****
   Data      BusLu/45/      ! Lu for the HPIB card. Modify...
   Data      PacketCount/1/

c   Get the number of packets from the user
   Call RMPAR ( Parms )    ! Get packet count
   if ( Parms(1) .gt. 0 ) PacketCount = parms(1)

c
c   Open communication to HP9000s
c   file names
c

   files(1) = 'testfile'
   if ( Parms(2) .gt. 0 ) then
c     Get file name
       write ( 1,* ) 'Enter filename [<cr>=none] _'

```



```

        read ( 1,'(a)' ) files(1)
    endif

    files(2) = 'testfile2'
c
c    Set up the timeouts
    if ( parms(3) .gt. 0 ) TransTime = parms(3)
    if ( parms(4) .gt. 0 ) ReceiveTime = parms(4)
    if ( TransTime .ne. 0 .AND. ReceiveTime .ne. 0 ) then
        Call ConfigTimeout ( TransTime, ReceiveTime )
    endif
c
c    Set the HP-IB addresses
    hp9000addr(1) = 28
    hp9000addr(2) = 26
*
    len = dlen

    HP9000Count = 1
c    Get the number of HP9000s and addresses if applicable
    if ( parms(5) .gt. 0 ) then
        write (1,*) 'HP-IB card Lu = _'
        read (1,*) BusLu
        write (1,*) 'How many HP9000s? _'
        read (1,*) i
        HP9000Count = i
        do i = 1, HP9000Count
            write (1,*) 'HP9000Addr('i,') = _'
            read (1,*) n
            HP9000Addr(i) = n
        enddo
        write (1,*) 'Buffer size (words) = _'
        read (1,*) n
        if ( n .gt. 0 ) len = n
    endif

    address(1) = HP9000Count
    address(2) = buslu
    address(3) = hp9000addr(1)
    address(4) = hp9000addr(2)
*
    Start up the timer
    call ResetTimer
    timeindex = 1

    Timer(timeindex) = ElapsedTime ()
    timeindex = timeindex + 1
c
    Try multiple opens to put HP9000s in starting state ( state A )
    i = 0
    RetryNeeded = .true.
    do while ( i .lt. 2 .AND. RetryNeeded )
        RetryNeeded = .false.

        call OpenCommunications ( address, files, status )

        do n = 1, HP9000Count

```

```

        if ( status(n) .eq. HP1000ERR ) RetryNeeded = .true.
    enddo
    i = i + 1
enddo

Timer(timeindex) = ElapsedTime ()
timeindex = timeindex + 1

write (1,*) 'Statuses are: ', (status(i), i=1,HP9000Count)
do i = 1, HP9000Count
    if ( status(i) .ne. 0 ) then
        write (1,*) 'Error on OpenCommunications to ', i , ' _'
        write (1,*) status(i)
        call ReportHPIBErr ( status(i) )
    endif
enddo

c
c Send some data
c
do i = 1, len
    databuff(i) = i                ! Portion of buffer
enddo

c Initialize status in case of no addresses open
do i = 1, HP9000Count
    status(i) = NOTOPEN
enddo

c
c The buffer length must be for the full buffer.
c Offset 3 words to account for protocol information.
c
Timer(timeindex) = ElapsedTime ()
timeindex = timeindex + 1

c
c Send some data buffers
c
do n = 1, PacketCount

    call SendBuffer ( buffer, len + 3, status )

    do i = 1, HP9000Count
        if ( status(i) .ne. 0 ) then
            write (1,*) 'Error on SendBuffer to ', i , ' _'
            write (1,*) status(i)
            call ReportHPIBErr ( status(i) )
        else
            write (1,*) '._'
        endif
    enddo

d
d     do i = 1, len
d     databuff(i) = databuff(i) + 1
d     enddo
d     enddo
d     write (1,*) ' '

```

```

Timer(timeindex) = ElapsedTime ()
timeindex = timeindex + 1
c
c Check communication
c
do i = 1, HP9000Count
    call CommunicationStatus ( address(i+2), status(i) ) ! offset to add
    write (1,*) 'Communication status to ', i, ' is_'
    if ( status(i) .eq. 0 ) then
        write (1,*) ' open'
    else
        write (1,*) ' closed'
        Call ReportHPIBErr ( status(i) )
    endif
enddo

Timer(timeindex) = ElapsedTime ()
timeindex = timeindex + 1

c
c Shut down communication ( and close the HP9000 files )
c
call CloseCommunications ( address, status )

do i = 1, HP9000Count
    if ( status(i) .ne. 0 ) then
        write (1,*) 'Error on CloseCommunications to ', i, ' _'
        write (1,*) status(i)
        Call ReportHPIBErr ( status(i) )
    endif
enddo

c
c Show the times
c
Timer(timeindex) = ElapsedTime ()
write (1,*) 'Milliseconds'
t = 0
do i = 1, timeindex
    write (1,500) i, Timer(i), Timer(i)-t
    format ( 'Timer(',i3,',) =',i8,', Diff =',i8 )
    t = timer(i)
enddo

c
Timer(4) - Timer(3) is time to send buffers
t = Timer(4) - Timer(3) ! milliseconds
write (1,*) 1000.0*PacketCount*len*2/t, ' Bytes/second'

write (1,*) 'Done with HPIB communications test'

** Added ability to send commands to the HP 9000
charbuff(1:1) = 'N'
write (1,*) 'Do you want to send commands? [n] _'
read (1,'(a)') charbuff

```

```

call Casefold ( charbuff )

550 Continue
if ( charbuff(1:1) .ne. 'N' ) then
c   Get commands from the user until done (/e)
c   Use the first address for sending commands
   address(1) = 1
   files(1) = ' '

   call OpenCommunications ( address, files, status )
   Call ReportHPIBErr ( status(1) )
   if ( status(1) .eq. 0 ) then
     write (1,*) 'Enter /e to exit.'
     write (1,*) 'Command: _'
     read (1,'(a)') charbuff
     do while ( charbuff(1:2) .ne. '/e' )
c       Force an even number of characters
       len = TrimLen ( charbuff )
c       Append a <LF> to line (keeping an even # chars)
       len = TrimLen ( charbuff )
       if ( mod (len, 2) .eq. 0 ) then
c         even number of chars. Append a space
         len = len + 1
         charbuff(len:len) = ' '
       endif
       len = len + 1

       charbuff(len:len) = char(10) ! <LF>
c     At this point, len is always even. Change to words
       len = len/2

       Call SendBuffer ( buffer, len + 3, status )
       Call ReportHPIBErr ( status(1) )

       write (1,*) 'Command: _'
       read (1,'(a)') charbuff
     enddo
     Call CloseCommunications ( address, status )
     Call ReportHPIBErr ( status(1) )
   endif
endif

** Added ability to send files to the HP 9000
charbuff(1:1) = 'N'
write (1,*) 'Do you want to send a file? [n] _'
read (1,'(a)') charbuff
call Casefold ( charbuff )
if ( charbuff(1:1) .eq. 'Y' ) then
c   Get commands from the user until done (/e)
c   Use the first address for sending commands
   address(1) = 1
   files(1) = ' '

   charbuff = ' '
   write (1,*) 'Enter /e to exit, /c to return to commands.'
   write (1,*) 'HP 1000 file: _'

```

```

read (1,'(a)') charbuff
if ( charbuff(1:2) .eq. '/c' ) goto 550
do while ( charbuff(1:2) .ne. '/e' )
  HP1000File = ' '
  HP1000File = charbuff(1:64)
  Call CaseFold ( HP1000File )

  charbuff(1:1) = 'A'
  write (1,*) 'Ascii or Binary Data [a]: _'
  read (1,'(a)') charbuff
  Call CaseFold ( charbuff )
  if ( charbuff(1:1) .eq. ' ' ) charbuff(1:1) = 'A'
  if ( charbuff(1:1) .eq. 'A') then
    asciimode = .true.
  else ! Binary data transfer
    asciimode = .false.
  endif

  write (1,*) 'HP 9000 file: _'
  read (1,'(a)') files(1)

  if ( asciimode ) then
    Call SendAsciiFile(address, HP1000File, files, buffer,
+                               MaxLength, status )
  else
    Call SendFile ( address, HP1000File, files, buffer,
+                               MaxLength, status )
  endif

  charbuff = ' '
  write (1,*) 'HP 1000 file: _'
  read (1,'(a)') charbuff
  if ( charbuff(1:2) .eq. '/c' ) goto 550
600   enddo
      endif

      END

```

```

Subroutine SendAsciiFile ( address, Srcfile, DstFiles, dcb,
+                               MaxLength, status )
*   (c) Hewlett-Packard Company, 1986
c   Description <870413.0948>
c   This routine copies a file to the HP 9000. SendAsciiFile
c   copies the file over and appends a new line character at the
c   end of each line.

c   Declarations
implicit none

integer*2      address(*)           ! HP 9000 addresses
character*(*)  Srcfile(*)           ! HP 1000 file name
character*(*)  Dstfiles(*)          ! HP 9000 file names
integer*2      dcb(*)               ! Large dcb for read
integer*2      MaxLength            ! Read length
integer*2      status(*)            ! Return status

```

```

c   Assume lines will be no longer than 1024 bytes so that we can
c   still stuff a space and a new line character (\n) on the end.
c   The protocol requires 3 words in front of the buffer and 1 word
c   behind the buffer.

```

```

integer*2      buffer(517)          ! PDU buffer: 512 + 4 + 1 wds
integer*2      wbuffer(512)         ! word buffer
character*1024 cbuffer              ! character buffer
Equivalence    ( buffer(4), wbuffer, cbuffer )

```

```

integer*2      err                  ! Fmp Err
integer*2      err1                 ! Fmp Err
integer*2      len                  ! Read length
integer*2      FmpRead              ! Function
integer*2      FmpRewind            ! Function
integer*2      dlen                 ! dcb length
parameter      ( dlen = 10000 )     ! dlen = 16 + 128n
integer*2      maxbytes             ( maxbytes = 1024 )
parameter      ( maxbytes = 1024 )
code

```

```

c   Try to open the HP 1000 file.
c   Call FmpOpen ( dcb, err, SrcFile, 'ros', dlen/128 )
c   if ( err .ge. 0 ) then
c       ! File exists

```

```

    err = FmpRewind ( dcb, err1)
    if ( err .lt. 0 ) stop
    Try to connect to HP 9000
    Call OpenCommunications ( address, Dstfiles, status )
    if ( status .eq. 0 ) then
c       Copy the file
        err = 0
        Do while ( err .ge. 0 )
            len = FmpRead ( dcb, err, wbuffer, MAXBYTES )
            if ( err .ge. 0 .AND. len .ne. -1 ) then

```

```

c       Append the <LF>
c       If ( mod ( len, 2 ) .eq. 0 ) then

```

```

            len = len + 1
            cbuffer(len:len) = ' '

```

```

        Endif
        len = len + 1
        cbuffer(len:len) = char(10) ! <LF or \n>

```

```

c       Change len to words
c       len = len/2

```

```

        Call SendBuffer ( buffer, len+3, status )
        if ( status .ne. 0 ) then
            Call ReportHPIBErr ( status )
            err = -1          ! Force exit
        endif

```

```

        elseif ( err .lt. 0 .AND. err .ne. -12 ) then
            Call FmpReportError ( err, SrcFile )
        endif

```

```

    Enddo

```

```

c          Clean up
          Call CloseCommunications ( address, status )
          Call ReportHPIBErr ( status )
          Call FmpClose ( dcb, err )

          else ! Error on OpenCommunications
            Call ReportHPIBErr ( status )
          endif
        else ! Error on FmpOpen
          Call FmpReportError ( err, SrcFile )
        endif

    return
end

```

```

Subroutine SendFile ( address, Srcfile, DstFiles, buffer,
+                   MaxLength, status )
*   (c) Hewlett-Packard Company, 1986
c   Description <870413.0948>
c   This routine copies a file to the HP 9000. SendFile
c   copies the file over as type 1. This implies that
c   the user must put an ASCII file through a filter
c   to convert it to an appropriate file on the target
c   system.

c   Declarations
    implicit none

    integer*2      address(*)          ! HP 9000 addresses
    character*(*)  Srcfile(*)          ! HP 1000 file name
    character*(*)  Dstfiles(*)         ! HP 9000 file names
    integer*2      buffer(*)           ! PDU buffer
    integer*2      MaxLength           ! Read length
    integer*2      status(*)           ! Return status

    integer*2      dcb(16)             ! File dcb
    integer*2      err                 ! Fmp Err
    integer*2      len                 ! Read length
    integer*2      FmpRead             ! Function

c   code

c   Try to open the HP 1000 file as type 1 for unbuffered read.
    Call FmpOpen ( dcb, err, SrcFile, 'rosxf', 1 )
    if ( err .ge. 0 ) then ! File exists
c       Call FmpSetPosition ( dcb, err, 0, -1J )
c       Try to connect to HP 9000
        Call OpenCommunications ( address, Dstfiles, status )
c       if ( status .eq. 0 ) then
c           Copy the file
            err = 0
            Do while ( err .ge. 0 )
                len = FmpRead ( dcb, err, buffer(4), MaxLength )
                if ( err .ge. 0 .AND. len .ne. -1 ) then

```

```

        Call SendBuffer ( buffer, len+3, status )
        if ( status .ne. 0 ) then
            Call ReportHPIBErr ( status )
            err = -1          ! Force exit
        endif
    elseif ( err .lt. 0 .AND. err .ne. -12 ) then
        Call FmpReportError ( err, SrcFile )
    endif
Enddo

c      Clean up
      Call CloseCommunications ( address, status )
      Call ReportHPIBErr ( status )
      Call FmpClose ( dcb, err )

      else ! Error on OpenCommunications
        Call ReportHPIBErr ( status )
      endif
else ! Error on FmpOpen
  Call FmpReportError ( err, SrcFile )
endif

return
end

```


Developing A Complex Engineering Test Database

Ann D. McCormick
INTELSAT

International Telecommunications Satellite Corporation
2250 E. Imperial Highway, Suite 750
El Segundo, CA 90245

I. Introduction

The use of database methodology has been well established in business computing for many years. During the pioneering days of computer development the business community quickly saw that databases could be used to reduce a large but well-defined set of facts to a table of numbers or, in recent years, a chart in an astonishingly small amount of time. The database structure imposed an order on information already rigidly defined. For example, a credit card company would have a database of cardholders containing name, address, place of business, credit references, account number, current balance, past due balance and so on. This database would not be significantly different from a bank's database of outstanding loans or a manufacturer's database of clients. Even allowing for differences from country to country, such things as name, account number, and outstanding balance are sufficiently universal in function that detailed software packages can be written and marketed to accommodate a large audience.

The engineering computing community has not been as fortunate as our brothers in the business world. There is very little uniformity between different engineering test databases, largely because the nature of the testing varies. An engineering database, for the purpose of this paper, may be defined as a collection of measurements made during the testing of an object. The object may be as simple as a can opener or as complex as a manned space vehicle. The collection of measurements may be fairly easy to define (how many ways can you test a can opener?) or be open-ended when the item being tested is a complex one-of-a-kind item.

I will be describing here the principles INTELSAT learned from designing, developing, and maintaining a complex test database for the INTELSAT6 telecommunications satellite. For this project we were dealing with test data from six spacecraft, each with twelve subsystems containing over 170 electronic units. At this writing, over 110,000 test data files have been generated, using a total of 2490Mb disc storage. To process and analyze this data more than 400 applications programs were written over four years. There were times that it seemed we were trying to number the sands on the beach. But with much perseverance we managed to develop a successful database that is regularly used by the engineering staff.

II. Defining the Test Database

To define the scope of a test database you must first have an understanding of what objects will be tested and how they will be tested. Some of the questions you must ask are:

1. How many types of units and how many of each type will be tested?
2. Are the units related or unrelated?
3. What is the likelihood of design changes that will effect the testing process?
4. How many measurements will be taken during a typical unit test?
5. Will all the test measurements be useful historically?

The quantity and type of units being tested gives the first factor in the database sizing. Once you have a contract or a blueprint with the unit descriptions you can physically count how many objects you will be testing. However, this is not always as easy as it may seem. Will all the items that look like "units" to the programmer have their data stored? What about data taken at other test sites? How many spare units will be tested? These questions must be answered early in the design phase with the help of Engineering. Then take the answers to these questions, assume "worst-case" conditions and then add five percent.

The question of relation between units gives an indication of whether you logically have two or more actual databases to design. If you are dealing with test data from two different automobiles you can reasonably expect to maintain two distinct databases. However, keep in mind any similar components between the units. The two cars may have very different suspension systems but virtually identical brakes. In any statistical analysis of the brakes, you can be assured that at some point Engineering is going to want to look at data from both cars. Thus the two "distinct" databases logically intersect in the braking system. It would be advisable to keep the internal format of the braking system data in the two databases compatible.

Design changes are impossible to predict, except to say that they will certainly happen. Avoid designing your database or its software around a peculiarity of the data. For example, at one point in the INTELSAT6 satellite project there was serious discussion about adding one or more antennas to the spacecraft. This would have had potentially disastrous to some of our software and data files because we had designed software around a specific number of antennas. Be constantly on the lookout for values used in the software that should be coded as a parameter rather than a constant.

Sizing the number of data points you expect to receive can be straightforward if you have previous experience with similar test data. First get an accurate count of the amount of test data from the similar unit. Be sure to allow for retests in your counting. Then take this figure and multiply by a complexity factor appropriate to the design differences between the two units. If Unit A has 15% more test points than Unit B, then Unit A will probably generate 15% more data than Unit B. If, on the other hand, you have no previous test database to use for comparison you are forced to make a manual count of number of measurements per test per unit. Since this is a less reliable method I recommend rounding up all your figures to allow for errors.

The final question is probably the most difficult one to answer. Do you really need all the data available? Not all test data needs to be stored on a computer. Most data that measures a "yes/no" condition does not merit being included in a test database. There is not much you can do, for example, with the answer to the question "Did the unit temperature exceed 100° F?" If no one cares what the precise temperature was, provided it did not exceed some limit, then the test result has no use beyond the immediate need of deciding whether the test passed or failed. Some data is also too complex to store digitally and make any meaningful computations. It may be necessary to examine the waveform output of a unit during test but if you do not expect to compare one unit's waveform to another or do any statistical analysis on the waveforms then the digital storage of this information is not really useful.

III. Selling "New Technology" to a Skeptical World

During the design phase of your database you will begin to identify pockets of skepticism and hesitancy among your data collectors, target users, and even your management. Whenever there is a potential change to "the way we've always done it" on the horizon you are guaranteed to find a large spectrum of responses. Here are a few of the gems I have encountered:

"I'll never use the computer."

"I just want to push a button and get a plot. If it is more complicated than that, I just won't remember it."

"I'm not sure the development cost is worth the effort. No one will use the system when it's done, anyway."

"But I like writing the numbers in my log book. It forces me to look at the data."

In the normal population of engineering test database users there is a high percentage of these users who have had some programming experience, usually with BASIC and usually on an overworked university

computer that would swallow their batch jobs and sometime later (perhaps hours later) spit out a sheet or two of cryptic messages in some obscure language for which the Rosetta Stone has not yet been discovered. This basis for their experience with computers usually puts them in one of two categories: Those That Do and Those That Don't.

Those That Do

- * Do find that the computer can be a positive challenge.
- * Do want to learn something about how it works.
- * Do want it to help them with their job.

Those That Don't

- * Don't really want to talk to you about how they can use your database.
- * Don't want a CRT installed in their area because that might mean they will have to use it.
- * Don't think you can possibly automate what they have been doing quite well, thank you, with paper and pencil.

You, the systems analyst/designer, must deal with these varied reactions. Every system needs its champion and you're elected to fill that position. This part of your job may not show up on any task schedule but it is critical to the success or failure of any test database, regardless of size or complexity. The finest database system in the world is worthless if it is not being used. The following are some guidelines to help ensure your system will be used:

1. Learn something about the job of the user or the data collector. Find ways to make your database work for him instead of against him.
2. Refuse to take the role of The Enemy, even if they do their best to cast you in that role. When you are faced with what seems like unreasonable resistance to automation, consider the possibility that you are dealing with "techno-phobia". Try to work privately with that individual or group to overcome this.

3. Keep communications open with everyone. Publish software or database changes well in advance of their implementation. Make sure that everyone knows you're willing to listen to suggestions or constructive criticisms.
4. Avoid falling into a "fortress mentality". A fortress wall not only keeps out the petty annoyances but prevents you from seeing the danger signals that could help you to avert disaster.

IV. Collecting Valid Data

Now that you know what test data is available, you must develop your data collection network. First you must identify all your data point-of-entry locations. Next you must determine the acceptable methods of data entry. Finally you must define your means of data validation and entry into the database.

The point-of-entry into your database should be as close to the testing as possible. The ideal situation is the digital storage of data during automated testing. Once the bugs are removed from the test software, you should get reliable, accurate data. The next best alternative is to have the test conductor enter the data into your collection via a series of menus or prompts immediately after the test. This method has the advantage that the test conductor can filter out the more glaring errors in the data. The least desirable point of data entry is to have the entry done by a data clerk who is not familiar with the test. Unfamiliarity with the data compounds the problem of getting numbers from a piece of paper, through a fallible interpreter, and into the computer.

The method of data entry can be as varied as the type of computer equipment you have to work with. A direct reading of the test equipment is always the best method. When this is not available, a variety of bar code readers, magnetic strip readers, and CRTs are available. Whatever input device you purchase, make sure that it can be made as user-friendly as possible. There is nothing more frustrating than attempting to store data and getting the cryptic and very unfriendly message "Data Storage Failed!" Also be cautious about purchasing devices that have just been put on the market. You could find yourself debugging their hardware design while struggling to get your data collection working. If the data entry process is too frustrating your data collection people can become uncooperative to the point of abandoning further attempt to use the system.

The validation of data should be done as the data is being entered into the database. This allows for feedback to the entry point and possible immediate correction. In writing validation software the cardinal rule is:

NEVER, NEVER ASSUME THAT DATA IS CORRECT

I believe that this principle cannot be overstressed. Numbers can be transposed or in the wrong position, test equipment can be turned off or not working, magnetic strips can be deGaussed, and time clocks can be set incorrectly. If there is any area where Murphy's Law can sabotage your database, this is it. The contents of your database must have a high rate of reliability or your end-users will not have the confidence in the outputs they receive. We have found that the amount of questionable data must be somewhere below 2% to ensure a reasonable confidence in the database. When confidence is low, usage (your ultimate measure of success) is also low.

Validation is accomplished by comparing each value with its known expected range. If the serial number is a four digit ASCII value, don't accept "003 " or " 2". Either make the data correction yourself (being careful not to introduce further error) or demand the data be reentered. If the value, such as a part number, has a discrete set of possible values, keep an up-to-date table in your software of these values and refuse anything that doesn't match. If you know that your data retrieval software will expect a value to be left justified or all capitals, make those corrections now. Even the test values can be filtered for valid information. If you know, for example, that a voltage can never exceed 120V without destroying the unit under test then it is safe to reject a value of 290.5V as an error. Be very conservative, however, about altering the test measurements. If significant portions of the data are being "corrected" your users' confidence in the system will drop.

In some cases, validation may not be possible. If the nature of the test is such that the outcome is unpredictable, as is possible with prototype testing, you will not be able to filter out errors programatically. In this case it may be necessary to have human intervention in the validation process to avoid interspersing good data with bad data. But be sure that this post-test validation is done very soon (within 24 hours at most) after the test is complete. If the validation takes place later than that you increase the risk of validation errors.

V. Helping the User to Define His Output Requirements

In helping your database users to define with you what the output requirements are for the test data system, you must recognize that you are filling the role of ambassador between the two worlds of the user and the programmer. To do this you must have a working grasp of the problems and needs of both sides. If your users are spacecraft engineers, as mine are, you must understand such things as what a space simulation test is and how it is accomplished. You must learn enough of the user's jargon that his descriptions of the reports or plots he needs don't leave you scratching your head. You must then be able to translate these user needs into a software design.

Do not expect the user to come up with a perfect, never-to-be-changed specification the first time around. This is almost impossible to do. Rather plan for a "controlled evolution" approach to software development. This method can be illustrated in the following example:

- Week 1 -- The programmer approaches the user about specifying how he wants to display data from Test A. The user pulls out some handwritten charts and says "Make it look like this."
- Week 2 -- The programmer returns to the user and says he doesn't have data for part of the chart. The user says "Oh, you just take the log of Column 2 and divide it by Column 3 unless the data was taken during thermal cycling."
- Week 3 -- The programmer shows the user the automated version of his chart. The user is surprised and pleased but asks if he can get the chart matrix inverted as he had never liked the format he had been using.
- Week 4 -- The user checks to see if the programmer has the matrix inverted yet. He then says, "Can you give me a plot of Column 3 for all the data we have so far? Can it be ready by next Tuesday?" The programmer grits his teeth and replies that he can't get it ready by then but he will see about making the plots.
- Week 6 -- With the matrix now inverted and the plotting software underway, the programmer wisely asks if the user has any other data he wants plotted. After some discussion they find several ways plotting software could be used to good advantage.
- Week 9 -- The user stops by the programmer's office and asks if he could have an option on the chart to display only values outside a user-defined limit. The programmer replies in the affirmative but wishes he had mentioned this in Week 2.

The point I am trying to make is that in the beginning the user doesn't really know what he wants automated and how it should look. In some cases he may have a time consuming manual method, such as plotting points pulled from a stack of tables, that prohibits him from doing some of the more sophisticated things he might really be able to use. In other cases, he may never have bothered to look at the data in detail. It's your job to encourage him to view the database computer as a powerful tool that can help him do his job better than ever before. Guide his thinking by educating him to what the computer can and cannot do well for him.

VI. Writing Applications Software That Will be Used

The first rule of applications software display is keep the user informed. Never leave him guessing what the program is working on. If you are searching through 5,000 records, periodically post to the CRT a message similar to "program: SEARCHING RECORD xxx of 5000". If your program runs into a resource problem that cannot be easily worked around (e.g., not enough disc space), tell the user in English what the problem is and what can be done to correct it. An informed user is a happy user.

A corollary to the rule of keeping the user informed is to keep the screen readable. Once you have finished debugging your program, be sure to remove your debug print statements from the executable code. If you use cursor addressing, confirm that you are not overwriting part of one message with another. Watch out for important messages that roll off the edge of the screen or are overwritten 0.5 seconds after they appear. A program that is viewed by the user as indecipherable or mysterious in its operation will probably not be used except under duress. Show that you take pride in your work by presenting professional looking output format.

Avoid forcing the use of long sequences or obscure codes to run applications programs. Most users do not want to memorize strings of numbers. Numbers may be quite acceptable for program-to-program communication, but they are deadly when used for people-to-program communication. A minor lapse of memory or eye-hand coordination can result in a totally different function being performed from the one the user wanted. To solve this I suggest a combination of soft keys and menus in your applications programs. The soft keys allow the user to move from one function to another with the push of a finger. The use of menus permits controlled entry of data values. It is worth pointing out that one of the reasons for the meteoric rise of the popularity of person computers is their heavy reliance on function keys and menus.

Finally, be sensitive to the user's need for fast response. The proverb "time is money" is especially true in the testing environment. If you user cannot get his answer in 15 seconds or less, he will probably not want to wait around for the system to respond. If this happens to you, explore the possibility of optional batch processing. Consider also upgrading your system's memory or disc power. The cost of additional memory is cheap compared to time wasted waiting for a CPU-bound system to respond.

VII. Keeping It All Working

Once the data collection and retrieval software is written, the unglamorous job of database maintenance begins. To avoid having your software fall out of step with the testing process you need at least

one person who is responsible for the health of the database. This champion of the database must be aggressive in his efforts to keep the database up-to-date and on-line. To be effective he must be able to:

1. Monitor the accuracy of the data. An unreliable database is a useless database.
2. Be informed of changes to the testing process that impacts the database. Make sure that the database is always current.
3. Ensure that hardware problems, such as a chronically bad data line, are dealt with swiftly.
4. Monitor the performance of the system and make appropriate system or hardware changes to improve response time.
5. Contact potential new users and give them a positive first impression of the system.
6. Identify changes and enhancements that will improve the database efficiency and pass these on to the programming staff.

VIII. Conclusion

Developing a complex engineering test database is a process that begins when the unit to be tested is still on the drawing board and doesn't end until after the last unit passes the last test. The database must be designed to allow for the variability of the testing process. The concept of a test database must be sold to both users and management as a desirable and achievable goal. Test data must be collected with a high degree of reliability. Display software should be friendly and tailored to the user's needs. These goals may seem like The Impossible Dream, but they are achievable. With careful planning and sustained efforts you can have a test database that is successful in the eyes of your management, your users, and yourself.





**Manufacturing National Account Products
Improve Manufacturers' Productivity
Philip J. Christ
Hewlett-Packard Company
1266 Kifer Road
Sunnyvale, CA 94086**

I. Introduction

For years, Hewlett-Packard (HP) has been involved in factory automation, as a manufacturer and user of computers and instrumentation. The company is also a major supplier of computer-based automation systems to its manufacturing customers, with sales of these systems estimated at \$900 million in 1986.

Through its own manufacturing experience, HP has developed a "think big, but start small" philosophy about factory automation. It recommends this same philosophy to its manufacturing customers. With this philosophy, the manufacturer starts with an overall plan, then looks for opportunities to install individual "islands of automation", such as Cell Controllers or Material Handling systems, using this plan. The manufacturer begins with opportunities offering the greatest potential returns. Once installed, the systems are analyzed for their effectiveness and fine-tuned. An important by-product is that the manufacturer learns more about the manufacturing process. The manufacturer then moves on to the next set of opportunities. Ultimately, the "islands" are linked together in an integrated manufacturing system.

Computer Integrated Manufacturing (CIM) has been characterized in many different ways. Frequently used are hierarchical models showing various levels of automation. Figure 1 (left hand side) shows one such hierarchy. The hierarchy shows a completely integrated system. This is the end goal of the stepwise automation strategy, where the Cell Controllers and Area Managers individually represent "islands" to be integrated. Another useful way of looking at CIM is by considering the applications manufacturers perform at the different levels of the hierarchy. These are superimposed on the right hand part of Figure 1. The focus of this discussion will be applications at the Cell Controller and Area Manager levels of the hierarchy. Key applications in Cell Control and Area Management are shown in Figure 2.

Central to HP's manufacturing automation strategy are third party Value-Added Business partners, which are the distribution channel for a major portion of HP's sales to manufacturers. These companies provide the software and services needed to provide solutions, using HP's standard computers, instrumentation systems and software, that meet individual customer requirements.

II. National Account Program

HP's Manufacturing National Account Program represents a select group of qualified Value-Added Businesses that provide software solutions and services addressing a broad range of Cell Control and Area Management applications. By partnering with its National Accounts, HP is able to provide a more complete solution to meet the specific needs of the manufacturing customer. Currently there are six National Account products that address the following Cell Control and Area Management applications:

- o Assembly Monitoring and Control
- o Quality Monitoring and Control
- o Continuous/Batch Process Monitoring and Control
- o Factory Data Collection
- o Material Handling

Hewlett-Packard intends to expand the National Account Program to incorporate other Cell Control and Area Management applications. The six existing National Account products will be briefly described:

STARNET from Denniston & Denniston is a Cell Control solution, primarily for discrete manufacturing processes, with powerful networking capabilities that supports a wide range of front ends. The networking capabilities allow the user to view an entire networked system as a single manufacturing process.

Monitrol from Hilco Technologies is a low-cost Cell Control solution with interfaces to a range of programmable controllers and instruments. The product provides real-time data to the process operator, allowing improvement of product quality and productivity at the process level. Monitrol is commonly used in batch process and discrete packaging applications. The solution is easily configured by a non-computer knowledgeable user.

The Dispatcher System from Logisticon is a family of integrated material management systems designed to provide real-time control of material movement and efficient utilization of personnel and equipment resources.

RQM from Automated Technology Associates is a real-time Statistical Process Control (SPC) software package. RQM is a scalable set of applications ranging from low-end engineering analysis, to mid-range SPC, to high-end Statistical Process Analysis. RQM has been linked with STARNET to form a combined Cell Control/SPC solution. RQM is particularly effective in high speed discrete manufacturing applications.

AIM from Biles & Associates is a Supervisory Control solution for continuous and batch processes. AIM supports a wide range of front-ends, with emphasis on distributed process control systems and programmable controllers. The solution has full historian capabilities and process graphics.

CAPTURE from Industrial Computer Corporation is a Factory Data Collection solution for applications including Work In Process tracking, product tracking, and labor tracking. CAPTURE utilizes a variety of automatic data gathering devices, including a broad range of bar code readers.

Several application examples will be described, illustrating how manufacturers have implemented National Account products in their facilities and the kinds of benefits they have received.

III. Application Examples

A. STARNET at Eastman Kodak

Eastman Kodak competes in a number of markets which require products incorporating sophisticated electronic assemblies. Examples are new camera systems, photocopiers, film processors, medical products, business and professional products. Further, increasing consumer interest in video photography will demand an even greater dependence on electronics in the future. Kodak's response to this need has been to create its own electronic assembly capacity.

In 1981 the company decided it needed the most advanced electronic manufacturing capabilities available. A new facility was proposed utilizing surface-mount electronic assembly techniques, which were selected for technological reasons. The technology permits a component density up to 5-7 times that of standard leaded, through-hole technology on the same size board. It promotes the maximum utilization of circuit board "real estate" or, alternatively, additional miniaturization of product design. It also provides certain electrical advantages, such as faster circuitry and lower noise.

Included in the design of the proposed facility was an automation system. Kodak made a basic decision not to try to implement a completely integrated manufacturing system all at once, based on the desire to provide engineers and operating people with in-depth experience and knowledge of the process and all its nuances before turning it over entirely to machine control. The facility designers decided to concentrate on a process monitoring and quality feedback system, leaving control automation to a later date. The system was designed to permit product defects to be isolated quickly, thereby preventing the loss or expensive repair of large numbers of high-cost assemblies.

In essence, Kodak was taking the first "step" towards CIM. The capability to make the transition to a more complete factory automation system has been carefully built into the process monitoring and quality feedback system.

Process Monitoring System

The process monitoring system is comprised of an HP 1000 A900 computer running STARNET software from Denniston & Denniston, two HP 3497 data acquisition front ends and two HP 9000 desktop computers to collect the process data. It was important that the system be able to integrate ten different machines from six different suppliers into a homogeneous information system. The machines were connected to the HP 9000s either directly via RS-232 links or through the HP 3497s. One of the key contributions of the STARNET software is its device-independence which gives users the needed capability to view the entire facility as an integrated system.

STARNET is responsible for monitoring all the datapoints of the process, collecting the data in real time, and passing the process information to one or more terminals located throughout the facility, where engineers can view reduced

data on process control charts. The system also provides technicians with an overall facility display, which visually alarms technicians if a data point on a machine goes out of specification.

Quality Feedback System

As circuit boards come off the assembly line, they proceed to one of four inspection stations, each of which is equipped with an HP Touchscreen terminal. Kodak found the Touchscreen particularly effective in helping users learn how to use the system with minimal training. Product design data fed into the HP 1000 is displayed on the Touchscreen monitor as a pictorial representation of the assembled board, with an indicator box for potential solder and placement faults. Should the technician detect a board defect, he touches the displayed component matching the one he has found defective, bringing a red dot onto the screen at that point, signifying the component is being inspected. He then touches the fault type listed in the fault box. This information is then entered into the HP 1000 and stored in the quality database. The system provides engineers with a count of the number of boards inspected and presents a current defect rate in parts per million.

Downloading Machine Programs

Product design data being fed to the Touchscreen inspection station monitors comes from the division's Computer-Aided Design system, a Digital Equipment VAX, which stores board design data. Via HP's NS/VAX link, HP provides Kodak the ability to download CAD data not only to the inspection stations, but to some of the assembly machines as well. For example, Kodak uses the link to provide a first-cut program to the placement machines. Kodak designers have determined such factors as optimum board travel and have written programs for the HP 1000 which enable it to convert CAD data into part placement programs.

Results

- o **Fast start-up.** Once the assembly equipment was selected, installed and wired for data acquisition and the system design was completed, two engineers and a part-time programmer had the initial versions of the process monitoring and quality systems in place and operating in 2 weeks.

- o **Reduced defects.** The process had typical defect rates of 1200 ppm when the system was first installed. After 4 months the process demonstrated rates less than 100 ppm.
- o **Improved product design.** With one customer's product, inspectors found that two components which were experiencing repeated solder-bridging faults were spaced too closely. The process engineer recommended moving the parts several thousandths further apart. Defects on the next lot were reduced from 2700 ppm to 250 ppm.

B. The Dispatcher System at Raytheon Co.

The Electromagnetic Systems Division (ESD) of Raytheon Co. manufactures electronic counter-measures equipment for jamming hostile radar. These products are used chiefly on military ships and aircraft. Because ESD's customer is the United States government, the plant's record keeping and inventory practices must conform to government requirements. To streamline its response to these requirements and to better supply assembly operations, ESD reconfigured its storeroom for subassembly components.

Material Handling System

Today, in addition to conventional shelving, the storeroom has four new carousels with automatic, mechanical extractors. Totes of parts move between the carousels and four workstations on a special belt conveyor (transporter). These storage-and-retrieval transactions and all other storeroom operations are computer controlled using The Dispatcher System software from Logisticon, Inc.

The storeroom that houses the storage system performs the following functions:

- o Kitting--collection of components for subassemblies into kits for assembly operations.
- o Filling orders for spare and replacement parts.
- o Supplying missing items to assembly areas when kits are "short" items.
- o Replacing any failed or damaged items for the assembly areas.
- o Maintaining inventory records in strict accordance with government procurement practices.

Work Flow

When components arrive in the storeroom, they are segregated into small items and bulk items. From that time until the parts go to assembly, the storeroom computer tracks both the small and the bulk items in real-time with the The Dispatcher System software.

Following inspection, the storeroom computer assigns shelf locations to bulk items, which then are stored manually. Small items are delivered to workstations for storage in the carousels.

All totes in the carousels are captive to the system. Each is a standard size, but is divided by corrugated containers into multiple storage locations.

To store parts in the carousels, an operator at one of the four workstations determines the container size needed and uses the station's computer terminal to request a tote with an empty container that size. The system then automatically delivers a tote.

Affixed to each tote is a unique bar code. When a tote arrives, the storeroom computer requests the operator to scan the code to verify that the tote is the correct one. Following this verification, the computer visually shows the operator where in the tote to place the parts to be stored.

After the parts are placed in the designated location, the operator communicates to the computer that the transaction is completed and releases the tote to the transporter for return to storage.

When kits are picked, the kits usually contain both large and small parts, requiring that both the carousels and the shelves be accessed to make up a kit. This means that The Dispatcher System must sequence both picks from the shelves and picks from the carousels. Then it must instruct operators so that the two collections are combined correctly for delivery to assembly.

Each of the four workstations can hold two totes in queue. This feature helps bring about the productivity gains mentioned by Raytheon management.

Computer System

Each carousel is controlled by its own programmable controller (PLC). The facility's IBM mainframe computer downloads requests for materials to the storeroom computer, an HP 9000. Storage and retrieval commands are downloaded from the storeroom computer to the PLCs. The PLCs, in return transmit transaction data to the storeroom computer for use in determining subsequent transactions and for inventory record keeping. The storeroom computer maintains inventory records based on data supplied manually from the terminals and automatically from the PLCs. It generates commands--store, retrieve, pick, place, verify, etc.--based on local conditions and pre-programmed procedures as well as real-time requirements in assembly. There is a data link between the storeroom and the facility mainframe computer. Both computers maintain inventory records for the storeroom. The storeroom computer's records are based on part numbers and locations. The facility computer uses part numbers and dollar values. The two sets of data are reconciled weekly.

Results

- o **Effective space utilization.** The carousels, the transporter and the computer system use less than 25 % of the storeroom's floor space.
- o **Increased productivity.** Estimated to be 60% higher, this is mentioned by nearly every manager whose operations are affected by the system.
- o **Better inventory control.** Parts are tracked in real time with an accuracy of 97.84%, compared to accuracies as low as 85% before installing the system.
- o **Reduced materials cost.** This is due to the tighter materials control possible with the system.
- o **Better physical security.** Because storage in the carousels is random, no one--except the computer--knows the specific locations of parts.
- o **Efficient cost transfers for residual materials.** This is where some of the biggest savings take place. Residual materials originally assigned to one contract can be transferred to another contract without physically moving them.

C. Monitrol at a Metals Processor

One of the largest mineral separation mills in the world is located in rural central Missouri. The company maintains a small maintenance staff with the nearest local support approximately 150 miles away. The mill process was manually controlled. Because of the long time constant of the process, results of operator adjustments could not be determined for 2 to 3 hours. This was compounded by the wide variability in the mineral content of the mined rock. Yields from the process were inconsistent and low. Operators were using excessive amounts of expensive chemical reagents to maximize leaching of the desired minerals from the rock. Downtime was excessive. Process engineers had no way of accurately tracking downtime or operator performance.

Process Monitoring and Control System

Two HP 9000 computers were installed with Monitrol software in a "Hot Backup" configuration. The computers were interfaced to an existing PLC. The PLC controlled levels, flows and variable speed pumps. The compositions of all major streams were monitored by an in-line X-Ray analyzer. Data from the analyzer was used to optimize control of the reagent flows. Monitrol, operating on the analysis data with control algorithms, downloaded computed setpoints to the PLC. The PLC controlled the speed of variable speed pumps based on the computed setpoints, adjusting the flow of the reagents.

All operator alarms are logged and operator-entered failure data are archived for maintenance records. Production and downtime reports are generated automatically at the end of each shift. Daily, weekly and monthly summaries are maintained by the system.

Results

- o Increased yields and reduced reagent usage. Resulted in a payback of approximately 9 months for the system.
- o 100% uptime. In over 2 years of operation.
- o Acceptance of system by operators. Minimal training was required. Configuration changes, based on newly acquired knowledge, were supported by existing personnel who had no prior experience with computer control systems.

- o Minimal changeover time. Installation and start-up was accomplished in less than one week, after which the user fully supported the system.

IV. Summary

Hewlett-Packard's Manufacturing National Account Program represents a select group of Software Suppliers providing solutions on HP systems for a number of factory automation applications. These products can be used as the basic building blocks for a manufacturer's CIM strategy. By implementing solutions one step at a time, manufacturers can realize significant improvements in productivity and product quality and, in doing this, learn much about the manufacturing process itself. Examples have been given of the kinds of gains several manufacturers have realized by taking the first step towards CIM. These successes are expected to ultimately contribute to more effective implementations of integrated factory automation systems.

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

**K. S. Burchette, J. D. deBethizy, T. G. Ashby, T. J. Hellard,
and M. B. Johnson. R. J. Reynolds Tobacco Company,
Bowman Gray Technical Center, Winston-Salem, NC 27102**

INTRODUCTION

One of the primary activities of the Pharmacokinetics Group of the Toxicology Division of the Research and Development Department of R. J. Reynolds Tobacco Company is performing radiolabeled xenobiotic disposition studies. A disposition study is used to determine how a compound, upon exposure to an animal, becomes distributed throughout the animal's body. A typical metabolic caging system for the animals and the types of samples that are collected during disposition studies are shown in Figure 1. Each animal is treated with a radiolabeled analog of the compound so that the parent compound and the resulting metabolites can be monitored as the chemicals are distributed to the tissues and eliminated in the excreta. If this is done for samples taken at a number of time intervals, a pharmacokinetic analysis of the disposition of the chemical and its metabolites can be achieved.

Disposition studies require the collection and analysis of a large number of samples. This is both labor-intensive and routine. Therefore a computerized system was needed to facilitate writing the protocol, collecting the samples, analyzing the samples for radioactivity, reporting the data, and storing the data for future retrieval and further analysis. In addition, this system would voluntarily meet Federal Drug Administration Good Laboratory Practices guidelines for computer systems. Such an all-inclusive system was not commercially available. Therefore, the Information Systems Data Acquisition Group of the Technical Services Division of R. J. Reynolds Tobacco Company designed and implemented a system for conducting xenobiotic disposition studies.

Copyr. 1987. R. J. Reynolds Tobacco Company. All rights reserved. The computer software system described in this paper is the exclusive property of R. J. Reynolds Tobacco Company and may not be used or copied except with its written permission.

COMPUTER HARDWARE, SOFTWARE, LABORATORY EQUIPMENT

It was determined early in the system specification phase that the system would reside on existing hardware. Therefore the system was designed so that it would be as portable as is practicable with emphasis for a possible port to an HP 1000A or to an HP 840. For the ease of defining studies and for queuing samples for data acquisition, block mode CRT's are used to allow the technologist to fill in forms. The required reports dictate the need for a dedicated printer and plotter. In addition, previewing of graphical reports using CRT's rather than a plotter are included as a time-saving feature; so graphics CRT's are used in some locations. The complex interconnections of the sample identifications and data require that all information ultimately reside in structured files, rather than in flat files. During data acquisition, flat files are sufficient for temporary storage of sample identification and the associated data. A microcomputer-based scintillation counter and high pressure liquid chromatograph with a radioactivity detector are required to pass results as each sample is processed.

A Hewlett-Packard 1000E mini-computer with the RTE 6/VM operating system is used. In addition, Hewlett-Packard CRT's (2692's, 2693's, 150, and Vectra), printers (2934 and ThinkJets), and plotter (7475) are used. The software packages that are used include: IMAGE(Hewlett-Packard), a database management system; FORMS/1000(HP), a CRT screens manipulation package for manual data entry; and GRAPHICS 1000-II(HP) for CRT and plotter graphics. In addition, GRAFIT, a Fourth Generation Language graphics package from GRAPHICUS, Incorporated is used to facilitate graphical display. The programming language is FORTRAN 77(HP).

For laboratory instrumentation, Mettler PE 1600 and Mettler AE 163 balances, a Packard CA 2000 liquid scintillation counter, and a high pressure liquid chromatograph with a Waters Ramona LS 4 radioactivity detector are used.

A minimum amount of computer and laboratory equipment is used to minimize the amount of space utilized in the laboratories. For dosing, animal sampling, necropsy work, and the associated worksheets, a CRT and ThinkJet printer on a cart are moved between the animal room and the necropsy room located in the animal facility. Each analytical and gross weight balance share a computer port with a CRT with a special 'Y' cable arrangement which can be quickly disconnected, moved, and reconnected. A CRT with the connected analytical balance, the scintillation counter and the high pressure liquid chromatograph are located in the main laboratory. In addition, the printer, plotter, Vectra, and other CRT's are available in the main laboratory. Additional computer equipment and instrumentation of the same types can be added with little effort when the sample load justifies the use of more equipment. All connections are RS-232 to a multiplexer.

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

DICTIONARIES: INSTRUMENT TESTS, BIOLOGICAL SAMPLES, CALCULATIONS

A very significant task was to design a database that would accommodate disparate samples. For the simplest case, some biological samples would require size determinations in volumes: urine, captured carbon dioxide, cage wash; the organs and feces would require gravimetric determinations. Several variations had an impact on the amount of data acquired and reported. Some samples required only total radioactivity analyses: captured CO₂, cage wash, blood; most required metabolite radioactivity analyses: urine, plasma, all organs, feces. Because activated charcoal may not release all substances into a solvent, it required further laboratory work to determine the total radioactivity in the sample prepared for metabolite determinations; all other biological samples were assumed to yield to the solvent. Since it was assumed that all biological samples except feces have the distribution of radioactivity homogeneous throughout that sample, feces is homogenized in a solvent to provide aliquots for determination of total radioactivity. To determine the total amount of radioactivity in each sample and the amount of radioactivity associated with the metabolites of the administered chemical, each biological sample required its own set of calculations.

The database was designed to incorporate these sample nuances. It contains three master sets. The protocol set contains the descriptive information for each definition. Another set describes the groups of animals. The third master set contains the information that describes the biological sample information and another set that describes the instrumentation information. There are six detail sets. Two sets describe the instrumentation information. One set holds individual animal information. One is used for Notebook entries. Two sets hold the data for total radioactivity and for metabolite radioactivity. Within these two sets are special records. Each biological sample type has, in each set, two special records. One record determines the instruments that perform the tests, or no test required. The other record determines the default values for sample-specific data, such as volume to inject onto the HPLC.

The protocol number is defined as an ASCII field of five numerals. The first two numerals are the current year's decade and unit values; the last three numerals are sequential as protocols are added. For example 86999 is the 999th protocol in 1986, and 87001 is the first protocol in 1987. The animal group is defined by a single alpha character, A through Z. The biological sample, also referred to as a biotest, is defined as an ASCII field of three numerals, 001 to 999. The animal number is a one word integer value, 1 to 999. Sample times are floating point numbers recorded as offsets from dose time. Pacing of animal dosing at different times of the day is done so that the timing for collection of samples is reasonable for the animal technician.

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

THE SAMPLE SET

The biological sample set in addition to the protocol number contains the following:

1. The number associated with the biological sample.
2. The biotest name that is to be used throughout the study and reports.
3. The type of organ to be taken at necropsy. The study director can choose among three possibilities: a defined organ that can be completely removed and weighed, a diffuse organ (referred to as a partial organ) that requires a literature value to estimate the total size of the organ based on the percent weight of total body weight, or non-organ such as excretion products.
4. The organ dissection order.
5. The size type for reports which determines whether the data is expressed as either "G" for grams or "ML" for milliliters.
6. The literature value for the percent weight for partial organs.
7. A solvent correction flag which indicates whether to use a solvent to homogenize the sample and thus later provide corrections to the data to remove the dilution effect produced by adding the solvent.
8. A total/metabolite flag which is used to determine whether total radioactivity data can be used for the metabolite calculations or if further homogenizing and filtration (and resulting data) must be used. This flag represents a concept that, if it has been previously determined that there is no radioactivity left behind in the residue, that biological sample's total radioactivity data can be used, representing a very significant savings of time and labor.

THE INSTRUMENT SET

The instrument set, in addition to the protocol number, has several items that are used for data acquisition from a specific instrument, as shown in Figure 2. Each instrument test has a unique name and number. The number is used internally to the system, and the name is displayed to the user. The fraction in the instrument test number indicates the number of positions in the database reserved for replicate samples in other sets. The instrument code is used in file-naming conventions during data acquisition. The instrument LU (logical unit) specifies whether the data is gathered at the same location as the user requests samples for data acquisition(1), or at another location where the data will be acquired unattended. For example, the instrument test number 5.0 refers to DENSITY, there are no replicates, the data must be entered

from the terminal (manual) , and the logical unit is 1 so that the instrument(the terminal) must be attended for each value. Test number 6.3 refers to scintillation volume with a total of 3 replicate positions, and it is manual. Test number 8.3 refers to a scintillation counter with 3 replicate positions and the data will be acquired at logical unit 30, unattended. Test number 17.3 refers to two HPLC's, one at logical unit 40 and the other at 41. The user can choose either HPLC with which to acquire data for the sample. For space considerations it was decided that three replicates would be typical. All of these positions are not required, and multiple records allow for more than three replicates.

OVERVIEW OF LABORATORY TASKS:

The tasks for this system are logically divided into five areas, each using its own log on procedure:

1. Protocol definition and modification and status modification which are performed by the principal investigator.
2. The in-life portion of the study including animal dosing, sampling and necropsy work which are performed by the animal technicians.
3. Sample testing which is performed by the laboratory staff.
4. Report generation which is performed by the principal investigator and the laboratory staff.
5. Historical data storage and retrieval which is performed by the Information Services group.

The appropriate person may log onto the system under one of these areas, if he qualifies with the security check.

In the examples that follow, please do not view the data as representative of a valid study. The examples are taken from dummy data used during a testing phase of the system.

1. PROTOCOL DEFINITION AND MODIFICATION, STATUS MODIFICATION

Upon logging on to the protocol area, the principal investigator (PI) encounters the main menu [Fig. 3]. The "Define and Initialize Protocol" choice leads to a number of menus allowing the origination of a protocol, including setting the protocol status to "initialization." Upon completion of the origination phase of the protocol, the PI can change the status to "testing" under the "Setup Protocol for Testing Phase" so that data can now be acquired under this protocol number. He can choose the "Modify Protocol During Testing Phase" for the limited number of modifications that are permitted under Good Laboratory Practices. Another choice is "Update Notebook" wherein he may enter any

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

observations into a notebook area. Any modification to the protocol automatically forces an entry to be made into the Notebook to justify the modification. This feature provides an audit trail that will permit accurate data audits. Upon completion of all data acquisition, the PI will choose "Finalize Protocol" to change the protocol status so that no further data can be acquired [Fig. 4]. Now all reports, both tabular and graphical will show "Final". Before finalizing the protocol the reports will have shown "Preliminary". Once all reports are printed, the principal investigator chooses the "Complete Protocol" to change the status to "Complete". This results in a message being sent to the Information Services system console to archive the data and then delete it from the database. No other reports can be printed. Had the PI previously requested archived data to be reinstated to the system, the status on the reinstated data can be changed so that it will be deleted from the system. Reports can be generated by selecting the "Report Menu" which allows the PI to select the types of reports to be generated.

As stated, the "Define and Initialize Protocol" menu choice leads to several other screens. The first two of these allow entry of general information [Fig. 5] and test material information [Fig. 6]. The next screen begins the animal group information [Fig. 7]. Animals are grouped so that these groupings may have statistical analyses performed after the data have been acquired. The following screen shows choices of samples [Fig. 8]. Each of these choices is followed by a screen for setting values for later use [Fig. 9] such as solvent density, volumes for scintillation counter filtrate, HPLC, and replicates of these. Following the definition of the biotest, the testing times are chosen [Fig. 10]. Each of the animal groups are defined with a number of biological tests and sampling times for those tests. During the testing phase, if it has been determined that certain tests will not be performed even though the tests were defined during the definition of the protocol, the principal investigator can select those tests and have the system mark them as unrequested, provided he makes an entry into the Notebook. The peaks from the HPLC are defined in order of elution, with peak name and relative response time indicated [Fig. 11].

2. ANIMAL DOSING, SAMPLING, AND NECROPSY

Once the protocol is defined, the work in the animal rooms can begin [Fig. 12]. Upon receipt, the animals are weighed and placed in standard cages for an acclimatization period. After this period, the animals are weighed, and the weight and protocol information are used to display to the technologist the dose volume to be administered to the animal. As soon as the dose is administered, the technician presses carriage return on the CRT. System time is determined and stored in the database as the time of dosing for the animal. A sampling worksheet listing the times for sample collection is printed and posted for each animal [Fig. 13]. In addition, the necropsy worksheet for that animal is also printed [Fig. 14].

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

As samples are collected, the sample size (volume or weight) is recorded using the CRT in the animal room. To do this, the technologist proceeds through screen choices of animal group; sample time; biotest such as feces, urine, or blood[Fig. 15]; instrument test of either total size weight or total size volume[Fig. 16]. He chooses all of the samples that fit the categories of interest and then proceeds to acquire the information for the sample as its identification is displayed.

At necropsy time, the technologist logs on to the system and chooses an animal group[Fig. 17], and then is presented, one animal at a time, with a list of organs to be removed in the order of dissection[Fig. 18] as predetermined at the protocol definition. The system is designed to provide him with the least number of key strokes to accomplish the weight or volume measurement of the tissues.

3. SAMPLE TESTING

Once a sample has been collected, it must be analyzed for total radioactivity. For instrument tests that are volumetric or weight determinations, the screens appear much as described above. For instrument tests that use the scintillation counter or the HPLC, there are differences. These differences are caused by the ability of the instrument to sequence through a number of samples and return results unattended by the technologist. The main screen for sample testing[Fig. 19] shows "SETUP FOR DATA ACQUISITION" so that a number of samples can be queued for the instrument, "ACQUIRE DATA" so that the data communications with the microcomputer controlling the instrument can be established, and "APPROVE DATA" so that the technologist can, at a later time, review the raw data and determine whether to accept the data or recycle the samples through the instrument. The technologist may start the automatic acquisition of data for an instrument and then go back to the "SETUP FOR DATA ACQUISITION" screens to append more samples to the queue for that instrument. As acquisition is completed, the status for the samples is changed to "Complete" so that the technologist can use the "APPROVE DATA" choice to look at the data [Fig. 20] and allow approval or not. On approval the data is automatically copied to the database.

This is a display of a file for which data has already been acquired. In all cases, because the data is received in ASCII form, the entire record structure is held to ASCII. Although the technologist goes through the setup phase with grouped samples, the records in the file are defined in such a way that no grouping needs to be implied. The record that starts with "I" is an identification record. It is from instrument lu 53, for protocol number 86999, biotest 001, group A, animal 310, for a sample taken at 48.00 hours. It is for record 1 in the set TOTALD, item SCCT1. The records that start with "D" are data records, "DL" implies the last data record for that identification. Using the "DL" designation allows the software to know that the last record of data has been found for that identification without reading

the next record to discover whether it is another "D"ata record or an "I"dentification record. For samples that have no multiple data records, such as weighing tissues at necropsy time, the records would alternate between "I" records and "DL" records. Upon setup, each data value is registered as "?". As each value is received, the "?" is replaced with the value, sequencing through the file. In attended data acquisition such as from a balance, the sample identification is displayed to the technologist. Upon receipt of the requested value, that result is also displayed. For unattended data acquisition such as an HPLC, the technologist views the results later. He has the capability at the approval stage of resetting values to "?" so that the incorrect samples can be cycled through the instrument again and have data acquired for only those incorrect ones, such as might occur if a sample vial is placed out of sequence. Once he approves this data and it is sent to the database, it requires intervention by the Principal Investigator to reset the sample items to be reacquired, and an accompanying entry into the Notebook for an audit trail.

4. REPORT GENERATION

Data reporting can occur at any time. Missing data are indicated with asterisks. The reports can be sent to the CRT with a pause at each screenful of data for ease of viewing or can be sent to the printer. In addition to tabular reports, these can also be graphical reports [Fig 21]. The reports are for data from either the determination of the total radioactivity in the biotest [Figs. 22,23] or from the metabolites in the biotest [Fig. 24]. Each report can be in percent dose per gram (or ml) of tissue, percent dose per tissue, nanomole per gram (or ml) of tissue, or nanomole per tissue.

In addition to these reports there are other reports: percent recovery of radioactivity for a biotest [Fig. 25], Notebook, and dose administration [Fig. 26]. Depending on the status of the protocol, these reports are marked "Preliminary", "Final", or "Historical".

5. HISTORICAL DATA STORAGE AND RETRIEVAL

The archiving of the data to magnetic tape is requested by the principal investigator automatically as a result of declaring the study complete. The Information Services staff's main screen shows "Copy database to files" for preparation of archiving to magnetic tape, "Copy files to database" for retrieving data from magnetic tape, and "Delete data from database" for removing protocols from the database once the study has been archived or else no longer needed if it has been retrieved from magnetic tape. The request to "Copy database to files" results in all data from that protocol number being copied to files, one file per set. A standard Hewlett-Packard magnetic tape utility appends those files to a magnetic tape. The status of the protocol is changed so that when the "Delete data" option is chosen, all data for that protocol number is deleted from the database. For retrieval from magnetic tape, the

standard utility is used to copy that data for that protocol number to files. Then the "Copy data from files to database" option copies that data from the files into the database, marking the status so that the data will be reported as "Historical." No data can be acquired, altered, or partially deleted when it is deemed historical. It can be either reported or deleted in total.

SUMMARY

The most descriptive word for this computer system is flexible. The flexibility appears in

1. Protocol definition
2. Biotest definition
3. Instrumentation definition
4. Calculations definition
5. Sample test selection
6. Report choices
7. Historical data storage and retrieval.

In addition to its flexibility, it is a system purposely designed to be easy to use. The laboratory staff familiar with disposition studies and animal technicians require an hour or so training to be able to use the software. The principal investigator has software that, while it is as easy to use as the other, requires more knowledge because the flexibility of the system dictates the capability of so much redefinition.

CONTRIBUTIONS

In addition to the co-authors, the following people have contributed to the system and to the paper: Lisa E. Bates, Alston Hildreth, Jr., D. Lee Solomon, Mary Ann Watkins, and Dorothy A. Ward.

The Anatomy of a GMC

(Glass Metabolism Cage)

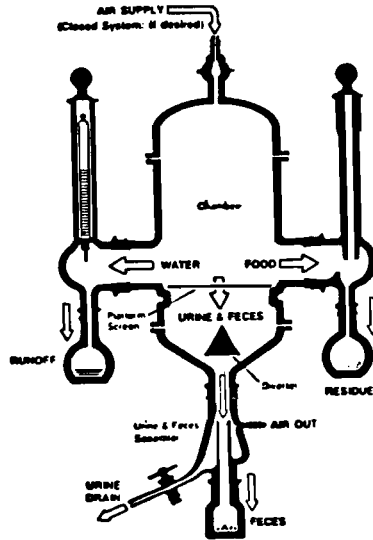


Fig. 1

Inst Test Name	Inst Test #	Instrument	Inst Code	Inst LU
DENSITY	5.0	Manual	M	1
SCINT VOLUME	6.3	Manual	M	1
SCINT COUNT	8.3	Scintillation Counter	S	30
HOMOG WEIGHT	9.0	Balance	B	1
HPLC PK	17.3	High Pressure Liquid Chromatograph	L	40, 41

Fig. 2

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

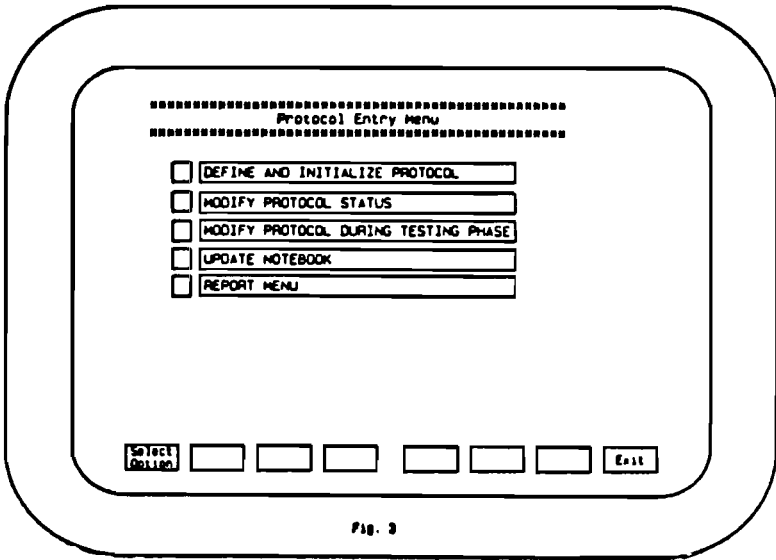


Fig. 3

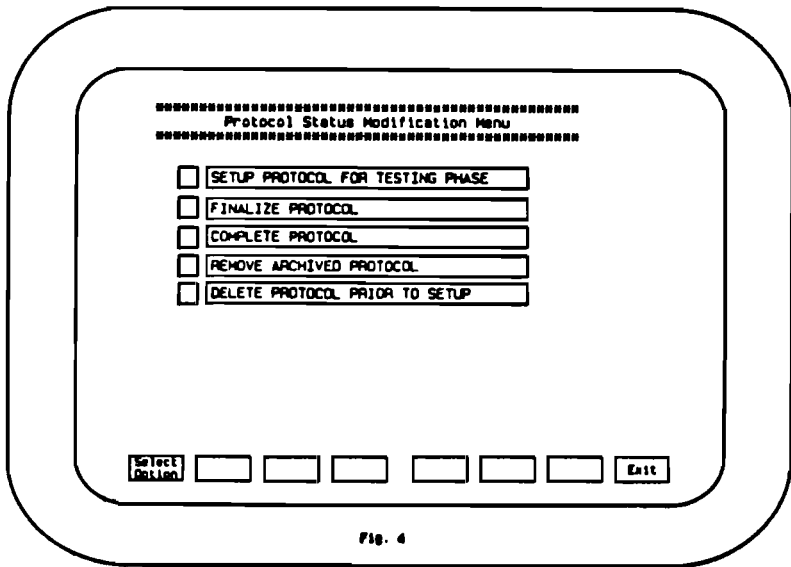


Fig. 4

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

Protocol Definition Input Screen Status IN

Protocol # -

Protocol Name: PHARMACOKINETICS OF NICOTINE INTRAVENOUS

Study Director: 1 J. DONOHUE
2 _____

Principal Investigator: 1 LISA E. BATES
2 _____
3 _____

Technologists:

1 <u>Steve Gentry</u>	2 _____
3 _____	4 _____
5 _____	6 _____

Species: MALE

Strain: SPRAGUE-DAWLEY No. of Groups: No. of Animals:

Acclimation Period: Sacrifice Method: CO2 ASPHIX

Supplier: CHARLES RIVER LABORATORIES

values MUST be entered in HIGHLIGHTED fields. Other fields with () OPTIONAL.

Update Entry Next Screen Exit

Fig. 5

YYPYP Test Material Input Screen Status IN

Test Material Name: 14C-2-PRIPROLOL (R)-NICOTINE

Vendor: AMERICAN

Lot Number: 123456

Purity (%): 98

Specific Activity (uCi/umole): 500

Update Entry Next Screen Exit

Fig. 6

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

86001 Group/Animal Input Screen Status IN

Group: Max Time (hrs.): Sex: An # An # An # An # An # An #

Husbandry

Cage System: CLASS KEY/ABSL/SLK ROOM TYPE

Feed: PURINA ROCKET FEAL #5002 AD LIBITUM

Water: TAP WATER IN BOTTLES AD LIBITUM

Environmental Conditions: TEMPERATURE 72-74° HUMIDITY 50-100

Dose

Pretreatment Period (days): 0 1 2 3 4 5 6 7 8 9

Route of Exposure: ORAL INTRAPERITONEAL INTRAVENOUS SUBCUTANEOUS INTRACUTANEOUS INTRAMUSCULAR INTRAARTICULAR INTRAVAGINAL INTRAVULVAR INTRARECTAL INTRACEREBRAL INTRACEREBELLAR INTRACEREBRAL INTRACEREBELLAR INTRACEREBRAL INTRACEREBELLAR

Vehicle: 0 1 2 3 4 5 6 7 8 9

Dose of Test Compound (umole/kg body wt): 0 1 2 3 4 5 6 7 8 9

Conc of Rad in Dose (dpm/ml): 0 1 2 3 4 5 6 7 8 9

Specific Activity (uCi/umole): 0 1 2 3 4 5 6 7 8 9

Total Dose of Rad to Animal (uCi): 0 1 2 3 4 5 6 7 8 9

(Note: Total Dose Value required for inhalation study.)

Values MUST be entered in HIGHLIGHTED fields. Other fields with [] OPTIONAL.

Fig. 7

86001 Select Biotests for Group Status IN

URINE FECEES BLOOD PLASMA CAGE MASH ACT DNA COE PARTIAL ORGANS ORGANS

Enter 'X' to select Biotests for this group.

Fig. 8

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

Biotest for Group Status IN

Total 14C Biotest Manual Entry

Total Vol, Wt, or Mt based on

 Vol to Wt Ratio

 Density of Solvent

 Scint. Counter Vol or Mt

 Scint. Counter Count

Metabolite 14C Biotest

Weight for Homog.

 Filtrate Vol

 HPLC Vol

 Scint. Counter Homog. Mt.

 Scint. Counter Homog. Count

 Scint. Counter Filtrate Vol

 Scint. Counter Filtrate Count

 HPLC peaks

0 - NO TEST 1 - TEST, NO REPS 2 - TEST, 2 REPS 3 - TEST, 3 REPS, ETC.

Fig. 9

Biotest Status IN

for Group with Max Time (hrs.)

<input type="text" value=".05"/>	<input type="text" value=".062"/>	<input type="text" value=".166"/>	<input type="text" value=".333"/>	<input type="text" value=".9"/>	<input type="text" value=".75"/>	<input type="text" value="1.0"/>
<input type="text" value="1.25"/>	<input type="text" value="1.5"/>	<input type="text" value="2.0"/>	<input type="text" value="3.0"/>	<input type="text" value="4.0"/>	<input type="text" value="5.0"/>	<input type="text" value="6.0"/>
<input type="text" value="9.0"/>	<input type="text" value="24.0"/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>
<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>
<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>	<input type="text" value=""/>

Enter sample times for this Biotest

Fig. 10

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

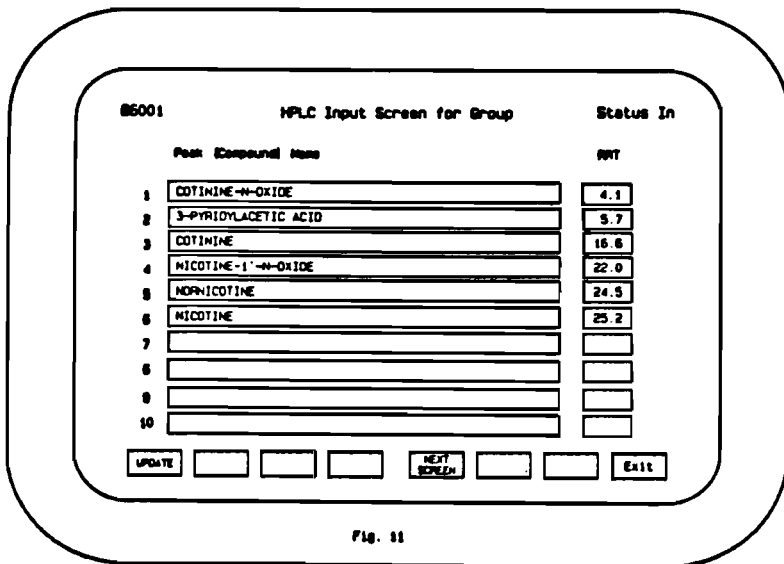


Fig. 11

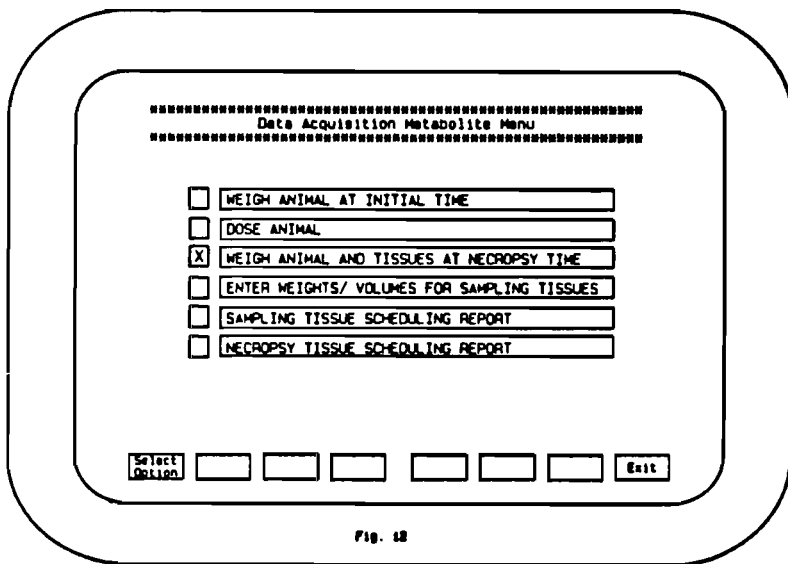


Fig. 12

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

PRELIMINARY REPORT

Date: FRI., 13 MAR., 1987 Time: 8:13 AM

Report/Protocol # TAD-ST5-86-100
 STUDY IN FEMALE RABBITS
 Sample Schedule Work Sheet
 Species: RABBITS Strain: BLACK
 [14C-2'-PYRROLIDINE]NICOTINE
 Animal : 0310 Group: A

Date	11/05	11/05	11/05	11/05	11/05	11/05	11/05	11/05	11/05
Time	11:09	11:12	11:13	11:20	11:39	11:54	12:09	12:24	12:39

* URINE
 - FECEES

PRELIMINARY REPORT

Date: FRI., 13 MAR., 1987 Time: 8:13 AM

Report/Protocol # TAD-ST5-86-100
 STUDY IN FEMALE RABBITS
 Sample Schedule Work Sheet
 Species: RABBITS Strain: BLACK
 [14C-2'-PYRROLIDINE]NICOTINE
 Animal : 0310 Group: A

Date	11/05	11/05	11/05	11/05	11/05	11/07
Time	13:09	14:09	16:09	17:09	19:09	23:09

* URINE
 - FECEES

Fig. 13

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

Report/Protocol: 8 TRD-S75-06-100
STUDY IN FEMALE RABBITS
Necropsy Worksheet
Species: RABBITS Strain: BLACK
114C-2'-PYRIDOLINE NICOTINE
Animal: 0310 Group: A

1. (013) MAMMARY GLAND
2. (014) SALIVARY GLAND
3. (015) PANCREAS
4. (016) SUBMAXILLARY
5. (017) SUBMANDIBULAR
6. (018) THYMUS
7. (019) THYROID
8. (020) TRACHEA
9. (021) ESOPHAGUS
10. (022) LUNG
11. (024) RIGHT CRANIAL
12. (025) LOBES
13. (026) RIGHT DIAPHRAGMATIC
14. (027) HEART
15. (028) EYES
16. (029) BRAIN
17. (030) CEREBRUM
18. (031) CEREBELLUM
19. (032) MID BRAIN
20. (034) PITUITARY
21. (035) ZYMBALS GLAND
22. (036) NOSE
23. (037) EXTERNAL NARES
24. (038) NASAL EPITHELIUM
25. (039) OLFACTORY EPITHELIUM
26. (040) GALL BLADDER
27. (041) LIVER
28. (042) RADIAL
29. (044) RIGHT LATERAL
30. (045) STOMACH CONTENTS
31. (046) FORESTOMACH
32. (047) GLANDULAR STOMACH
33. (048) SMALL INTESTINE
34. (049) SMALL INTESTINE CONTENTS
35. (050) DUODENUM
36. (051) JEJUNUM
37. (052) ILEUM
38. (054) LARGE INTESTINE
39. (055) LARGE INTESTINE CONTENTS
40. (056) COLON
41. (057) RECTUM
42. (058) PANCREAS
43. (059) SPLEEN
44. (060) ADRENALS
45. (061) KIDNEYS
46. (062) BLADDER
47. (064) SEMINAL VESICLES
48. (065) UTERUS
49. (066) OVARY
50. (067) SCIATIC NERVE
51. (068) FEMUR
52. (069) SPINAL CORD
53. (070) CARCASS

Fig. 34

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC
DISPOSITION STUDIES

Biostat Selection Screen Protocol # -

<input type="text" value="FECES"/> <input checked="" type="checkbox"/>	<input type="text" value="PLASMA"/> <input type="checkbox"/>	<input type="text" value="URINE"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>
<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>	<input type="text"/> <input type="checkbox"/>

Fig. 15

Instrument Test Selection Screen Protocol # -

Biostat

Test	Instrument No.	Test	Instrument No.
<input type="text" value="URINE WEIGHT"/> <input type="checkbox"/>	(1) (max= 1)	<input type="text" value="SPIN VOLUME"/> <input type="checkbox"/>	(1) (max= 1)
<input type="text" value="HDOB WEIGHT"/> <input type="checkbox"/>	(1) (max= 1)	<input type="text" value="CHLOR PH"/> <input type="checkbox"/>	(1) (max= 1)
<input type="text" value="SPIN VOLUME"/> <input type="checkbox"/>	(1) (max= 1)	<input type="text" value="BIOTEST COUNT"/> <input checked="" type="checkbox"/>	(1) (max= 1)
<input type="text" value="BIOTEST VOLUME"/> <input type="checkbox"/>	(1) (max= 1)	<input type="text" value="BIOTEST PA VOL"/> <input type="checkbox"/>	(1) (max= 1)
<input type="text" value="BIOTEST HDOB CT"/> <input type="checkbox"/>	(1) (max= 1)	<input type="text" value="BIOTEST HDOB INT"/> <input type="checkbox"/>	(1) (max= 1)

Fig. 16

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

Select ONE Group to set up Necropsy Weights for Protocol #

Select Group

Groups Available for this Protocol

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

Group Selected:

Enter ONE group letter available in the protocol. Press 'Select Group' key.

Fig. 17



Animal - Necropsy Weigh Selection Screen Protocol # -

Group Animal No. Select All (this animal)

<input type="text" value="FINAL WEIGHT"/> <input type="checkbox"/>	<input type="text" value="MAMMARY GLAND"/> <input type="checkbox"/>	<input type="text" value="SALIVARY GLAND"/> <input type="checkbox"/>
<input type="text" value="PAROTID"/> <input type="checkbox"/>	<input type="text" value="SUBMAXILLARY"/> <input type="checkbox"/>	<input type="text" value="SUBMANDIBULAR"/> <input type="checkbox"/>
<input type="text" value="THYMUS"/> <input type="checkbox"/>	<input type="text" value="THYROID"/> <input type="checkbox"/>	<input type="text" value="TRACHEA"/> <input type="checkbox"/>
<input type="text" value="ESOPHAGUS"/> <input type="checkbox"/>	<input type="text" value="LUNG"/> <input type="checkbox"/>	<input type="text" value="RIGHT CRANIAL"/> <input type="checkbox"/>
<input type="text" value="LOBES"/> <input type="checkbox"/>	<input type="text" value="RIGHT DIAPHRAGMATIC"/> <input type="checkbox"/>	<input type="text" value="HEART"/> <input type="checkbox"/>
<input type="text" value="EYES"/> <input type="checkbox"/>	<input type="text" value="BRAIN"/> <input type="checkbox"/>	<input type="text" value="CEREBRUM"/> <input type="checkbox"/>
<input type="text" value="CEREBELLUM"/> <input type="checkbox"/>	<input type="text" value="MIDBRAIN"/> <input type="checkbox"/>	<input type="text" value="BRAIN STEM"/> <input type="checkbox"/>

Fig. 18

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

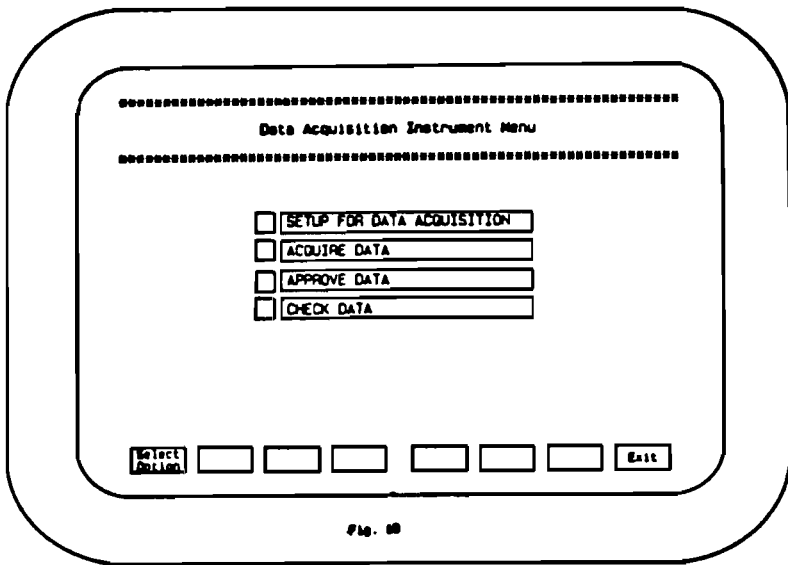


Fig. 28

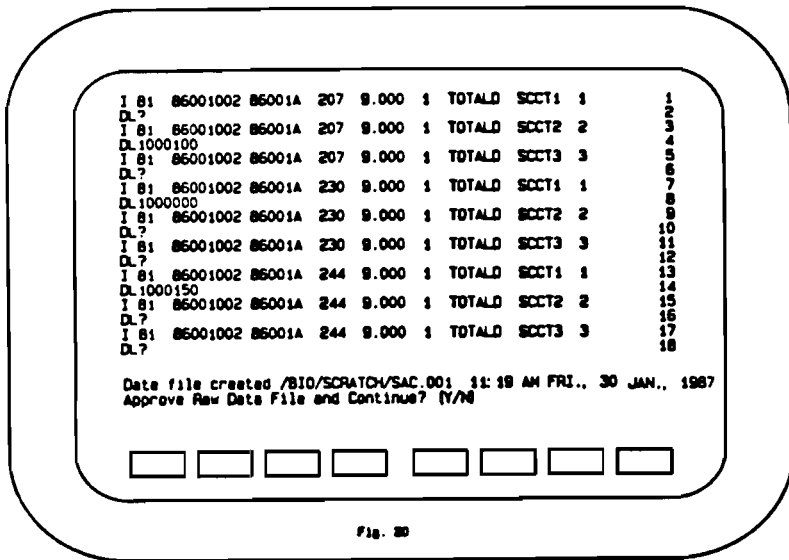
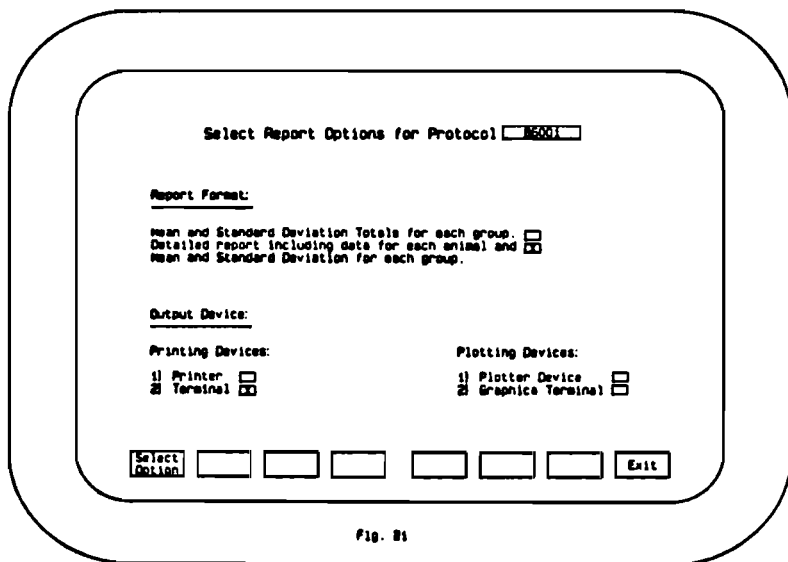


Fig. 29

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES



FINAL REPORT Page 1
 Date: MON., 30 MAR., 1987 Time: 10:57 AM

Report/Protocol # TRD-STS-86-999
 PHARMACOKINETICS OF NICOTINE, INTRAVENOUS
 Amount of Total Radioactivity
 DL-(pyrrolidine-2-14C) nicotine
 Sample: URINE
 Species: RATS Strain: SPRAGUE-DAWLEY
 Dose Expressed as: nanomole

-----TIME(HR)-----	
AN#	48.00

Group:A Dose: .49 micromole/kg Sex:M Route:INTRAVENOUS	
310	63.200
311	112.842
312	126.392
313	94.800
314	103.830
Mean	100.213
StdD	23.795
Group:B Dose: 2.47 micromole/kg Sex:M Route:INTRAVENOUS	
410	210.344
411	328.623
412	353.445
413	454.556
414	395.413
Mean	348.476
StdD	98.768

 * No data available. See notes for study.

Fig. B2
 A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC
 DISPOSITION STUDIES

Report/Protocol # TRD-STS-86-001
 PHARMACOKINETICS OF NICOTINE, INTRAVENOUS
 Amount of Total Radioactivity
 Sample: URINE
 Species: RAT Strain: SPRAGUE-DAWLEY

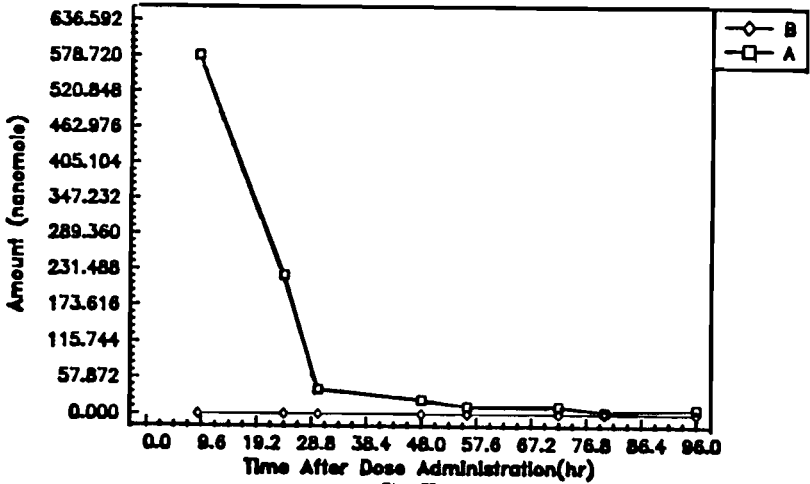


Fig. 8B

PRELIMINARY REPORT Page 2
 Date: MON., 30 MAR., 1987 Time: 10:58 AM

Report/Protocol # TRD-STS-86-777
 PHARMACOKINETICS OF NICOTINE, INTRAVENOUS
 Concentration of Total Radioactivity
 Metabolite Report
 Sample: URINE Strain: SPRAGUE-DAWLEY
 Species: RATS
 Data Expressed as: nanomole

Group	Dose: 4.00 nanomole/kg Body Weight						Total
	1	2	3	4	Parent Recovery	Metab Recovery	
0201	0.480	0.975	0.619	0.497	0.930	0.973	8449.844
0202	27.731	27.860	28.009	26.142	27.751	27.899	8644.000
0203	110.919	119.049	126.463	120.970	110.434	119.949	8900.011
Mean	91.713	92.130	92.361	92.993	91.840	92.130	7963.322
StdD	90.992	99.120	99.770	60.020	99.270	90.301	994.300

Metabolites:
 1. COTININE-N-OXIDE
 2. 3-PYRIDYLACETIC ACID
 3. COTININE
 4. NICOTINE-1'-N-OXIDE
 Parent compound:
 114C-2'-PYRROLIDINE NICOTINE

Fig. 8A

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

Report/Protocol # TRD-ST5-86-999
 PHARMACOKINETICS OF NICOTINE, INTRAVENOUS
 Material Balance of Total Radioactivity
 DL-[pyrrolidine-2-14C] nicotine
 Species: RATS Strain: SPRAGUE-DAWLEY

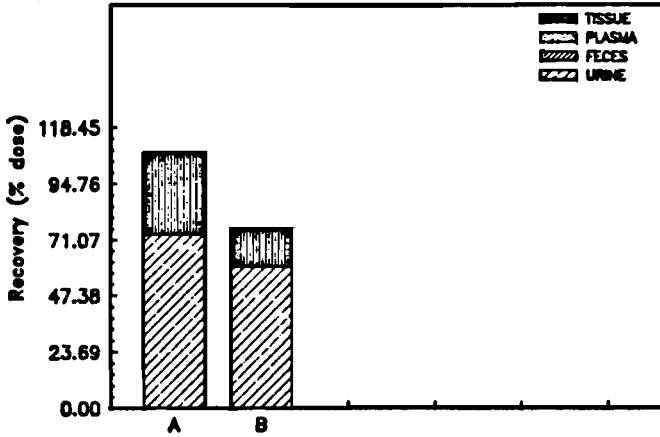


Fig. 8

FINAL REPORT Date: FRI., 27 MAR., 1987 Time: mid PM Page 1

Report/Protocol # TRD-ST5-86-999
 PHARMACOKINETICS OF NICOTINE, INTRAVENOUS
 Dose Administration:
 DL-[pyrrolidine-2-14C] nicotine
 Species: RATS Strain: SPRAGUE-DAWLEY

		Body Weight (g)		Dose		Sacrifice		Dose Administered	
NO	INIT	DOSE	FINAL	DATE	TIME	DATE	TIME	PL	DPM
Group:		Dose:		.49 micro mole/kg		Sex:R		Route:INTRAVENOUS	
310	189.0	281.0	192.0	06/05/86	19:47	06/07/86	19:47	.281	.220E+07
311	176.0	311.0	189.0	06/05/86	19:48	06/07/86	19:48	.311	.338E+07
312	172.0	302.0	184.0	06/05/86	19:50	06/07/86	19:50	.302	.378E+07
313	198.0	256.0	202.0	06/05/86	16:06	06/07/86	16:06	.296	.2E02E+07
314	191.0	302.0	203.0	06/05/86	16:07	06/07/86	16:07	.302	.378E+07
Group:		Dose:		2.47 micro mole/kg		Sex:R		Route:INTRAVENOUS	
410	176.8	245.0	186.0	06/05/86	16:13	06/06/86	16:13	.245	.1340E+08
411	181.0	238.0	195.0	06/05/86	16:16	06/06/86	16:16	.238	.1302E+08
412	185.0	232.0	197.0	06/05/86	16:17	06/06/86	16:17	.232	.1269E+08
413	187.0	256.0	199.0	06/05/86	16:18	06/06/86	16:18	.256	.1408E+08
414	168.0	219.0	188.0	06/05/86	16:19	06/06/86	16:19	.219	.1198E+08

Init body weight = weight at start of acclimation
 Dose body weight = weight at dosing
 Final body weight = weight at sacrifice

Fig. 8

A REAL-TIME MINI-COMPUTER SYSTEM FOR AUTOMATING RADIOLABELED XENOBIOTIC DISPOSITION STUDIES

Operator Menu

1. Copy data from database to files
2. Copy data from files to database
3. Delete data from database

Choose option (/, 1, <cr> to quit) >

Fig. 27

A Word Processing System For the HP1000
F. Stephen Gauss
U.S. Naval Observatory
Washington, D.C. 20392

Introduction

Word processing on the HP1000? On the surface, perhaps, an interesting idea of little consequence in these days of PC word processing. However, for all the advantages that the PC's offer, the environment in which the HP1000 is usually found may contain all of the necessary components of a word processing system and offer some significant advantages over the single-user system. One important factor is the fact that the text is entered using the same editor that is already familiar to the user from his programming environment. Although EDIT/1000 is not intended as a text processor, it lacks only a few of the desirable features (such as automatic carriage return at right margin) for such use. Additionally, the data to be included in a scientific paper are often already in the HP1000 system, making it very easy to print text, tables, figures and graphs all in the same environment. A major problem with PC's is the sharing of resources. On a multi-user system not only can the authors all work on the same paper, but they can do it from their own terminals. In addition there is usually a choice of printers, plotters, etc. depending on the speed and quality desired in the output.

The programs described here - TYPER, SPELR, and FONTEDIT - can be (and have been) used for everything from production of printers copy of conference proceedings (the 1984 and 1985 INTEREX conferences) to memos to viewgraphs. They provide for managing the LaserJet+ soffonts, checking the spelling of the document, and formatting and printing the text on a variety of printers.

One of the biggest advantages of such a set of programs is that they are all written mostly in FORTRAN and the source code is maintainable by the user. In addition, in many cases modules can be custom tailored to individual applications. The drawback, of course, is that once they are modified by the user, he can no longer expect debugging help from the author and each new update must also be modified.

I. TYPER

Background

TYPER is a document formatter and word processor for the HP1000. It began life as TYPO by Alan Whitney, was modified by Don Wright as TYPE, and in its current form is known as TYPER. It is described in the paper, presented at the first conference of the HP1000 IUG (San Jose I), entitled TYPO - A Word Processing Program¹. The presentation here will describe changes made in the last few years, especially with respect to the LaserJet, with some comments on the methods used to implement the features. A complete revision history is given in the TYPER manual that accompanies the program.

The initial impetus for modifying TYPE came from the introduction of small dot matrix (Epson-like) printers to the systems used at the US Naval Observatory. Although their quality is inferior to the daisywheel printers that TYPO was designed for, the dot matrix printers had certain features that were very desirable for technical papers, such as high speed draft mode printing along

1. Minicomputer Research and Applications, ed. Helen K. Brown, 1981

```

****      2686 LASERJET PRINTER DRIVER TABLE
****      FONTS for the 92286J Math Elite Cartridge
****      The following commands depend on the system,
****      rather than the printer
DRIVER    000000 000000 000000
ZTRANSP  002000
****
DOWNQ     000002 015475 000000 000000
UPQ       000010 015475 015446 060455 030522 000000
VERTQ     002042 000006 015446 066000 000103
NEGLFQ    000006 015446 060455 030522
FFQ       000002 006015
ABS TAB   002042 000006 015446 060400 000103
LMARGQ    002042 000006 015446 060400 000114
ZHMI      006343 000012 015446 065400 000000 000000 000110
ZHCURSOR  006143 000012 015446 060400 000000 000000 000110
ZBOLD     000000 000005 015450 071464 041033 024163 030102
****      Prestige Elite USASCII
CSEQ B    000020 015446 066060 047433 024060 052433 024163 030160 030462
CSEQ B    000012 064061 030166 030163 030142 034124
****
>SCRATCH SCRATCH
>TP MARG  0
>BT MARG  0
>HEIGHT   100
>LU       26
END

```

Figure 1. - Partial Printer Driver Table For LaserJet 92286J Cartridge

```

****      DIABLO PRINTER TABLE 630/620
DRIVER
ZADJUST   000001
DOWNQ     000002 015525 000000 000000
UPQ       000002 015504 000000 000000
VERTQ     000031 000003 015436 000000
NEGLFQ    000002 015412 000000 000000
LMARGQ    000031 000005 015411 000033 034400
ZHMI      000031 000003 015437 000000
ZBOLD     000000 000002 015527 015446 000000 000000 000000
ZUL       000000 000002 015505 015522 000000
ZLINES    000060
ZRESET    000003 015415 050012
ZTRANSP   000100
ALTNSSET  000000
ALT1SET   000000
ALT2SET   000000
ALT3SET   000000
ZSUPERSC  000005 015436 002033 005000 000000 000000
ZSUBSC    000004 015436 002012 000000 000000 000000
ZCNCLCD   000000 000000 000000 000000 000000 000000
****      DEFAULT MARGIN SETTINGS IN .1 INCHES
>TP MARG  12
>BT MARG  15
>HEIGHT   110
>LU       20
END

```

Figure 2. - Partial Printer Driver Table For Diablo 630/HP2601 Daisywheel

with slower speed near letter quality output, built-in alternate character sets, including math and Greek and international symbols, and italic and bold face type. Upon investigation it appeared not to be difficult to modify the escape sequences that drive the daisywheel to perform the equivalent functions on the dot matrix printers. It is helpful that most of these printers use Epson-compatible escape sequences. By expanding the arrays that contained the escape sequences to multiple dimensions it was possible to select, through the runstring, the type of printer being used for a specific printout. Each vector of the array represented a printer and up to six different printer types could be handled. A potential problem was the amount of memory used by these arrays, since TYPER was already quite large. However, as long as enough space was available, any kind of printer could be supported by including its escape sequences in the array. A printout of a complex mathematical equation within text with only a few TYPER commands demonstrated that the program worked as designed.

Then HP introduced the LaserJet. For a price approaching the cost of a good daisywheel it was possible to do everything that both the dot-matrix and the daisywheel printers could do with better quality than either one. However, in order to implement the various fonts provided on cartridges, it was necessary to treat each cartridge as a separate printer. In addition, the escape sequences required to select a font were much longer than anything encountered up to that point. All the arrays in TYPER were much too small!

Printer Driver Tables

Taking a lesson from the approach used in some PC software, the concept of a printer driver table was introduced. In this approach only one set of arrays is needed. A printer driver table (PDT) is a file containing a list of the functions supported by a specific printer with the escape sequences needed to implement each function. Figure 1 is a sample portion of a printer driver table for one font cartridge for the LaserJet. Figure 2 shows the same functions implemented in a PDT for the Diablo 630 daisywheel. Now all that is necessary is to specify a PDT at run time and TYPER will fill the arrays with the proper escape sequences.

One of the difficulties encountered in creating the printer driver tables was the way in which variable data are coded. For the Diablo and NEC daisywheels certain information is provided as a binary value; for example the vertical spacing is given in 48ths of an inch as a binary 8-bit number. For the HP escape sequences, the same information is coded as an ASCII character string embedded in the escape sequence. In the former case the escape sequence is always a fixed length. In the HP case the escape sequence is delimited by codes, but can vary in length depending on the size of the number to be represented in ASCII. For the LaserJet many of the sequences can have fractional values, thus adding to the complexity of the string. To accommodate these different formats a code was developed as shown in figure 3, which is used to communicate the form of the expected number.

First word is octal 00KLMN where

N=1 Value is to be inserted as a binary word (ie, as is)

2 Value is to be inserted as an ASCII integer character string

3 Value is to be inserted as an ASCII decimal character string

M= byte number to begin filling in value starting with the third word of the array as bytes 1 and 2

L= the number of digits following the decimal point, if N=3

K= the number of digits total, including decimal point if N=3

Thus, 5233B would put Value= 12.5 into the array such that it would be
5233B 8 esc& 12 .5 OK

Figure 3. - Control Word Coding For Printer Driver Table Array

In most cases a variable in the printer driver table is defined by a group of octal numbers; the first is the byte count and the remainder are the actual escape sequence data. When a control word is required, it precedes the byte count (see ZHMI, for example, in figure 1). When dealing with various types of printer, accomodation must be made for unsupported features. This is handled by setting the byte count to zero. In cases where the feature can be emulated, TYPER will do so. Thus, if the printer does not support a boldface mode (ZBOLD byte count = 0), TYPER will attempt to print the character, then backspace, increment by one HMI unit, and print again. If the printer does not support HMI (ZHMI byte count = 0), that is micro-spacing in small (usually 1/120th of an inch) units, then the character is simply printed twice with a backspace in between. Some operations are obviously impossible - line printers usually do not support negative line feeds, for example, and the italics command is meaningless on the daisywheels. In this case it is up to the user to either avoid such commands or interpret the resulting printout accordingly.

Page Layout

Most of the printers used for document printing provide for controlling the page size. This involves setting the left, right, top and bottom margins with respect to the paper and some method of determining what to do when the bottom of the page is encountered. In the case of the daisywheel and dot-matrix printers, the paper can be physically positioned horizontally and vertically. The left edge of the paper should be aligned with the "0" column mark on the printer. The top edge usually must be positioned above the paper guide and so that printing will begin about an inch from the top. TYPER assumes a 1" top margin and will begin printing without moving the paper vertically. When it reaches the bottom margin as defined by the \HZ command, its action depends on the mode of operation. In batch mode it will space the paper such that it will then continue printing 1" from the top of the next page. Batch mode assumes continuous form paper. In interactive mode single sheet paper is assumed and TYPER stops after printing the last line. The user can then eject the page with the E command or print one (or more) additional line. In any case it is the user's responsibility to feed a new sheet of paper and align the margins before giving the G (go) command. The default margins for these printers are:

HZ	left marg	right marg	paper width
	1.0"	1.0"	8.5"
VT	top marg	bottom marg	paper height
	1.0"	1.2"	11.0"

The LaserJet is somewhat more complicated to control, but the principle is the same. The main differences are that it is smart enough to know all about the paper in use and that the physical and logical page sizes are not the same. Therefore, it is more critical to avoid giving commands that conflict with the printer's own settings. The LaserJet has a 1/4" dead zone at the top and bottom of the page where no printing can appear. There is also a 1/4" dead zone at the left and right edges, but this is not documented as well in the manuals. Therefore, the logical page is only 8 x 10 1/2 inches. The printer can also determine the actual size of the paper in use. The LaserJet manual suggests avoiding the top and bottom 1/2" and the printer defaults to a 1/2" top and bottom margin. These margins can be changed (for example in the RESET command of the PDT), but setting the text and page length requires first setting the number of lines per inch, since the text and page lengths are given in lines. In addition the LaserJet has the annoying habit of ejecting the page automatically as soon as printing enters the bottom margin, even if the incursion is by only 1/300th of an inch. TYPER's footer command, for example, allows printing in the bottom margin, so this can become a real problem. The solution to all of these difficulties was to allow the printer to control the margins, but let TYPER control the page ejection. This is done by turning off the perforation-skip mode in the

PDT RESET command. Since printing will start at the top margin it was decided to let TYPER think that the page is only 10 inches long with no margins. This prevents any printing in the dead zones.

A careful reading of the manual will reveal that the left margin is set by column number and that column 0 starts 1/4 inch in from the left hand edge of the page. Thus, one would specify a left margin of .75" to produce a real 1" margin. This becomes more confusing when margins are frequently reset. To avoid the problem, the PDT command >LOG LFT n is provided. Here, n is the distance from the physical left edge to the logical left edge of the paper. The margin commands are always given with respect to the physical left edge, so for the LaserJet the value of n is 3 (tenths of an inch). Thus, the command \HZ10 will set the left margin at .7" from the logical left hand edge or 1" from the physical left edge, making it transparent to the user.

The defaults (set with the >TP MARG, >BT MARG and >HEIGHT commands in the PDT) are:

HZ	left marg	right marg	paper width
	1.0"	1.0"	8.5"
VT	top marg	bottom marg	paper height
	0.0"	0.0"	10.0"

The important thing to remember is that even if the full page size were specified, none of the TYPER commands will reset the physical top margin of 1/2". That would have to be done through the RESET parameter in the PDT. None of the TYPER commands can access the left 1/4" of the page; this area can only be accessed by moving the cursor in decipoints.

An important advantage of the LaserJet is the ability to print in both portrait and landscape mode. Landscape mode is especially useful for viewgraphs, although TYPER also has the ability to print two logical pages side by side in compressed type on a physical page. This is described in the TYPER manual. It is not intended for printing exact copies of a full page at reduced size, but rather for a quick printout of a lengthy document in a small amount of space.

Printing in landscape mode is easy if three points are kept in mind:

- a. changing orientation forces a page eject. Thus, the two modes cannot be mixed on a single page. Furthermore, there will be a blank sheet ejected before the first page, since the default is usually the portrait mode.
- b. a landscape font must be selected. Landscape fonts are not just portrait fonts rotated in the printer. They are separate font sets which must be loaded into the LaserJet. Selecting a landscape font as the primary font automatically puts the LaserJet into landscape mode and ejects any partially printed page.
- c. the page parameters must be reset. Both the \HZ and the \VT commands must be issued either just before or just after the font is selected. Usually the following commands will do the job:

```
\HZ10,10,110,8  
\vt0,0,75,3
```

For viewgraphs, a 14 or 16- point font is convenient. The vertical line spacing should be reset along with the pitch, as shown in the example. Remember that with the above parameters the page will be restricted to 72 characters horizontally and 22 lines vertically. This is usually plenty for an overhead slide.

Right Margin Alignment

One of TYPO's main claims to fame was the ability to produce right justified output with proper spacing between the words. A popular expedient is to simply insert blank spaces between words in order to make the right margin line up. The method used in TYPO was to take advantage of the

horizontal micro-spacing and to vary the pitch in such a way that the correct alignment was produced. Thus, part of the line might be printed at 12 characters per inch and the remainder at 13 cpi. The pitch change is virtually unnoticeable, except when short lines with a lot of long words are involved. That case, however, appears even worse using the extra spaces method. The LaserJet allows the same method to be used as on the daisywheels, thus producing very nicely aligned copy. However, an even better degree of alignment is available through the use of fractional pitches. The LaserJet can print the fonts at any pitch with a resolution of 1/120th of an inch. The coding method of figure 3 allows either of the two methods to be used. The PDT's for the LaserJet are normally set to use fractional spacing. This is required when proportional fonts are used.

Control Sequence Command

The PDT concept allowed some commands to be re-defined depending on the printer used. The commands {, }, \{, and \} were originally intended to allow the use of different printwheels on the daisywheel and a different thimble on the spinwriter. These functions are still defined in the PDT's for those devices, but, for the LaserJet, they allow access to additional fonts and, on the dot matrix printers, to additional character sets. On the LaserJet it was convenient to use the first pair of symbols to shift in and out of the upper 128 bits of the 8-bit character sets and to use the second pair to select the primary or secondary character set.

A special command had to be implemented to select a font on the LaserJet, but it, too, was done in a general way. The command is called a control sequence (\!x). There can be 26 different control sequences, one for each alphabetic character (\!A through \!Z, case insensitive). Normally, they are defined as the escape sequence necessary to select one of the fonts in the LaserJet, as shown in figure 4. However, since they are really just a string of characters that is sent to the LaserJet, they can be used to force any desired action by any printer. One caution should be noted, however. To force the selected font to be the secondary character set, the command \!x is given. This actually modifies the control sequence from a primary sequence to a secondary sequence by changing all occurrences of (to). Thus, using this command for a different purpose could cause undesired results.

```
****          esc& ll      Oesc  (8  Uesc  (s  Op  10  h1
CSEQ B  000022 015446 066061 047433 024070 052433 024163 030160 030460 064061
****          2v      Os      Ob      3T
CSEQ B  000010 031166 030163 030142 031524
```

Figure 4. - Sample Control Sequence (\!B) From PDT CARTD

Runstring Parameters

A common requirement is for a single printing to be made with slightly modified conditions. Another requirement is for certain conditions always to be in effect regardless of whether the user asked for them or not. TYPER allows a single command to be input in the runstring. Since TYPER commands are defined with a \ character, it is necessary to enclose such commands in the runstring in back quotes (`) to avoid incorrect parsing. The most useful such command is the \FI command, which allows a file of additional commands to be introduced through the runstring ahead of any commands in the actual document file. Figure 5 is an example of such a use to set up for math and Greek character printing at 12 pitch.

The one command that cannot be introduced this way is the \VT command to set the top margin on the page. To set the top margin of the first page this command must be given before any output is sent to the printer. Even though it may not print anything, any command other than a type 1 command will, in fact, cause something to be sent to the printer. In particular the control sequence command, which often must be given before anything is printed, is not a type 1 command even though it usually appears in column 1.

File MATH_FONT contains

```
\HZ, , , 12
\!A
\!!E
```

Then RUN, TYPER, TEXT, CARTJ, '\FIMATH_FONT'

Figure 5. - Example of the use of a file to set default commands

The LaserJet can make multiple copies of a document. The COPIES= command in the runstring allows up to 99 copies of each page to be produced automatically.

Other runstring commands can be found in the TYPER manual.

The "D" Mux And Handshaking

A frequently asked question these days is "Will it work at release 4.1?" In other words has it been modified to run with the new multiplexor. Although it has not, as of this writing, been tested with the new multiplexor, it is hoped that this will be done before the time of the conference. In any case the answer should be "No problem!" In the PDT is a variable called ZTRANSP representing the function bits for the output control word of the EXEC call. In fact all of TYPER's output is through a single XLUEX call whose function bits are set from the ZTRANSP word of the PDT. Thus, it should be possible to set that variable to whatever is necessary to make it work with any particular set of hardware. In addition a number of handshaking options are provided. The simplest is DRIVER; that is, the handshaking will be handled by RTE and TYPER need not worry about it. Some printers do not support ENQ/ACK (such as the Epson-like printers) and some hardware, such as the BACI card, do not support XON-XOFF. Additional choices allowed by TYPER are ETX/ACK (which is supported by some dot matrix printers) and DELAY. ETX/ACK is supported directly by TYPER generating the ETX. The DELAY mode allows TYPER to suspend for a known interval after each line is sent. Note that normally TYPER should not operate through a printer driver, but rather through a port configured as a terminal. TYPER wants to handle all page spacing itself.

Printer Implementations

Implementing the various printers was both a chore and an education. When the HP2934 was announced with an HP2601 emulation mode, it appeared to be a good bet for use with TYPER. There is a PDT for this device (WP2934) and it does, in fact, work. However, it is unlikely that anyone would be very happy with it. The reason is in the design of the 2934 and in what is meant by "emulation". The printer is designed to print at a fixed pitch. The pitch is built into the specific type font being used. To attain its rated speed the motors driving the print head must run in a continuous motion. For a single character it appears that the print head must move for several inches at full speed. If the required pitch is not the same as the font pitch, then every character must be printed separately and each involves a complicated back and forth motion of several inches. The net effect is that the time necessary to print even a single line is inordinate.

The 2934 can be operated in 2934/2631 mode, however, and produce good results, although by no means letter quality. This PDT is PR2934.

It is sometimes useful to be able to create a file containing the results of passing the document through TYPER, if only to get nicely formatted paragraphs. The PDT TAPE8 allows the output to go to a tape drive.

One printer that has not been implemented is the Okidata. Its escape sequences are even more difficult to work with than those of HP. However, there is no reason that such a PDT could not be

written.

Finally, there is even a PDT for a terminal. As usual, TYPER will attempt to perform as many functions as possible. Margins can be set, boldface will appear as inverse video, etc. In this mode the program will pause at the end of each page, otherwise the text would be lost off the screen.

Perhaps the most interesting test of the ability of the PDT to do the job was an experiment in which an Atari 800 with terminal emulator was used to dial in to the HP1000. The terminal emulator was set to upload to the Atari's printer (a Gemini 10X) and the PDT for an Epson-like printer was set with the LU of the communication line. Thus, TYPER sent Epson-like control information to the Atari which passed it on to the printer. The printer produced perfectly aligned copy as though it were connected directly to the HP. This can be done with an HP terminal with a printer connected, but the terminal will see and respond to the escape sequences as they pass through. Therefore, it is necessary to turn on DISPLAY FUNCTIONS. The PDT should define the printer, but the LU should be set to that of the terminal. The system LU should be used to prevent TYPER from pausing at the end of each page, thinking that it is going to a terminal.

The Run Command

Another useful command that has been added to TYPER is the \RU command. This allows a program to be run from within TYPER. One could, for example, produce graphics on the same page with text by running a program to output the graphics to the printer. The \RU command can also be used to insert the output data from a program directly onto the page. In such cases, where the \RU command is used to place data on the printed page, TYPER will no longer know where the cursor is. Such a command should usually be followed by a \BP command. As will be shown later, it is often desirable to load softfonts into the LaserJet with each particular printout. The command

\ru,fontedit,tr,loadfont.list

will ensure that the fonts listed in the command file are installed (see the FONTEDIT transfer file section).

The Table Of Contents Command

Some word processors provide the capability for index and table of contents generation. The \:x command allows a table of contents to be created, although it must follow the text. Each value of x represents the storage of a page number. The first time a particular \:x command is encountered, the current page is saved. From then on, whenever that particular \:x command is encountered, the page number that was saved the first time is output to the page. Up to 26 different page numbers can be saved. The TYPER manual Table of Contents serves as an example of the use of this command.

Equations And Equation Mode

TYPER is quite suitable for printing scientific and engineering text containing mathematical symbols. An example is shown in figure 6.

$$\sigma_{\epsilon} = (\epsilon/T) [(n-1)/n]^{1/2}, \text{ where}$$

$$T = (2\pi/n)^{1/2} / B[(n-1)/2, 1/2] = (2\pi/n)^{1/2} \Gamma(n/2) / (\Gamma[(n-1)/2] \Gamma[1/2])$$

$$\text{and } B(m,n) = \int_0^1 x^{(m-1)} (1-x)^{(n-1)} dx, \text{ is the Beta function}$$

Figure 6. - A Sample Mathematical Equation

```

\!a
\!!g
\hz,,13
\ce{\(s\)\)\(e)\ [= (\(e\)/T) [(n-1)/n]\[\(1\)\[\(2\)\]], where

T = (2\(\p\)/n)\[\(1\)\[\(2\)\]]/B [(n-1)/2,1/2] =
      (2\(\p\)/n)\(1\)\[\(2\)\(+)\(n/2)\{\(+)\(n-1)/2\}\(+)\[1/2\}

      \(\)\[\(1\)\]
and B(m,n)= \(") X\[\[\(m-1)\]\]\(1-x)\[\[\(n-1)\]\]\]dx, is the Beta function
      \[\(=\)\]\]\(0)\[\[

```

Figure 7 - Standard Commands Used To Create Figure 6

```

\!a
\!!g
\hz,,13
\eq1
\ce^#s#_#e#^ = (#e#/T) [(n-1)/n]^#1#^/_#2#_, where

T = (2#p#/#n)^#1#^/_#2#_/B [(n-1)/2,1/2] = (2#p#/#n)^#1#^/_#2##+#(n/2)/{#+#[(n-1)/2]#+#[1/2]}

      #'^#^#1#_
and B(m,n)= #'# X^^^(m-1)_ _ (1-x)^^^(n-1)_ _ dx, is the Beta function
      ^#=#_ _ #0#^

```

Figure 8 - Equation Mode Commands Used To Produce Figure 6.

The commands needed to produce figure 6 are shown in figure 7. Since each TYPED command consists of at least two characters (a backslash followed by one or more additional characters), a complex equation can easily exceed the 80 column screen width or even the 150 column EDIT/1000 limit. The equation mode (`\EQ`) was implemented to make such equations easier to enter and to read. Single characters are used to substitute for several of the most common TYPED commands. For example, while equation mode is in effect, a superscript can be obtained with a caret (^) as well as with the `\[` command. Thus the equation of figure 6 can also be written as shown in figure 8.

Font Management On the LaserJet

TYPER is most impressive when used with the LaserJet Plus or LaserJet Series II. With multiple soft fonts loaded into the printer it is easy to produce impressive looking documents. The secret, of course, is to be able to select the desired font easily.

Each font must be stored in the LaserJet in some way; it can be an internal font residing permanently in the LaserJet's memory, a font residing on a cartridge inserted into the LaserJet, or a font described by a file residing in a computer, which can be copied to the memory of the LaserJet. The latter fonts (soft fonts) can be designated as either temporary or permanent when they are loaded. Permanent fonts are deleted only when power is turned off or with a special delete escape sequence (see FONTEDIT). Temporary fonts are also deleted when a RESET is performed. Since TYPER resets the printer before each document (although this can be changed with a PDT entry), soft fonts should usually be loaded as permanent.

Once a font is loaded into the printer, it must be selected as the active font. This can be done by specifying the characteristics of the font or by giving its ID number. The characteristics are arranged by priority, as shown in figure 9. To select a font it is necessary to specify enough characteristics at a high enough priority to unambiguously select the font desired. For example, specifying landscape mode and 9 point may or may not select the desired font. You will definitely get a landscape font, since there is always at least one available - the internal font. However, the characteristics of the previously selected font are still in effect. If you have not given enough specification to override them, they will help to determine which font is activated. Since the pitch is a higher priority than the point size, it will take precedence. If the previously selected font had a pitch of 12, then the landscape font with the pitch nearest to 12 will be selected, regardless of the point size. In this case it would probably be the 10 pitch internal Courier font. This is not as complicated as it seems, but it is easy to make a mistake. The best approach is to fully specify the font that you want.

```
Orientation: PORTRAIT   LANDSCAPE

Symbol set:  ROMAN-8    MATH SYMBOLS   MATH8A   MATH7    ANSI-8
              KANA-8    LINE DRAW      MATH8B   USASCII  ROMAN   EXTENSION
              MATH-8    US LEGAL      PIFONT   PIFONTA

Spacing:     PROPORTIONAL  FIXED

Pitch:       nnPITCH  where nn is the desired pitch

Point size:  nnPOINT  where nn is the desired point size

Style:       UPRIGHT   ITALICS

Stroke:      BOLD     MEDIUM   LIGHT

Typeface:    LINE PRINTER  ELITE    HELV     PRESTIGE  COURIER  ORATOR
              PICA        GOTHIC   TMS RMN  CASLON    SCRIPT
```

Figure 9. - Keywords and priority sequence for LaserJet font characteristics

Another area of common confusion has to do with boldface and italics. As mentioned before, every font must reside in the printer. Boldface fonts are not just the regular font printed darker - they are actually different fonts. Therefore, specifying boldface will only work **correctly** if there is a boldface equivalent to the font in use loaded into the printer. Otherwise, a boldface font with the wrong characteristics will be selected. Thus, in the demonstration files provided with TYPER, boldface is produced by double printing, since there is no internal boldface font. Once the fonts are available in the printer, TYPER can select among them.

To allow LaserJet fonts to be selected easily and unambiguously the control sequence command (\!x) has been designed with considerable flexibility. It allows three methods of selecting a font. In each case the particular font is assigned a letter code by the control sequence command. Until it is redefined, all that is necessary to select that particular font is to specify its corresponding letter code. The letter code is case insensitive, so up to 26 fonts can be defined at one time. Any letter code can be redefined at any time. The actual fonts, of course, must be available to the LaserJet. The following describes the three methods of font selection and the extensions used for proportional fonts.

A. Selecting a cartridge font

The LaserJet contains a number of internal fonts depending on the LaserJet model. In addition one or two font cartridges can be plugged in, each cartridge containing several fonts. Each cartridge has a part number ending in a letter and it is this letter that is used to designate the cartridge. Thus, cartridge 92286J is the "J" cartridge containing 10 point math and Greek characters along with a standard Prestige Elite character set. Printer Driver Tables can be written for each cartridge and several are supplied with TYPER. The PDT's are named CARTx where x is the cartridge letter. Thus, CARTJ is the PDT for the 92286J cartridge. Each PDT for a cartridge contains control sequences for the fonts contained on the cartridge. Each control sequence is defined in the PDT as the escape sequence necessary to completely select the particular font. It is not necessary to completely define a boldface or an italic font, since TYPER will select those by requesting the bold or italic versions of the currently enabled regular font.

Since the cartridge PDT has predefined certain control sequences, it is only necessary to give the command to specify the particular pre-defined control sequence. Thus, the command \!B will enable the font whose escape sequence is stored in the B control sequence of the PDT. In most cases the B control sequence is set up to select the most "normal" font on the cartridge and the Y and Z fonts select the two internal 12 point fonts. The LaserJet allows fonts to be designated as primary and secondary and TYPER allows a font to be selected as the secondary font with the secondary control sequence (\!x). This is useful when alternating between two fonts. The control sequence commands can be used to switch between two fonts, but each time the command is given, the complete escape sequence needed to select the font is sent to the printer. This makes for considerable overhead and, in the worst case (each letter in a different font), can overflow the TYPER line buffer. Designating one of the fonts as a secondary font and switching between them with the alternate character set command (\(, \)) results in far less overhead since these commands each represent a single character (SI/SO).

As an example, the following selects font set B, prints some text, selects font set G, prints some text, and then prints a line with a mixture of the fonts.

characters

```

26      \!b
32      This is printed with the B font.
26      \!g
32      This is printed with the G font
26      \!b
162     This \!gis \!b printed \!gwith \!balternating \!gfonts.
```

The following does the same thing, but much more economically.

```
26      \!g
26      \!b
32      This is printed with the B font.
34      \/(This is printed with the G font\)
38      This \is \ printed \with \)alternating \font.\)
```

To summarize, if you are using a cartridge font and there is a PDT for that cartridge, the fonts can be selected by giving the control sequence command with the appropriate font letter.

B. Selecting by characteristic

In addition to the cartridges the LaserJet Plus and LaserJet Series II contain varying amounts of memory into which soft fonts can be loaded. TYPER assumes that the desired font is already loaded into the printer. (The FONTEDIT section of this paper describes how to load the soft fonts). If the soft font desired matches the characteristics of one of the fonts described in a PDT for a cartridge, then the font can be selected as described above. However, in most cases either there is no pre-defined control sequence for the font or, if there is one, it is not easily accessible. In this case a font description can be established by naming the characteristics of the font desired as part of the control sequence command. Remember that the success of this method depends on **completely** specifying the characteristics and ensuring that the terms used conform to those expected by TYPER. The terms used are the same as in the LaserJet manual and are shown in figure 9. For example, MATH-8 selects the 8-bit math character set, whereas MATH 8 would not be recognized. Spaces and case are not significant. The following are some examples of selecting different fonts.

```
\!l= 12pitch,portrait,10point,math-8a,prestige elite
\b= landscape,fixed,16pitch,lineprinter,usascii
```

Note that this format will replace any pre-defined font occupying the same control sequence letter. Whenever that letter is referred to it will select the new font, not the pre-defined one.

The most likely source of difficulty here is either a mis-spelling or an incomplete font description. This can cause either no change in the selected font or the selection of a font totally unrelated to the one desired. For example, if the second line above had been mis-spelled as

```
\!b= landscape,fixed,16pitcj,lineprinter,usascii
```

then the default 10 pitch courier font would be selected, since that is the closest font in landscape mode to the previously selected 12 pitch. Everything below the pitch characteristic in figure 9 would be ignored.

C. Selecting by ID number

The easiest and surest method of selecting the font is by a pre- assigned ID number. These numbers are assigned at the time the soft font is loaded into the printer using FONTEDIT. Note that the ID numbers printed by the Series II TEST function are **not** the true ID numbers and cannot be used to select the font. Once an ID number has been assigned, then the control sequence command can be used to select that font. The format is

```
\!x=[id]
```

For example, `\!L=[192]` causes control sequence L to select the font identified as number 192. Once this command has been given, the L control sequence will select that font each time it is used, unless a new ID number is assigned to the same letter.

Spacing

Once the font is selected it is necessary to set the spacing required by the particular font. The following discussion of the horizontal and vertical spacing leads into the subject of proportional spacing.

A. Horizontal spacing

All of the fonts discussed so far have been fixed-spacing fonts. That is, for a given font the horizontal size of each character is the same and is described by the pitch. The size is the inverse of the pitch in inches. Therefore, in a 12-pitch font each character is 1/12th of an inch wide. The pitch parameter of the `\hz` command tells TYPER the spacing to be used for the current font. Note, however, that TYPER can only change the spacing between characters, not the actual size of the character. For the LaserJet and the daisywheels, then, the character pitch (width) is determined by the font or wheel installed, but the spacing of the characters can be set by TYPER. A 12-pitch font printed at 10 pitch (that is, 10 characters per inch) will appear more open than usual. The same font printed at 16 pitch will have the characters crowded very closely together. Therefore, anytime the pitch of the font or printwheel changes, the pitch parameter of the `\hz` command should also be changed. A heading printed with a 30 point font (very large), for example, should be printed at around 6 pitch. A compressed font, such as 9 point, should be printed at 16 pitch. However, varying the pitch by 1 or 2 characters per inch is usually not very noticeable and allows more text to be squeezed onto a page. In the case of the dot-matrix printers the pitch and the horizontal spacing are bound together by the design of the printer. The PDT's have been set up to select the built-in font with the spacing closest to the pitch called for by the `\hz` command. Thus, for the LaserJet and daisywheels, specifying a pitch of 13 with a 12 pitch font or print wheel will produce characters 1/12" wide spaced 1/13" apart. On the Epson the 12 pitch font would be selected and the spacing would be 1/12".

The feature of a printer that allows this variable spacing is called the horizontal motion index or HMI and usually has a resolution of 1/120th of an inch. Some printers, such as the LaserJet, allow the pitch to be set to a fraction of an inch. This is the mechanism by which TYPER produces its right margin alignment. The desired pitch is given by the `\hz` command. TYPER determines the number of characters that will fit on a line at that pitch. It then fills the line with text, stopping after the first word that crosses the right margin. It then adjusts the pitch by the amount required to compensate for this overflow. If the printer does not allow a fractional pitch, TYPER prints part of the line at one pitch and part of the line at the next whole pitch higher. The point at which the pitch is changed is determined by the number of overflow characters. In most cases the dual pitch nature of each line is not noticeable. In the case of the LaserJet, where a fractional pitch is allowed, the true pitch required to compensate for the overflow is calculated and the entire line is printed at that pitch. This paper is printed in that way. For printers where there is no variable HMI, there can be no right margin alignment. In this case, the line will overflow the right margin by up to a full word.

B. Vertical spacing

Generally, the default vertical spacing of 6 lines per inch will be satisfactory. For most fonts, especially the 10 point fonts, a spacing of 8 lines per inch can be used to squeeze more on the page with no ill effects. However, it is sometimes desirable to print as much as possible onto a page. In this case the vertical spacing (set with the `\vt` command) can be varied to find the best compromise

between economy of paper and readability. In general the smaller the point size the larger the number of lines per inch can be. However, a close inspection of the fonts will show an interesting phenomenon - the presence or lack of descenders. Some dot-matrix printers, for example, have fonts designed without descenders so that all of the pins in the printhead can be used to improve the resolution of the character. The letter g, for example, extends no lower than the letter o. In this case, a much tighter vertical line spacing can be used. Less common are fonts with unusually high ascenders. An example, is the script font provided with FONTEDIT. Many of the script letters have ascenders when they normally would not. They also tend to have more descenders. In this case even 6 lines per inch looks crowded.

Most dot-matrix printers allow considerable control over the vertical spacing. Spacing is specified in 48ths of an inch per line through the VMI (Vertical Motion Index). The \VT command in TYPER sets this value in terms of lines per inch. The LaserJet, interestingly, when setting lines per inch limits the vertical spacing values to even divisors of 48. Thus, 6, 8, and 12 lines per inch are allowed, but 10 is not. However, as with the dot-matrix printers, the PDT is set up to supply the line spacing in 48ths of an inch as a VMI. Thus, the \VT command will set the line spacing to any integer number of lines per inch. The horizontal and vertical spacing can be used to advantage when printing a very rough draft of a lengthy document. By simply setting the font to the smallest point size available, the horizontal pitch to the one appropriate to that font, and the vertical spacing to 12 or even 16 lines per inch, a fairly readable document can be produced. For example, if the standard font is a 12-pitch font (10 point), a page will contain up to 5400 characters. The same document printed at a pitch of 17 with 12 lines per inch will squeeze 15,300 characters on a page - almost a three to one compression - and still be perfectly readable. To take advantage of this feature the pitch and line spacing should not be changed within the document.

C. Proportional spacing

The highest quality printing is obtained with the proportional spacing fonts. Each character in such a font may be a different size from all the others. The character widths are still fixed, but, for example, a 1 takes up less horizontal space than a W. In addition, the spacing between characters is also fixed. The only variable is the width of the blank character, which is changed by the HMI command. Since each character is different, the only way to align the right margin is to know the widths of every character in the line and the number of blanks in the line. The widths are usually supplied through a proportional spacing table. Many of the dot matrix printers supply such tables in their manuals, although the HMI cannot be varied, but the LaserJet does not. The tables for most of the cartridge fonts were obtained from HP through the cooperation of the Boise division. These tables are contained in the [PROP variable of the PDT. The soft fonts, of course, contain that information inherently in the file. The FIXFONT program creates the table of proportional spacings in a form that TYPER can use (see the FONTEDIT section). These tables must be accessible to TYPER while the line is being assembled, so that it can determine the pitch as described above.

TYPER has to resolve several problems when proportional spacing is required.

It must first obtain the tables. In the case of cartridge fonts, the table is built into the PDT. The [PROP x variable contains the table for the same value of x as the CSEQ x variable. Thus, each font will have both a control sequence array and a proportional table array. Space is at a premium in TYPER and these tables are not suitable for EMA, since they are accessed several times for every character. Therefore, arrays for up to ten proportional tables are provided. They are used in the order that they are required. A 26-word array (ATTPROP) contains pointers for each letter to the corresponding proportional table. When a soft font is required, its table is loaded from a disk file into the next available array. The disk file is the same file that was loaded into the LaserJet to provide the font. The name of the file is supplied to TYPER through the PFILE= parameter of the control sequence command. It is up to the user to ensure that the PFILE named is, in fact, the same as the font requested by the control sequence command. Thus, if the file TR100RPN.USPM was loaded

into the LaserJet (using FONTEDIT) with an ID of 600, then the only command required is

```
\!J=[600],PFILE=TR100RPN.USPM
```

Another problem has to do with boldface and italics. These, of course, are actually different fonts with their own proportional spacing tables. Since they are selected with their own special commands, the control sequence command is not required. However, to provide the necessary linkages, through words 28 and 29 of ATTPROP, a control sequence command is required somewhere with the additional parameters: LINK=ITALIC or LINK=BOLD. For example, the following would completely specify the 10 point Times Roman font with both italics and boldface available.

```
\!K=[602],pfile=tr100ipn.r8pm,link=italic  
\!L=[601],pfile=tr100bpn.r8pm,link=bold  
\!J=[600],pfile=tr100rpn.r8pm
```

Finally, the secondary control sequence must be accounted for. A two-word array, (FONTS_ARE), is maintained containing pointers to the entries in ATTPROP for the primary and secondary fonts. A final variable (PROP_IS) points to the currently active entry in ATTPROP. When an alternate character set command is given, the pointer for the new current font is copied from FONTS_ARE (either the primary or secondary value) into PROP_IS. The value in PROP_IS, then, always points to the proportional spacing table for the currently active font or is 0 for a fixed font.

Remember, two steps are necessary in selecting any font. One is to get the font into the LaserJet and the other is to select that font through TYPER. If, for example, the italic font in the above example was not loaded into the printer, the proportional spacing table will still be set correctly. However, the font cannot be selected. Since italic is not high on the priority list, the chances are that no change will occur; that is, you will not get a fixed font in italics. However, the italic spacing will be used and the line will not be right-aligned properly.

Conclusion

This discussion of the operation of TYPER contains far more information than is necessary to actually do some printing. The TYPER contribution contains a number of sample font files and demonstration documents. However, to try to summarize what has been described here, appendix A shows a sample of a simple letter using a fixed font for the heading and a proportional font for the body of the letter.

For simple text printing, TYPER is unbeatable in its simplicity if you are already familiar with EDIT/1000. For even complicated scientific documents, it is easier to use than many professional word processors. It does not do automatic indexing (yet). It is quite fast, since it is implemented completely in regular memory and the segments are involved only with infrequent operations. For the ultimate in publication quality it is difficult to beat T_EX. However, TYPER is considerably faster, takes less memory and disk space, and will run on most existing HP1000's. The T_EX in the CSL 1000 is restricted to the A-series, although the T_EX from J_KJ Wordware runs on all of the 1000's.

II. FONTEEDIT

The use of the LaserJet with TYPED opened a whole range of possibilities for the creation of nicely printed documents. However, with multiple users accessing the LaserJet, keeping track of which font cartridge was inserted became a real problem. In addition it was discovered that, even with all of the fonts available, there were some commonly-used symbols that were missing (for example the mathematical second derivative symbol - the double dot). The appearance of the soft fonts for the LaserJet Plus, coupled with its increased memory, seemed to offer a solution which could be adapted to the word processing system already in use. Unfortunately, despite several attempts made by various people both inside and outside HP at the 1985 and 1986 INTEREX conferences, the soft fonts were issued only on floppy disks in a form suitable for the PC's, but not for the 1000. Upon investigation it became evident that the soft fonts could be loaded from the 1000, if two obstacles could be overcome.

First, the soft fonts are in PC file form. They had to be put into a form that the 1000 could accept. Secondly, while the fonts can be downloaded directly into the printer as files, it is more convenient if a font ID can be attached to each one. This allows fonts to be selected unambiguously, as well as providing for specific fonts to be deleted.

The first problem was solved by acquiring the soft fonts on micro-floppy disk for the HP150. A 150 was attached to the HP1000 as a terminal (at 9600 baud) and Kermit was used to copy the files from the 150 to the 1000. On the 1000 the Kermit was the version of Paul Schumann's Kermit supplied with the ICT product, CONNECT. On the 150 the Kermit was from CSL/100 disk # 40. Since the Kermits accept wild cards it was a painless, but very lengthy process to copy all of the files to the 1000. Each font is in its own file and there are many fonts per disk with as many as seven disks per soft font set. Transferring all of the files for one soft font set took most of a day!

Because of the second problem, it was necessary to write a short program to load the files into the LaserJet Plus. This program can assign the ID number to a font, designate it as permanent or temporary, delete a single font or delete classes of fonts. Since TYPED usually does a reset before printing a document, it was important to be able to designate the fonts as permanent.

Once the program was written it was a simple matter to download a font file into the printer. It was simple, that is, until the second file attempted overflowed the input buffer in the program. The files copied from the micro-floppy disks turned out to contain all of their data in only two records. Most of the files consisted of 10,000 to 20,000 bytes of data in each of the two records! To get rid of the difficulties imposed by the very large buffers, the program FIXFONT was written. FIXFONT converts a file, obtained as described above from a PC disk, into a type 3 file in which the first record is the font descriptor and each character takes up one record. The input file name can be a mask with wild cards. The output files have the same name as the input files, but with an M (for modified) added to the type extension. Thus, file PR100R12.M8P is the Prestige Elite 10 point 12 pitch regular font of the portrait math-8 character set. The output of FIXFONT would be PR100R12.M8PM.

A word here about the file naming convention would be in order. The names of the files supplied by HP follow the format:

aabbccdd.eef

where aa is the type style, such as TR for Times Roman

bbb is the point size in tenths

c is the style: r=regular, i=italic, b=boldface

dd is the pitch or PN if proportional

ee is the symbol set, such as US for USASCII

f is the orientation: P for portrait and L for landscape

Thus, PR100R12.M8P is Prestige Elite, 10 point, 12 pitch regular in the Math-8 symbol set with portrait orientation

In order for a word processor, such as TYPED, to properly right justify proportional fonts (for Word Processing System For HP1000

true proportional spacing) it must have access to the spacing table for each font.

For the cartridge fonts it was necessary to obtain this table from HP. For the soft fonts this information is available with each character, so it was decided to have FIXFONT build the spacing table whenever a proportional font is encountered. The first record, the font descriptor, has the characteristic that its length is indicated by a byte count, which may be greater than that required to hold the minimum necessary information. Thus, this record usually contains comments having to do with copyright laws and the font style in addition to the font descriptor. The proportional spacing table is automatically added onto the end of this record and consists of 256 bytes of binary data preceded by the characters "PROP=>". Thus, a word processing program can pick up the spacing table by reading the first record of the file containing the font.

Meanwhile, the requirement for the double dot had come up, so the project expanded to permit a limited capability to create new characters. The program used to download the fonts to the LaserJet Plus, thus, became FONTEDIT. It has a number of operating modes with help screens. In its simplest form, it assigns an ID number, sets the font to permanent or temporary, and downloads the file to the printer. These parameters can be entered interactively, through the run string, or in a command file. The ID number is incremented by one with each file loaded until it is changed. The type of the font (permanent or temporary) remains the same until it is changed. Usually, only a certain subset of the fonts will be used in the course of a day. It is convenient to load these all at once when the printer is turned on in the morning. If unique ID numbers are assigned to them, they can always be accessed, regardless of what other fonts might be loaded by other users. This can be accomplished by running FONTEDIT with input from a command file; for example:

The run string for the command file LOADLASER.FONT is

```
RU,FONTEdit,TR,LOADLASER.FONT::FONT_FILES
```

The command file LOADLASER.FONT is:

```
* Fonts for NAVOBSY Memo form          [comment]
L-26                                     [set to system lu 26]
S150                                     [set ID to 150]
P/FONT_FILES/PR100R12.USPM              [upload file as permanent]
P/FONT_FILES/PS240B05.USPM              [ID is 151]
P/FONT_FILES/LG120B12.USPM              [ID is 152]
S10                                      [reset ID to 10]
T/FONT_FILES/PR100B12.USPM              [upload as temporary]
T/FONT_FILES/PR100I12.USPM              [ID is 11]
T/FONT_FILES/PR100R12.M8PM              [ID is 12]
```

FONTEdit displays the amount of memory being used for each file that is loaded, but it has no way of knowing when memory is exceeded. All displays can, optionally, be suppressed in non-interactive mode.

With each character in its own record it became feasible to display the bit pattern on the screen and allow it to be modified. It appeared unrealistic to expect that the average user would want to create an entire font from scratch. To do that would involve considerable effort involving the calculation of character sizes and alignments, as well as the aesthetics of character construction. It is, however, reasonable to modify an existing character to get a new one. For example, the first effort at producing a double dot was to modify the German umlat from the Roman-8 extension set. This did not work very well, because the entire character cell is not saved. Each character is only as big as the

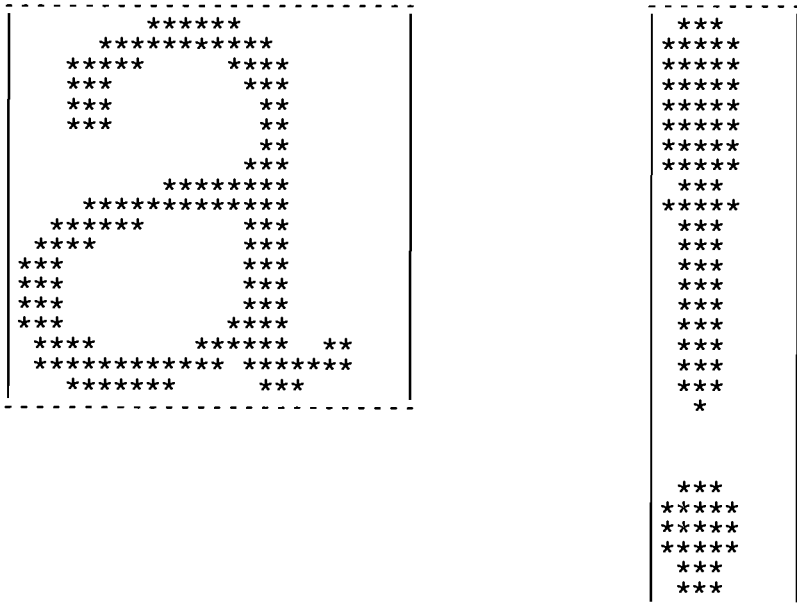


Figure 10 - Two characters in cells

smallest box that will contain it. For example, two characters that take up considerably different spaces within the cells are shown in figure 10.

Therefore, FONTEDIT allows any character to be edited within its already existing character space. In addition, one character can be substituted for another. Thus, if the requirement is to turn the exclamation point into a happy face, simply replace the exclamation point with a capital O and then edit it. The original O is unchanged.

In some cases, it is easier to start with a large empty space, rather than a pre-existing character. Such a space, called a CELL, can be substituted for any character. That character need not exist (as shown below).

Since edit operations are done directly into the file, FONTEDIT allows a COPY operation for creating a new file to modify. COPY also provides for copying only the first record of the file. This record is the font descriptor. A file containing only the font descriptor is an empty font file. Using the REPLACE command, empty CELLS can be added to this file and a character drawn in each one. Some examples may make the technique clearer.

For the purpose of creating a ballot, we need to make a large square box, about the size of a capital letter. The letter M is an appropriate mask. Using FONTEDIT, make a copy of the file PR100R12.USPM. Call it PR100R12.NEW. This is a standard 10 point prestige elite font. Now edit PR100R12.NEW. Edit the character M to produce the square box. This new file can be uploaded to the printer. Accessing the character M in this font will produce the square box. However, a font with all the same characteristics may already be loaded, perhaps on a cartridge. This is where the ID numbers are important. By assigning a known ID number to the font, it can be unambiguously selected, even if there is another font loaded with identical characteristics.

Alternatively, the square box could have been created by using the REPLACE command to replace an unused character in the Roman-8 extension of an 8-bit character set with the capital M and proceeding as above.

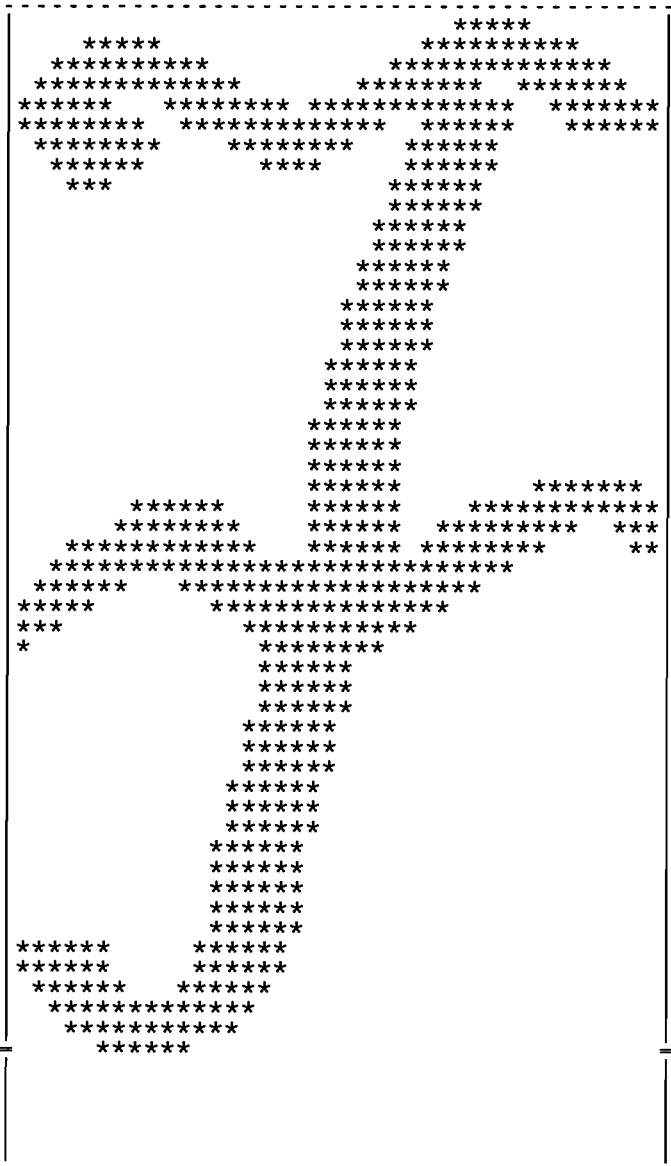


Figure 11. A character created from scratch

A third method would be to create a new font from scratch. The letters in FONTEDIT were created in script in this manner for one of the demo files (the script F is shown in figure 11). First the font descriptor was copied from the file PS140B08.USPM, the presentation 14 point bold font, using the COPY command with the first record option. Next the REPLACE function was used on the new (empty) font file. The letter F was replaced with an empty CELL. Since there was no letter F in the font, that letter's value was assigned (automatically) to the new cell. Now the script F was drawn into the empty cell. After a few tries it is not hard to get a passable image. It took about 20 minutes to Word Processing System For HP1000 -19-

draw and debug each character. The T was easier than the others. The REPLACE function was used to copy the F into T (since no T existed, a new cell was created) and the cross-bar was removed. Since this is a sparse font, containing only 7 letters, it loads very quickly.

Once a character is displayed on the screen, a number of editing commands are available including those to insert a dot, delete a dot and copy and delete a row. The commands have been made similar to the screen mode commands in EDIT/1000, for convenience. For example, control-O will copy the current row down one (but will not insert the new row- rather it replaces what was there). Control-Q will read the screen and replace the record on the disk for that character with the pattern from the screen. Control-X will exit and **not** replace the disk record.

There is a DISPLAY option which is the same as edit mode, but prevents any writing to the disk. This is useful for seeing a font character without the risk of modifying it.

There are a few caveats. The LaserJet returns no information to the program. Therefore, it is not possible for FONTEDIT to detect such conditions as memory full or off-line. FONTEDIT makes use of the double-width character mode on the 239x series of terminals. This produces a more realistic aspect ratio for the characters. 264x terminals do not recognize these escape sequences. No graphics are used; however, color has been used to enhance the readability. At this time, only the multiplexor is supported (B or C only), not the BACI or ASIC cards.

With the advent of the LaserJet Series II with up to 4 megabytes of memory it should be possible to load all the fonts that are required in normal use at the beginning of the day. If the users agree on a set of ID sequences, they can all co-exist quite happily. Of course, at 9600 baud this will take awhile. It can be run at 19.2k baud on the multiplexor. If the LaserJet is near an HPIB connection, one can purchase a Centronics-to-HPIB converter and take advantage of the parallel bus.

III. SPELR

A spelling checker can be a very useful adjunct to a word processor. Spelling checkers on PC's are becoming more and more common as part of a total word processing system. SPELR was originally written many years ago, but was too slow for everyday use. With the advent of EMA it became practical to re-write it in a much more efficient form. At about the time SPELR became really useful, John Johnson's JSPELL became available. However, enough effort had already been put into SPELR that it was not worth abandoning it.

SPELR was designed with several specific goals in mind, some of which were suggested by experiences with PC and standalone word processors. The remainder of this section will describe both the general and the specific goals that were set and how they were implemented. In this discussion the source refers to the document to be checked for spelling errors. An error refers to a word not found in the dictionary. This may be a mis-spelled word, a correct word not in the dictionary, or a combination of characters that is correct, but is not a word.

General goals refer to characteristics that most programs of this nature should have.

a. The program should, of course, be fast. For a spelling checker that means that the user should not have to wait "a long time" for results. Some spelling checkers scan the entire document before any feedback is given to the user. SPELR is interactive. The source is scanned and when an error is found SPELR stops and displays the error. Therefore, it appears fast to the user, since most of the time is spent waiting for the user to make a decision.

b. The program should run on all HP1000's. Since HP uses EMA for most of its standard

utilities now, SPELR can make free use of EMA. The entire dictionary is stored in EMA, which accounts for much of the speed of SPELR.

- c. It should make it difficult to corrupt the dictionary. It should allow the dictionary, itself, to be easily checked for errors.

The dictionary

The dictionary must be large enough to hold a useful number of words, but must also allow them to be accessed quickly. It must be capable of modification by the user. The dictionary for SPELR originally consisted of a type 1 file designed for very fast access. When EMA was implemented, this entire file was simply read into EMA, leaving the structure intact. Each 128-word record contains 31 cells of 8 characters each. The first 26 records in the file are keyed to the first letter of the word stored in the record. Thus record 1 contains 31 words beginning with the letter "a", while record 26 contains the "z" words. As words are added to the dictionary the cells are filled in the correct record. The 125th word of each record is the pointer to the next available cell in that record. The 126th word points to the next record number containing words beginning with that letter. Thus, to check the entire dictionary for a word beginning with "d", all that is necessary is to check the 31 cells of the 4th record, then all the cells of the record indicated in the 126th word of the 4th record, etc.

All words in the dictionary are stored in lower case. When searching for a word, the initial test is on the first 2 characters of every cell. Only when the test is equal are the remaining characters checked.

The astute reader will recognize that this scheme allows only for words of up to eight letters in the dictionary. However, provision is made for "double words". If the high-order bit of the cell is set, then that cell is treated as the next eight characters of the previous cell. Therefore, the practical maximum word size is 16 characters. Provision was made to expand to 24, but it has not been necessary to implement it. "Double word" cells are not allowed to cross record boundaries.

Certain decisions had to be made which were a matter of taste and the amount of effort required. All words are stored in the dictionary just as they are found. Thus, plurals, possessives, etc. are all treated as separate words. Hyphenated words are currently treated as two separate words. As mentioned before, SPELR has no sense of case.

Reading the type 1 file into EMA is the most time-consuming part of the SPELR operation. Therefore, Shared EMA has been implemented. If the Shared EMA partition contains the dictionary from a previous run, the file will not be read in again on successive runs of SPELR. Naturally, tests are performed to ensure that the file is re-read if words were added to the dictionary on the last run. In RTE-6, where the EMA labels are fixed, problems can occur if another program using the same shared EMA is run while SPELR is running. It is desirable for SPELR to have its own SHEMA partition. However, it will also run using standard EMA.

Words can be added to the dictionary as they are checked. Thus, a dictionary can be created simply by asking SPELR to check a source document and telling it to add each word to the dictionary. Similarly, a document that has been checked for spelling by another spelling checker can be run through SPELR and all error words added to its dictionary. To simplify this process, there is a "hidden" code that can be used to tell SPELR to simply add all of the error words to its dictionary without stopping for verification. This hidden code is found in the documentation accompanying SPELR.

Normal operation

To check a source document, simply tell SPELR the name of that document. It will stop at the first Word Processing System For HP1000

error word and ask for a decision. At this time the user has several options:

1. Add the word to the dictionary. The word will not be displayed again even if it occurs later in the document. However, after the entire document has been checked, a list of the added words is presented. The user can then delete any that were added by mistake. He is then asked for a final verification before the words are actually added to the dictionary.
2. Ignore the word. In this case it is not added to the dictionary and it is not displayed again.
3. Change the word. The user can re-type the word. The re-typed word is also checked. SPELR will not proceed until the word is spelled correctly or is added or ignored. After the document has been checked, the user is told that words were changed and he is asked for verification before the original document is modified.
4. Show similar words. SPELR will find all words in its dictionary that are similar to the error word and will display them. The process used is described later.
5. Exit. Since words may have been previously modified, this option tells SPELR to process the remainder of the document without stopping. The effect is as though the Ignore option was permanently on.

Similarity checking

The "show" option searches the dictionary for words that sound like the mis-spelled word and displays them in alphabetical order. As described above, all of the dictionary entries for words beginning with the same letter as the error word are searched. The SOUNDEX code for the error word is calculated and compared against all words in the dictionary with the same SOUNDEX code. A restriction has been placed on the comparison that the error word and the matched word must agree in length to within three letters. This seemed like a reasonable way to keep the number of spurious matches to a minimum.

The SOUNDEX code is a standard method of encoding a word in such a way that words with a similar sound generate the same code. The first letter of the word is kept, all vowels are discarded, and multiple-letter sequences are reduced to one letter. Each remaining letter is assigned a not-necessarily-unique weight and these weights are combined to produce a single number. Although a SOUNDEX code normally begins with the first letter of the word, this is unnecessary here because only words beginning with the same letter as the error word are searched.

Special functions

A number of special functions have been implemented. Simply typing SPELR,?? will display the various options. The dictionary can be saved in a standard format and restored from this format. The format is simply an ASCII file with one word per line. This allows the dictionary to be checked for erroneous words. It is this saved version of the standard dictionary that is supplied with SPELR. This way the user will not inadvertently overlay his own dictionary. Provision is made for by-passing the special escape sequences used with the display if a non-HP terminal is used. The running count of the line number, displayed in the upper right of the screen can be suppressed. This is handy if SPELR is run from a slow modem, since putting the display there involves a lot of overhead.

Hints

It is certainly useful to check a word against a (paper) dictionary before adding it. Once a mis-spelled word is added to the dictionary, it is not easily found. SPELR ignores words that begin with special characters or numbers. Thus, for example, TYPER commands are transparent to SPELR. If a number appears as an error word, there is a good chance that the first character is a small "L" typed in place of a number one. A decision was made to treat hyphenated words and words containing a slash as separate words. This is done in one (commented) statement in SPELR and can easily be changed. Currently the dictionary allows up to 600 blocks (about 18,000 words). This should be adequate for most applications. However, it should be possible to change this with the size parameter noted at the beginning of SPELR. Note that before running SPELR the disk area to contain the dictionary must be set in the source code and the code re-compiled. The default is to LU 15.

IV. Conclusion

The programs presented here have been developed over a number of years with a considerable amount of input from users all over the world. The initial inspiration for SPELR and the LaserJet version of TYPER came from the production of the proceedings of the 1984 INTEREX Technical Users conference (San Jose II). Many of the improvements to TYPER have originated in helpful comments and advice from Alan Whitney and many other HP1000 users and from the scientific publication requirements of the Astrometry Department of the U. S. Naval Observatory.

Each of these programs can be used by itself, but combined they provide the user with a professional word processing system with great flexibility at a reasonable price.

APPENDIX A
Sample Output From TYPED

U.S. NAVAL OBSERVATORY

34th and Massachusetts Ave., NW
Washington, D.C. 20392-5100

1987 INTEREX HP TECHNICAL COMPUTER CONFERENCE

SAN JOSE, CALIFORNIA
October 18-22, 1987

10 March 1987

Dear John,

Plans for the INTEREX conference in San Jose are progressing rapidly. As a separate conference for technical computer users, we will be less constrained to an overall conference format. Therefore, we hope to try some different types of sessions. I anticipate only three parallel HP1000 tracks and one HP9000 track. There will be at least one tutorial every day. Abstracts of papers will be reviewed by the HP1000/9000 paper review committee (hopefully, including yourself). Paper reviewers are needed in all of the categories, especially for the HP9000.

Thank you for your past interest in INTEREX.

Sincerely,

APPENDIX B

The Commands That Produced
APPENDIX A

```

\!v=roman-8,fixed,12pitch,10point
\hz,,,12
\ceAPPENDIX A
\CESample Output From TYPER
\fg5
\hz,,,10
\vt,,,8
\!r=[152]
\ceU.S. NAVAL OBSERVATORY\
\hz,,,13
\!S=[154]
\CE34th and Massachusetts Ave., NW
\ceWashington, D.C. 20392-5100

\vt,,,6
\pp,,1
\HZ,,,8
\VT0
\!a=usascii,10pitch,14point,presentation,bold,fixed
\CE||1987 INTEREX HP TECHNICAL COMPUTER CONFERENCE
\!a=usascii,12pitch,12point,gothic,bold,fixed
\hz,,,8
\CESAN JOSE, CALIFORNIA
\CEOctober 18-22, 1987

\!G=[5002],PFILE=/PROP_FONT/TR100IPN.R8PM,LINK=ITALIC
\!J=[5001],PFILE=/PROP_FONT/TR100BPN.R8PM,LINK=BOLD
\!k=[5000],PFILE=/PROP_FONT/TR100RPN.R8PM
\hz
10 March 1987

```

Dear John,

\PP,3

Plans for the INTEREX conference in San Jose are progressing rapidly. As a separate conference for technical computer users, we will be less constrained to an overall conference format. Therefore, we hope to try some different types of sessions.

I anticipate only three parallel HP1000 tracks and one HP9000 track. There will be at least one tutorial every day.

Abstracts of papers will be reviewed by the HP1000/9000 paper review committee (hopefully, including yourself).

Paper reviewers are needed in all of the categories, especially for the HP9000.

Thank you
for your past interest in INTEREX.

\ce Sincerely,



T_EX on the HP1000 and Laserjet+

Alan R. Whitney
MIT Haystack Observatory
Westford, MA 01886

T_EX, the typesetting system intended for the creation of beautiful documents, and *especially* for documents that contain a lot of mathematics, is now available from the Interex CSL/1000 library. Written originally by Donald Knuth at Stanford, T_EX has been ported to many machines, and is without question the penultimate word-processing system available today. With no other word processing software is it as *easy* to write a mathematical expression such as

$$I_0(z) = \sum_{k=0}^{\infty} \frac{\left(\frac{z}{2}\right)^{2k}}{(k!)^2}$$

nor to have it look as beautiful!

The version of T_EX available from CSL/1000 was ported to RTE-A by Tor Lillqvist of the Technical Research Centre of Finland, and produces output on the HP Laserjet+. The implementation is complete in all respects, including support for the popular L^AT_EX and A_MS-T_EX packages. This paper will discuss some of the capabilities of T_EX, as well as its use on the HP1000. Anyone with a need to produce the highest-quality typeset documents should investigate the capabilities of T_EX.

1 Introduction

I suspect that a large number of you are already familiar with the T_EX typesetting system, or at least have heard of it. Particularly if you need to write documents with any substantial amount of mathematics. T_EX has been quietly invading the engineering and scientific world for several years now, and has become a *de facto* standard in parts of the technical publishing world; the *Journal of the American Mathematical Society*, among others, now accepts papers on electronic media in T_EX format.

T_EX is a typesetting/formatting system, as opposed to the WYSIWYG ("What you see is what you get") word-processors that are so popular on PC's. And, although T_EX is not difficult to use, it is somewhat more complicated than the run-of-the-mill word-processing systems. This slight complication in use, however, is more than offset by the flexibility and extreme power of T_EX.

2 T_EX Input Files

T_EX input files are ordinary ASCII files generated by any ASCII editor. [They are therefore extremely portable from one machine to another.] T_EX defines a default environment in which ordinary text files made up of ordinary sentences and paragraphs are transformed into a beautiful typeset page with no additional commands. A paragraph is defined, in the input file, as a contiguous set of non-blank lines. Paragraphs are separated by one or more blank lines.

T_EX uses ASCII *control sequences* and *control symbols* embedded in the input to specify special things. A *control sequence* is simply an 'escape character' (usually a backslash) followed by one or more letters(A-Z,a-z), followed by a space or *non-letter* character. For example, the character string `{\it this text is italicized}` is converted by T_EX into *this text is italicized*, and the character string `\TeX` is converted to T_EX. A *control sequence* consists of the escape character followed by a single *non-letter* character, and need not be followed by a space. For example, the string of characters:

`George P\'olya and Gabor Szeq\'o`

is converted by T_EX to 'George Pólya and Gabor Szeqő'. Many of the control sequences are simply the names of special characters used in math formulas. For example, you type `\pi` to get π and `\sum` to get \sum .

There are 900 or so control sequences defined in T_EX, but that isn't the whole story. It's easy to define more. For example, if you want to substitute your own favorite names for some particular math symbols, you're free to go ahead and do it.

3 Mathematics

This is where T_EX really shines. Mathematics formulas are written in a manner which is thoroughly understandable. The character `⌘` is used to place T_EX into *math mode*. For example, the string `⌘\sqrt{x+y}⌘` is converted into $\sqrt{x+y}$. And the string

`⌘⌘I_o(z) = \sum_{k=0}^{\infty} \frac{\left(\frac{z}{2}\right)^{2k}}{\left(k!\right)^2}⌘⌘`

is processed by T_EX to produce

$$I_o(z) = \sum_{k=0}^{\infty} \frac{\left(\frac{z}{2}\right)^{2k}}{(k!)^2}$$

With just a little study, you will have to agree that the text that produces this relatively complicated equation is in fact quite readable. Greek text and special characters are easily handled. For example, the string `⌘⌘x = \frac{\alpha + \Phi/2}{\sigma^2 + 1}⌘⌘` produces

$$x = \frac{\alpha + \Phi/2}{\sigma^2 + 1}$$

T_EX on the HP1000 and Laserjet+ 2

T_EX takes care of all the details of formatting the mathematical expression so that it looks and reads properly. A large library of mathematical symbols is available to write virtually any expression that is desired.

4 A Sample of Other Capabilities

The full capabilities of T_EX are simply too numerous to illustrate completely. But we will show just a few simple examples here. Of course, you can choose virtually any type size you wish, from *tiny* to *small* to *normal* to *large* to **Large** to **LARGE**. And any type style from Roman to *Emphasized* to **Boldface** to *Italics* to *Slant* to **Sanserif** to **CAPS** to **Typewriter**, all with simple commands. Footnotes¹ are easy to generate and are handled automatically by T_EX.

A common word-processing task is to generate itemized lists. T_EX has extreme flexibility in this regard and is an expert at the task.

- This is the first item
 1. sub-itemized lists are also easy to generate
 2. T_EX takes care of placing everything properly on the page
- This is the second item. Itemized lists may also be enumerated as well.

Of course, T_EX allows use of different type styles and fonts, as well as mathematical expressions, to be placed anywhere within these itemized lists.

Lists of tabular items are also easy to generate, even including horizontal and vertical lines for visual separation:

<i>type</i>	<i>style</i>	
small	red	skinny
rather tall	blue	fat

This is just small sample of the capabilities of T_EX. Other capabilities include making Tables of Content, Indexes, and Bibliographies. The capabilities of T_EX are amply documented in the references listed at the end of this paper.

5 Implementations of T_EX

T_EX has been implemented on a large variety of computers, from PC's to supercomputers. The T_EX source code is written in a special high-level literate language called WEB, which is then translated into PASCAL code using a program called TANGLE. Theoretically, then, if the PASCAL compiler on any given machine is up to the task, T_EX can be implemented on that machine.

¹A footnote is generated simply typing the footnote in the main text with a simple command. Mathematical expressions are perfectly legal in footnotes, too.

\TeX has been ported to the HP1000 on two occasions (to my knowledge). First, a version of \TeX is available for both RTE-6VM and RTE-A from J_DJ Wordware which supports output to several devices, including the HP2680 and HP2688 laser printers. And second, Tor Lillqvist of the Technical Research Center of Finland has ported \TeX (actually $\text{\TeX}82$) to RTE-A which supports output only to the HP Laserjet+ (or compatible); Dr. Lillqvist has generously donated this version of \TeX to Interex for inclusion in the CSL/1000 library.

As you can well imagine, \TeX is not a small program. As implemented under RTE-A, it requires some 700k bytes of memory and some tens of Megabytes of disc space for a full-up \TeX system. Included are full implementations of the popular \LaTeX and \AMS-TeX , which are really \TeX molded into specific environments for specific tasks. In fact, this paper was actually written under \LaTeX , which provides a nice environment for document preparations such as this one.

When \TeX , \LaTeX , or \AMS-TeX are run, they produce a device-independent output file ('DVI' file, so-called). The data in the DVI file is then passed through a device-specific program to create the actual output to the printing device. In the case of the Lillqvist implementation, this is a program called DVIPLUS which was written by Dr. Lillqvist specifically to write output to the Laserjet+.

\TeX is in the public domain, and its algorithms are published. Anyone is free to attempt to implement \TeX on any machine they so wish, free of charge. However, Prof. Knuth, the author of \TeX , disallows any specific implementation from calling itself ' \TeX ' unless two conditions can be met; (1) The person who wrote it must be happy with the way it works at his or her installation; and (2) the program must produce the correct results from **TRIP.TEX**. **TRIP.TEX** is a fiendish test file that tests the remotest corners of a \TeX implementation. I am happy to report that the RTE-A implementation by Lillqvist passes the **TRIP.TEX** test with flying colors.

Interestingly, \TeX is so bug-free that Prof. Knuth offers a monetary reward (\$10.24, at last report) to anyone who finds a bonafide bug!

6 The Rest of the Story

What I have outlined so far is really just a sampling of \TeX . For anyone interested in further pursuing \TeX , numerous publications on various aspects of the subject exist, some which are listed below. In addition, a \TeX User's Manual specific to every installation normally exists as a guide to local \TeX users. An active \TeX user's group, called TUG (for ' \TeX User's Group'), is open to all comers for a modest fee. Their periodic publication, called *TUGboat*, comes as part of the membership.

If \TeX implemented on the HP1000 and Laserjet+ is of interest to you, contact the Interex office for information on obtaining a copy of this invaluable CSL/1000 contribution.

Bibliography:

D. E. Knuth, *The \TeX book*, Addison-Wesley Publishing Company, 1986.

L. Lamport, *\LaTeX :A Document Processing System*, Addison-Wesley Publishing Company, 1986.

M. D. Spivak, *The Joy of \TeX* , American Mathematical Society, 1986.

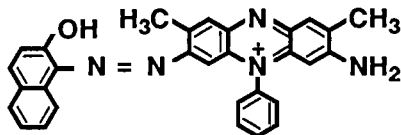
GOSPEL: Generating Organic Structures for Printing Entirely on a LaserJet+

Jock McFarlane

David Sarnoff Research Center
CN 5300
Princeton, NJ 08543-5300

Abstract

An integrated set of FORTRAN programs, system command files, and edit command files for a Hewlett-Packard 1000 (M/E/F or A-Series) minicomputer has been developed to provide the capability of including complicated organic structures in a report printed by a Hewlett-Packard LaserJet+. The FORTRAN programs download a set of "macros" to the LaserJet+. Each macro is a set of commands that generates a graphic design that is a given organic six-membered ring or bonds within or attached to a ring. To generate the desired organic structure, the user includes a set of simple commands within the report text, e.g., %H3 for a horizontal ring with 3 double bonds. The text of the report is usually prepared with commands of a text-formatting program such as WOLF or TEXED. The text-formatter output containing the commands for drawing structures is sent to a scratch file. That file is then edited using an edit command file to change the commands for generating the structures to the appropriate escape sequences that execute the macros in the LaserJet+. Once the draft text has been created, the user transfers control to a system command file which runs the text-formatter, generates the scratch file, schedules the editor, and sends the output to the LaserJet+. For example, the structure shown below was created using GOSPEL commands.



Introduction

Reports and tables discussing cyclic organic structures usually require the inclusion of figures showing structures of the molecules of interest. These figures are often drawn by hand with appropriate templates and must be added to the final text without error. The need for an easier way to generate these structures as part of the draft text and to edit them as required has led to the method discussed here for printing organic structures with an HP LaserJet+.

Several areas of interest to organic mass spectroscopists involve chemical structures assembled using regular hexagons. Although this project has addressed those particular needs, similar procedures can be developed for structures containing other regular polygons.

This paper will give an overview of how the system uses FORTRAN programs, text formatting programs, the editor, and system command files on an Hewlett-Packard 1000 Minicomputer (M/E/F or A-Series) connected to an HP LaserJet+. A complete users guide to the commands which generate structures is included.

An HP LaserJet+ (not just a LaserJet) is required, as the procedure utilizes the "macro" capability of the LaserJet+. Similar programs could be developed using another computer system, but the LaserJet+ would still be necessary for printing. The interested reader should refer to the LaserJet Technical Reference Manual (HP 02686-90912) for more information on the LaserJet+.

LaserJet+ Macros

The first step in generating any structure is to define basic structural components and send to the LaserJet+ the raster graphics commands to create those components. The structures printed with the system discussed here are hexagons, either in a vertical or horizontal orientation, with double bonds along any one, two, or three edges. In addition, single or double bonds can extend from any vertex of the hexagon. Thus, the components are the hexagons, interior bonds, and exterior bonds.

A FORTRAN program is used to send the graphics information to the LaserJet+ and to assign each structural component a Macro ID Number. The LaserJet+ has a limit of 32 Macro ID's, and 28 ID's are needed to identify the structure components in this system.

A macro is executed by sending `<esc>&f nny 2X` to the LaserJet+, where `<esc>` is the escape character, `nny` identifies the current macro as ID #nn, and `2X` is the command to execute the current macro. The characters are case sensitive, i.e., `f` and `y` must be lower case and `X` must be upper case. The spaces are for clarity here and are allowed but not required.

Edit & System Command Files

Printing the various hexagonal rings and bonds can involve calling for as many as three different macros, as well as moving the LaserJet+ graphics cursor. Therefore, a set of commands was developed that enables the user to indicate what structure elements are required for printing. These commands can be imbedded in the text of a report or table and translated (via the edit command file) into the appropriate LaserJet+ escape sequences before sending the text to the LaserJet+.

Text files on the HP1000 systems are prepared using the Hewlett-Packard editor, EDIT/1000. A file from which a report is to be generated may be prepared with commands that will be recognized by a text formatting program. For example, this report is being printed using TEXED on an HP A-900 minicomputer. Users on the E-Series system employ the program WOLF for text preparation. Each of these programs can direct the formatted text output to another disc file. A convention has been established to call this temporary disc file ***TEMP***.

Once created by TEXED or WOLF, the ***TEMP*** file is then edited to change the simple GOSPEL commands for structure generation into the appropriate escape sequences before sending ***TEMP*** to the LaserJet+. The edit commands to accomplish this are contained in a separate edit command file.

GOSPEL: Generating Organic Structures for Printing Entirely on a LaserJet+

The entire sequence of scheduling the text formatting program, running EDIT/1000 on ***TEMP*** with transfer of control to the Edit Command File, and sending the edited version of ***TEMP*** to the LaserJet+ is accomplished using a system command file or transfer file. The transfer file for M/E/F systems, ***JWOLF**, assumes HP File Manager (FMGR) files and the WOLF text formatting program. The command file for A-Series systems, **TX.CMD**, assumes Command Interpreter (CI) files and the TEXED text formatting program. Both command files expect the name of the file containing the source text for the text formatting program to be a passed parameter.

GOSPEL Commands

Before discussing the commands which generate an organic structure, it may be useful to highlight some general considerations for printing text and graphics on a LaserJet+. Contrary to our usual experience in sending text to a line printer, the LaserJet+ doesn't care where text is being placed on the page as it is sent from the computer. That is, with appropriate commands, the bottom line of a page can be sent first and the top line as the last part of the data stream. A page is finally ejected from the LaserJet+ only when the form-feed character is sent or when the LaserJet+ is commanded to print a line below the current printing area on the page (exactly where that is depends on page size and printing orientation). Thus, text and figures can be placed anywhere on the page, in any order.

Cursor Positioning

Positioning the LaserJet+ cursor on the page can be accomplished in several ways using the **%Move** command with an appropriate argument. In the examples that follow, the "**%Move**" characters are changed by the edit command file to the appropriate escape sequence for transmission to the LaserJet+. Refer to the LaserJet Technical Reference Manual for a more complete discussion. When the numbers used to position the cursor the cursor are given in the command without "+" or "-" signs, the command is to move to that ABSOLUTE position. If, however, the number is preceded by "+" or "-", the command is to move the indicated RELATIVE amount from the current cursor position. The 0,0 position is the upper left corner of the page and points to the right and down are positive numbers.

Row - Column: %Move nr mC

Moves to a row and column position are given by **%Move nr mC**, where n & m are the current row and column positions of the current font. (The actual position on the page will change with different fonts.) The following three command sequences all place the cursor at the 21st row and 33rd column (spaces for clarity, not required):

- 1) **%Move 21r 33C**
- 2) **%Move 33c 21R**
- 3) **%Move 21R %Move 33C**

Note that "r" and "c" are case specific. Row and column moves are most useful for relative moves, due to the fact that the absolute page position is font dependent.

Decipoint - Horizontal, Vertical: %Move nv mH

Moves to an absolute position on the page in "decipoints" at a resolution of 1/720 inch are given by **%Move nv mH**, where n & m are the vertical and horizontal positions in decipoints, respectively, and are independent of font selection. Again, the "v" and "h" are case specific. The center of an 8.5" x 11" sheet of paper is at approximately %move 3640v 2950H.

Store & recall current cursor position: %Push & %Pop

The current cursor position can be saved (**%Push**) in a position stack and recalled later (**%Pop**). Up to 20 positions can be saved, with the last position saved being the first one restored. This feature is useful in generating organic structures when one realizes that the text required to create even a fairly small molecule may exceed the 150 characters per line allowed by EDIT/1000. Ending a command line with "%Push" and starting the next with "%Pop" has the same effect as writing one continuous line. (Without storing the cursor position, the next line will start where the line-feed & carriage-return place the cursor.)

Hexagon Commands

The command to print a hexagon is of the form

%Obs

where

% = always first command character

O = Orientation, either H or V (horizontal or vertical)

b = Number of double bonds: 0, 1, 2, 3, or P for 2 parallel bonds

s = Side designation as follows:

b = 0, **s = not used**

b = 1, **s = side for double bond, top or top right is 1 & 2 .. 6 clockwise**

b = 2, **s = side between two double bonds, identified as above**

b = 3, **s = not used**

b = P, **s = side parallel to 2 parallel bonds, (1, 2, or 3)**

A horizontal hexagon has one pair of sides horizontal. A vertical hexagon has one pair vertical. Thus, a horizontal hexagon with two double bonds parallel to the horizontal sides is printed with **%HP1**, a vertical hexagon with one double bond along the left vertical side is **%V15**, and a horizontal hexagon with double bonds along the lower left and lower right sides is **%H24**. Note that none of the GOSPEL commands is case-specific. All 34 possible rings are shown in Table I.

Exterior Single Bond Commands

The command to print an exterior single bond on a hexagon is of the form

%OBv

where


% = Always first command character


O = Orientation, either H or V (horizontal or vertical)

B = Always "b" or "B"

V = Vertex for exterior bond, top or top right is 1 & 2 .. 6 clockwise

An exterior bond command assumes that the cursor is at the right edge of a hexagon, i.e., a hexagon has just been printed. Thus, if several exterior bonds are required, each command must be preceded with **%Push** to save the cursor position and followed with **%Pop** to return the cursor to the right edge of the hexagon for the next exterior

bond command. The command line for a horizontal hexagon with a bond at the upper right vertex is **%H0%HB1**  , while the command for a vertical hexagon with bonds at the upper left and lower right vertices is

%V0%Push%VB6%Pop%VB3  . Note that there is no command for a bond at the

left-pointing vertex of a horizontal hexagon (**%HB5**). The assumption is that the bond would be printed before printing the hexagon with a **%HB2** command. The exterior bond commands are given in Table II.

Exterior Double Bond Commands

The command to print an exterior double bond on a hexagon is of the form

%ODv

where


% = Always first command character

O = Orientation, either H or V (horizontal or vertical)


D = Always "d" or "D"

V = Vertex for exterior bond, top or top right is 1 & 2 .. 6 clockwise

The exterior double bond command also assumes that the cursor is at the right edge of a hexagon, i.e., a hexagon has just been printed. The command line for a horizontal

hexagon with a double bond at the upper right vertex is **%H0%HD1**  ,

while the command for a vertical hexagon with double bonds at the upper left

and lower right vertices is **%V0%Push%VD6%Pop%VD3**  . The only exception

to the assumption that the cursor is at the right edge of a hexagon when a double bond command is given is for a horizontal double bond at the left vertex of a horizontal hexagon. In this case, the hexagon is printed AFTER the double bond. The exterior double bond commands are given in Table III.

Combining Hexagons



After a hexagon has been printed, a second hexagon can be printed immediately adjacent to the right, directly above, directly below, or adjacent to either of the slanted edges on the right side of the hexagon. (No commands have been developed to place a second hexagon to the left of the first hexagon.) The command to move the

cursor before printing a second hexagon is of the form

%0aa

where

- %** = Always first command character
- 0** = Orientation, either H or V (horizontal or vertical)
- aa = TP** Second hexagon directly above first hexagon
- aa = BT** Second hexagon directly below first hexagon
- aa = UP** Second hexagon adjacent to upper right slanted edge
- aa = DN** Second hexagon adjacent to lower right slanted edge

There is no special command to print a second hexagon to the right of the first hexagon, i.e., %H0%H3 and %V0%V3 print as  and , respectively. It should be noted that the hexagons in real organic structures will always join along adjacent edges, never with vertices touching. Commands to combine hexagons are given in Table IV.

Substituents and Symbols

Bonds exterior to rings may connect 2 rings or may connect an atom or group of atoms (substituents) to a ring. The GOSPEL command to print an exterior bond leaves the cursor at the end of the bond so that the substituent information can be printed if desired. If more than one letter is to be printed, it may be necessary to move the cursor to the left a sufficient number of spaces to accommodate the characters to be printed.

The LaserJet+ text font most appropriate for the structures generated with GOSPEL commands are the 12 and 10 point Helvetica bold fonts. These fonts are on the 92286U Forms Portrait Font Cartridge or are soft font files HV120BPN.USP and HV100BPN.USP provided with the HP 33411AC/34412AC Soft Font.

Subscripts and superscripts in chemical formulae can be generated with the commands %up and %dn. The %up and %dn commands generate a half-space up or down shift, respectively. The cursor must be returned to the starting line after the half-space shift. For example,

H%dn2%upO

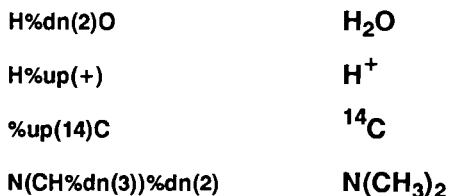
H₂O

H%up+%dn

H⁺

Another way to print superscripts and subscripts is to establish the 12 point Helvetica bold font as the primary font and the 10 point Helvetica bold font as the secondary font. Then, if the text to be superscripted or subscripted is included within parentheses after %up or %dn, the text will print in the secondary font. Since this command returns the cursor to the starting line after the shifting up or down, a second

shift command is not necessary. For example,



Two other GOSPEL commands permit special superscripts. A "+" (%+) or a vertical double bond (%VDB) can be placed directly above an elemental symbol. For example,



Examples of these commands are given in Table V.

Complex Structures

The set of GOSPEL commands permits the generation of rather complex organic structures. Examples of several molecules are shown in Table VI. These structures were generated for use in reporting results from organic mass spectrometry. Careful study of these examples should permit duplication of similar and even more complicated molecular structures. The heavy brackets around some examples are generated with the line drawing character font.

Acknowledgement

This project began as a 'there oughta be a better way' challenge from P. Jane Gale and has benefited from her cheerful encouragement, honest criticism, and intelligent use of the resulting GOSPEL system.

Table I : GOSPEL Commands for Hexagons

Horizontal Hexagons

Double Bonds:	None	One	Two	Two parallel	Three
%H0		%H11	%H21	%HP1	%H3
		%H12	%H22	%HP2	
		%H13	%H23	%HP3	
		%H14	%H24		
		%H15	%H25		
		%H16	%H26		

Vertical Hexagons

Double Bonds:	None	One	Two	Two parallel	Three
%V0		%V11	%V21	%VP1	%V3
		%V12	%V22	%VP2	
		%V13	%V23	%VP3	
		%V14	%V24		
		%V15	%V25		
		%V16	%V26		

Table II : GOSPEL Commands for Exterior Bonds













Horizontal Hexagons		Vertical Hexagons	
%H0%HB1		%V0%VB1	
%H0%HB2		%V0%VB2	
%H0%HB3		%V0%VB3	
%H0%HB4		%V0%VB4	
%HB2%H0		%V0%VB5	
%H0%HB6		%V0%VB6	

Table III : GOSPEL Commands for Exterior Double Bonds













Horizontal Hexagons		Vertical Hexagons	
%H0%HD1		%V0%VD1	
%H0%HD2		%V0%VD2	
%H0%HD3		%V0%VD3	
%H0%HD4		%V0%VD4	
%HD5%H0		%V0%VD5	
%H0%HD6		%V0%VD6	

Table IV : GOSPEL Commands to Combine Hexagons








Horizontal Hexagons	Vertical Hexagons
%H0%HTP%H3 	%V0%V3 
%H0%HUP%H3 	%V0%VUP%V3 
%H0%HDN%H3 	%V0%VDN%V3 
%H0%HBT%H3 	

Table V : GOSPEL Commands for Text

(Primary Font: Helvetica Bold, 12 point Secondary Font: Helvetica Bold, 10 point)

Superscript	H%up(text)	H^{text}
Subscript	H%dn(text)	H_{text}
Centered "+"	N%+	N⁺
Vertical Double Bond	C%VDBO	O C

Table VI-(a) : Examples of Complex Organic Structures

%Move1600v3300H%H3%HB2 N=N %HB2%Push

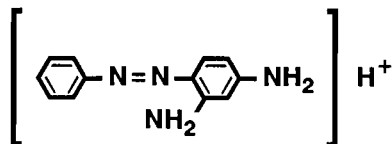
%Pop%H3%Push%HB2 NH%DN(2) H⁺

%Pop%HB4%Move-4CNH%DN(2)

%f4

%Move-7R

q	w
:	:
:	:
:	:
:	:
a	s



Example 1

%f1

%Move2300v3800H Example 1

%Move3500v3100H%H3%Push%HB1COOH

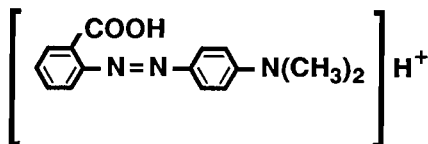
%Pop%HB2 N=N %HB2%H3%Push

%Pop%HB2 N(CH%dn(3))%dn(2) H%up(+)

%f4

%Move-5R

q	w
:	:
:	:
:	:
:	:
a	s



Example 2

%f1

%Move4000v3500H Example 2

%Move4900v3700H%V3%Push%VB5%Push

%Pop%Move-13C(CH%dn(3))%dn(2)N

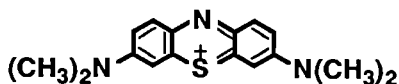
%Pop%Push%VB2N%Pop%Push%VB3S%+

%Pop %V22%Push%VD6%Pop%Push

%Pop%Push%VD5%Pop%VB3%Push

%PopN(CH%DN(3))%dn(2)

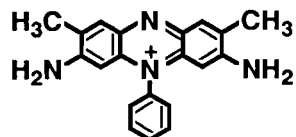
%Move5300v3500H Example 3



Example 3

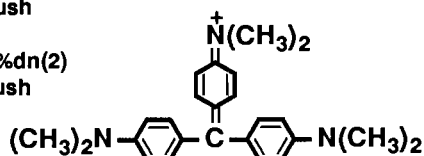
Table VI-(b): Examples of Complex Organic Structures

%Move1200v4000H%V3%Push%VB5%Move-5CH%dn(2)N
 %Pop%Push%VB6%Move-5CH%dn(3)C
 %Pop%Push%VB2N%Pop%Push%VB3N%+
 %Pop%Push%VDN%VDN%Move+0r+7C%Dn%V3%VB1
 %Pop %V22%Push%VD6%Pop%Push
 %Pop%Push%VD5%Pop%Push%VB3NH%DN(2)
 %Pop%VB2CH%DN(3)
 %Move1800v4000HExample 4



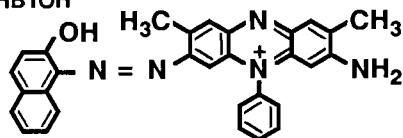
Example 4

%Move3100v3000H(CH%dn(3))%dn(2)N %Push
 %Pop%HB2%H3%HB2 C%Push
 %Pop%Push %HB2%H3%HB2 N(CH%dn(3))%dn(2)
 %Pop%Push%Vdb%Move-55h-65V%VP2%Push
 %Pop%VD1N%Push%+
 %Pop(CH%dn(3))%dn(2)
 %Move3500v3300HExample 5



Example 5

%Move4500v3000H%H24%Htp%H3%Push%HB1OH
 %Pop%HB2 N = %Push
 %Pop%Move+200h-130V%V3%Push
 %Pop%Push%VB5%Move-1CN
 %Pop%Push%VB6%Move-5CH%dn(3)C
 %Pop%Push%VB2N%Pop%Push%VB3N%+
 %Pop%Push%VDN%VDN%Push
 %Pop%Move+0r+7C%Dn%V3%VB1
 %Pop %V22%Push%VD6%Pop%Push
 %Pop%Push%VD5
 %Pop%Push%VB3NH%DN(2)
 %Pop%VB2CH%DN(3)
 %Move4900v3300HExample 6



Example 6

OPTICAL SOLUTIONS TO MASS STORAGE USER NEEDS

Ed Pavlinik
Hewlett-Packard Company
11413 Chinden Boulevard
Boise, Idaho 83714

INTRODUCTION

Nearly everyone involved in the world of information processing has heard of laser-based optical mass storage. Optical recording technology is new, exciting, and captures imagination - not only due to its inherent advantages over magnetic recording, but also because of its appeal to user emotional needs.

This paper will examine the types of optical technology used in mass storage and describe the advantages and disadvantages over magnetics. Next, a typical mass storage "user needs" analysis will be developed, showing where potential optical products fit into this matrix. Issues that arise when a new technology enters a market will be presented and analyzed. Finally, we'll take a look at Hewlett-Packard directions in the field of optical mass storage.

OPTICAL TECHNOLOGY

Why Optical?

The key advantage of optical technology is its areal density, which is anywhere from ten to twenty times higher than that of magnetic discs. Since a beam of laser light can be focused to an extremely small spot size, more data can be written on a unit area on the media. Given that the recorded mark (or data bit) is essentially circular, the resultant track density is nearly as high as the linear bit density. The end result is significantly higher areal density.

Another advantage of optical is the large head/media separation. Since a beam of coherent light can be narrowly focused by lenses, an optical head may be positioned a millimeter away from the media, compared to magnetics where the separation is a fraction of a micron. With a head/media spacing 4,000 times greater, optical disc drives are much less susceptible to contamination than magnetic discs. Fingerprints, dust, and smoke particles will not cause head crashes on an optical disc, since these contaminants are much smaller than the head/media separation. These same contaminants would be catastrophic to data stored on a magnetic disc! This attribute also makes the manufacture of optical disc drives easier, since tolerances are less stringent and manufacturing environments do not need to be as controlled.

As a result, these characteristics of optical recording make possible a very high capacity, removable media disc drive, using a single optical platter that is not only compact, but also reliable and manufacturable at low cost. Any applications, therefore, that require high capacity, removable media, and a low cost per megabyte are good candidates for optical disc drives.

These advantages of optical discs are offset somewhat by the slower performance of first generation devices, due to the higher mass that needs to be moved in an optical head. This head is a complex assembly containing a laser, lenses, mirrors, and positioning servos. Therefore, it moves more slowly than a magnetic head, resulting in higher access times. First generation devices will have access times comparable to floppy discs.

Read-Only

Optical technology has produced three major types of optical disc drives, the first category being **read-only**. As the name implies, data is pre-recorded at the factory at the time of disc manufacture. After distribution of the disc media, data may only be read; there is no write capability with this technology. A good example of this type of disc is the CD-ROM or Compact Disc Read Only Memory, derived from the audio world where compact discs (CD's) are used to store music. The CD-ROM is based upon the same type of technology with the addition of significant error detection and correction functionality. The CD-ROM disc is 120 millimeters in diameter (4.72 inches) and the standard format contains 550 megabytes on one side only. The set-up cost (or mastering cost) on a read-only disc is relatively high. However, once the mastering cost has been overcome, these discs are very inexpensive to reproduce. A typical mastering cost is anywhere from \$3,000 to \$10,000 to prepare the data and produce the first copy. Subsequent copies cost anywhere from \$30 to \$50. Turnaround times as short as one day have made optical publishing a reality!

Some of today's applications for CD-ROM consist of reference material distribution and storage. MicroSoft has announced its Bookshelf product which consists of a CD-ROM containing a dictionary, thesaurus, and a host of other reference material. Another CD-ROM is available that stores a full set of encyclopedia. Lotus Corporation has the One Source program where weekly financial and stock information is distributed on a CD-ROM and mailed to all subscribers of the service. This is a convenient way to distribute large quantities of data very inexpensively. Compare this to the time and cost of transmitting 550 Mbytes of data over a telephone line at 9600 baud!

Another type of read-only optical is the 12-inch video disc typically used for training and general information distribution. The industrial markets have applications for these read-only discs, while the consumer markets are nearly non-existent.

In summary, any application that enjoys a wide distribution, is written just once at the factory and retained for periodic reading, might be a potential candidate for read-only optical technology. This technology has been available for a number of years and products are shipping.

WORM

A second category of optical technology is called **WORM** (Write Once, Read Many). These discs can be written upon by the user once, but can never be erased or rewritten on the same area on the disc. If a file needs to be updated, a pointer must be provided to locate an updated copy on a new area of the disc. This technology uses a laser to actually burn a pit in the media substrate. Once the pit has been physically burned, there is no way to fill it back in again. As a result, if the data changes frequently, media capacity is soon exhausted.

WORM discs have been available for a number of years, and are now being shipped in 5.25-inch, 8-inch, and 12-inch form factors with storage capacities of several hundred megabytes to two gigabytes per side. The 12-inch drives are available with jukebox-type autochangers which can load a selected disc and put it on-line within several seconds. This type of technology is useful for archiving data and storing images. For example, applications that are very paper intensive or that use microfilm or microfiche might be good candidates for WORM technology optical disc drives.

Erasable

The third type of optical technology is called erasable optical. Disc drives incorporating this technology are most similar to magnetic discs in that they can read, write and erase information. There are currently three different kinds of technology being researched for erasable optical discs. The first type (and the one that is closest to market) is called **magneto-optical** or **MO**. As the name implies, it utilizes both magnetic and optical effects. A laser is used to heat up a point on the optical media and when the spot is heated above the Curie point, it becomes susceptible to a magnetic field (lowered coercivity) positioned underneath the disc platter, which can then re-orient the magnetic domains. When the laser is turned off, the material in the optical media is "frozen" in its new state, insensitive to the magnetic field (higher coercivity). As you can see, in this technology, no permanent pits or bumps are etched on the disc, making the disc reusable.

In a magneto-optic system, the data must be erased before it can be rewritten. This requires an extra disc revolution before new data can be recorded. Consequently, this type of disc drive takes a bit longer to write data (at least with first generation technology), yielding performance somewhat slower than magnetic discs.

A second type of erasable disc drive is based on a technology called **phase change**, where information is stored on discs made from materials

that convert from a crystalline state to an amorphous state with the application of laser light. Phase change technology is currently in the laboratory, and is further away from product integration than MO.

The third type of erasable technology is called **dye polymer**, where a special media is used consisting of organic dye layers, each of which absorbs laser light of different frequencies. Information is stored in deformations of the polymer layer. This type of technology is also currently in the laboratory stage of development.

The erasable technology that is closest to producing a product is magneto-optic technology. First generation MO devices will be slower than magnetic discs, however, future generation MO-type devices will be faster and will approach the access speeds of today's Winchester disc drives. In summary, erasable optical disc drives will initially complement magnetic disc drives. In the future, there is potential for erasable optical disc drives to replace magnetic disc drives in certain applications.

TIMING

If all this sounds fascinating, the next question might be "when?". CD-ROM and other read-only disc products have been out for a number of years now and are shipping in volume. Similarly, write-once drives are currently being shipped in production volumes as well. I think that 1988 will be the year for shipments of erasable-optical disc drives, if all goes well. The technology is ready now, the infrastructure is in place, and optical recording is real. Bring on the applications!

USER NEEDS/OPTICAL FIT

I have categorized mass storage user needs into five general sections as shown on the attached matrix. The first section is called "primary on-line storage," and is characterized by fast and frequent access, high transfer rates, and changing data. Welcome to the exclusive domain of the high-speed Winchester disc drive!

The next category is called "secondary on-line storage" which is characterized by slower access requirements, perhaps to store data that migrates from primary, infrequent access, and no operator intervention. Secondary on-line storage requires a lower cost per megabyte than primary on-line storage, and thus is better suited for archival data that needs to be kept on-line.

The third category of storage is referred to as "off-line archival." This is the realm of magnetic tape, characterized by removability, stable data, large capacities, and low media cost.

The fourth category is "backup," where the requirements are for removability, fast access, high transfer rate, large capacity, a

"verify" capability, and low media cost. In an ideal environment, backup would be invisible and transparent to an operator.

The last category is "data exchange," e.g., data movement from system to system, user to user, or software distribution. Some of the characteristics are removability, standard media, large capacity, high transfer rate, and low media cost.

Mass Storage User Needs

	<u>Primary On-Line</u>	<u>Secondary On-Line</u>	<u>Archival</u>	<u>Backup</u>	<u>Data Exchange</u>
User Needs	Fast Access High Xfer Frequent Access Changing	Slower Access Migrate from Primary Infrequent Access No Operator Intervention Low Cost/MB	Removable Off-Line Stable Data Lg. Capacity Low Media Cost	Removable Fast Access High Xfer Lg Capacity Verify Low Media Cost *Invisible*	Removable Std. Media Lg. Capacity High Xfer Low Media Cost
Applications	Temporary Index Programs Data Bases	Batch Programs Journals Logging Data Bases	Legal Records Reports History Security	Partial Full	System to System User to User S/W Distribution

main
spad
087

Now, where does optical fit into these mass storage areas? The first generation magneto-optic devices will probably be too slow to replace primary on-line storage devices. Secondary on-line, on the other hand, might be a good fit because access time is not as important and optical devices are certainly inexpensive when comparing dollars per megabyte.

Given that the forte of optical discs is high capacity removable media, we need to pursue these implications a bit further. With a removable media disc drive, user storage can be expanded modularly on an "as needed" basis through purchase of additional media modules. In the case of optical disc drives, since the areal density of the media is so high,

Optical Solutions To Mass Storage User Needs

incremental storage can be added for less than thirty cents per megabyte! Compare this to adding more fixed disc magnetic storage which costs a hundred times more on a per megabyte basis.

As a result, those applications which are characterized by high growth in storage are good candidates for optical. High security applications that require a "lock up" of sensitive or classified data require high capacity removable media. Similarly, there's a fit in Computer Aided Design (CAD) workstations that are shared by a number of designers, each of whom store their own drawings on removable media, carry them from station to station, and modify them as required. These media can later be inserted into NC machines for actual parts fabrication. Data collection and data logging jobs are other examples of applications that may be a good match for optical mass storage devices. Electronic publishing applications continue to expand storage requirements for information systems.

Another potential application for optical may be in backup, where on-line discs need to be copied regularly and quickly to high capacity, small size, low cost media for storage and potential recovery. Optical media is removable, and there will be some displacement of magnetic tape in backup applications. An added benefit is that file recovery from a random access optical disc is much faster than tape.

In addition, the large write-once drives are ideal for archiving applications where legal issues require storage of data for years. Jukeboxes of optical discs for image and data storage can conveniently replace paper, microfilm, and magnetic tapes at a low cost. In the future, with higher performance erasable drives, optical may well move into primary on-line applications, particularly for smaller systems where performance is not as critical. Finally CD-ROM's can be useful for wide distribution of data and reference storage, such as manuals.

In summary, the optimal optical disc applications fall somewhere in between the mass storage applications spectrum, bounded by magnetic discs and magnetic tape. As such, users like yourselves will determine the best fit of the technologies available to meet your needs.

MARKET ISSUES

A number of issues need to be discussed that are highly relevant to optical mass storage in the world of information processing.

Standards

The first issue is that of standards. A standard media format is essential to facilitate data exchange among systems, since virtually no one wants to be locked into a non-standard type of storage, particularly if it is removable. Therefore, market acceptance of a non-standard media is very difficult to achieve. The American National Standards Institute (ANSI) currently has committees whose members are defining and

debating various formats to be used for standard optical media for WORM and also for magneto-optics. The benefit to the user of a standard media format is that it can be read on a multitude of machines, not just that of one particular vendor. Another obvious benefit is lower cost, for if everyone adopts the standard, production runs become larger and cost decreases attributable to quantities of scale result. Standards are very important for optical mass storage, and significant progress is being made in this area.

System Integration

Integrating an optical disc drive into a computer system is a job that may be quite complex. This is particularly true of WORM, where the non-erasability of the media causes problems that must be dealt with in complex software routines. All versions of a unique file remain on the disc and pointers must chain to the latest version. This may cause performance degradation, particularly if changes are frequent and consume a lot of storage space. If the directory is stored on the WORM, each change requires a directory rewrite, which consumes a lot of time as well as media space. These are all issues that are being addressed by system integrators of WORM drives to hide the write once nature of the disc from the operating system.

For erasable drives, the integration task is much easier, since the optical disc can be erased and updated much like a magnetic disc. Only the latest version of a file is kept and only one directory is stored. Very little in the way of operating system or file management software needs to be changed. Device drivers, of course, need to be developed for all new mass storage devices.

Speed

First generation optical devices are best used in applications that do not require the quick access times characteristic of new Winchester disc drives. As we discussed earlier, secondary storage and on-line archiving applications do not require this high speed data access, and may be a better fit for first generation optical products. Be realistic in your choice of application, and bear in mind that optical devices are, in fact, slower than most Winchesters, although future enhancements and technology advances may allow optical to achieve comparable access times.

Acceptance

In addition to the benefits of high capacity, removability, and low dollars per megabyte, optical drives have a certain sex appeal attributable to the technology. They use semiconductor diode lasers, coated lenses, flat glass, and the media reflects light to display a full spectrum of colors. To what extent will optical achieve market acceptance? This depends entirely on how well the optical discs fill user needs for mass storage in a reliable and cost-effective manner.

How much are users going to take advantage of the removability feature, particularly since most large capacity removable drives are becoming scarcer in today's environment? If the market accepts optical mass storage devices, future technology will continue to improve them and they will offer strong competition to magnetic discs.

HEWLETT-PACKARD DIRECTIONS

Hewlett-Packard's goals are first and foremost to become an early leader in the field of erasable magneto-optic technology. Several projects are underway to develop 5.25-inch magneto-optic disc drives. We are also investigating an autochanger device to cycle through optical media for on-line archival applications that consume many gigabytes of data storage. The media used in both of these first generation devices will be interchangeable and will be of a standard format. Two standards are currently being debated in the ANSI X3B11 Committee, one being a continuous composite servo format, the other using a sampled servo. At this time, the continuous composite standard appears to be the front runner. It defines a capacity of approximately 300 Mbytes per side of optical media, for a total of 600 Mbytes per 5.25-inch removable cartridge. Access times will be comparable to floppy discs. The standard optical recording industry joke is that optical has always been two years away from introduction. Hopefully, MO devices will be introduced within that window. The first generation device will, generally speaking, complement magnetic discs, in applications similar to those mentioned previously.

Future efforts are aimed toward a higher performance, second generation magneto-optic type optical disc. As discussed earlier, these second generation drives may find applications in which they displace magnetic discs and enjoy a wider market appeal.

SUMMARY

We have taken a brief look at optical mass storage devices, including advantages and disadvantages. High capacity, removability, and low cost per megabyte are the distinguishing characteristics of optical mass storage. Optical mass storage applications fall in between those of primary on-line discs and magnetic tape. In addition, optical combines the best features of both, namely random access, removability, and low cost storage. First generation optical devices will complement existing mass storage devices, and later generations may very well erode into the magnetic disc market, as user acceptance of the technology expands.

Optical mass storage technology is not only fascinating but emotionally alluring, and will provide incremental capabilities not currently available to information processing. Be prepared to take advantage of the technology and incorporate it into your data processing applications soon, because optical mass storage is real, the infrastructure is developing quickly, and the future is nearly here.

MONITOR & CONTROL OF TEST SOFTWARE EXECUTING ON A REMOTE DISSIMILAR SYSTEM

**T. Josal, G. McNabb
Naval Undersea Warfare Engineering Station
Keyport, Washington 98345**

ABSTRACT

To fully test the functionality of a MK 50 torpedo it is necessary to completely disassemble it and monitor numerous signals while running test software. Presently, after being reassembled, the unit is subjected to only a brief go-no go test. Clearly there is an advantage in conducting a final test to verify the integrity of connectors mated during re-assembly.

In order to accomplish this, several programs were written which execute on the torpedo's on-board computer and run self tests on each of the microprocessor controlled electronics subassemblies. A monitor program, written in FORTRAN, executes on an HP 1000 and provides control over the loading, execution, and logging of data from each of these programs. This monitor program continually updates a display of selected memory locations of the torpedo's computer and writes to these locations as required to control the execution of tests.

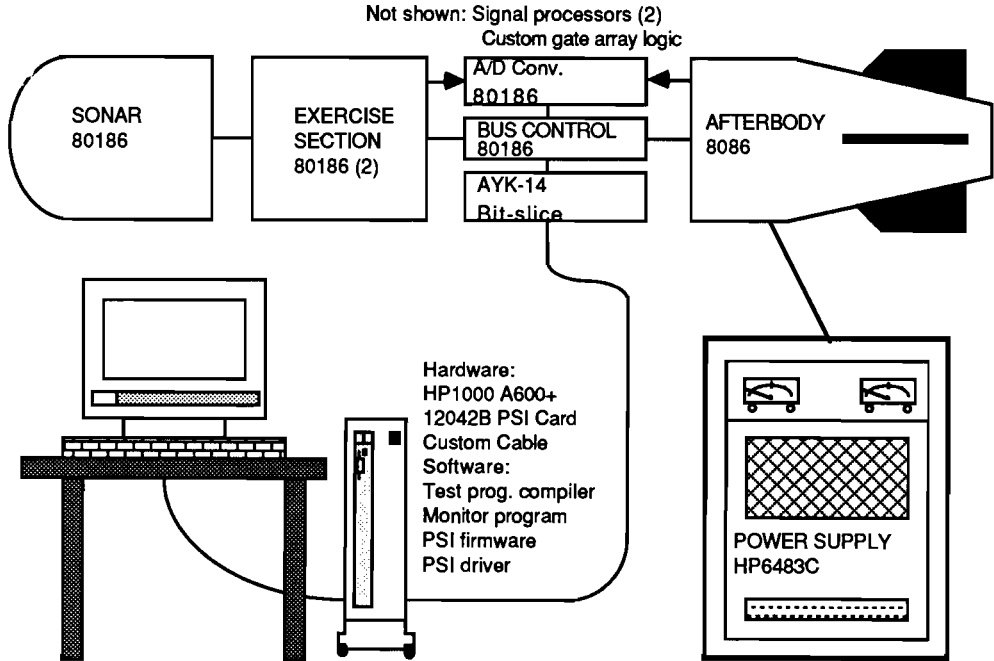
This paper briefly describes the mission of the Naval Undersea Warfare Engineering Station, the Navy's philosophy of separating weapons into functional assemblies to which faults can be quickly isolated, and the functional parts of the torpedo under test. The paper then focuses on the implementation of the communication link with the torpedo via the programmable serial interface card and the development of the monitor program itself.

INTRODUCTION

The Naval Undersea Warfare Engineering Station is the Navy's only repair depot for torpedoes and torpedo targets and as such is responsible for reliable, prompt, and cost effective test and repair of a wide variety of components. In order to facilitate maintenance, weapons are procured by the Navy based on the 'functional item replacement' concept, where the weapon is divided into easily replaceable sections, each of which is typically responsible for one particular type of function, i.e. sonar, guidance, etc. The latest generation of lightweight torpedo, the MK 50, follows this philosophy. In this case, each section contains an 80186 microprocessor to interface to the torpedo's communications bus and control functions within the section, while the main computer is executing tactical software. For test purposes, tactical software can be replaced by specially written test software to interrogate the various sections and determine their operational status. The present systems test scenario consists of automatic test equipment (ATE) providing stimulus and taking measurements while controlling test software executing within the torpedo. The ATE is HP 1000 based and controls the instrumentation as well as test software,

however it requires complete disassembly of the unit under test. This paper describes a stand-alone HP 1000 based system adapted from the ATE that tests the unit completely assembled to establish the integrity of connectors mated following system test. Due to the scope of this project, there have been many people in different groups involved in the development of the system, particularly Honeywell, Inc. personnel. As a result, detailed technical discussion in any one area is beyond the scope of this document. I cover the monitor in the most detail but also relate an overview of the complete system since the basic scenario has many potential applications.

TEST SCENARIO



The figure above shows the elements of the system. The unit under test utilizes an AYK-14, manufactured by Control Data Corp., as the command and control computer. CDC also manufactures the standard support equipment used to talk to the AYK via its 150Kbaud serial port, the Computer Control Unit (CCU). The CCU is an 8080 based smart terminal used to monitor execution of AYK software. An 80186 microprocessor in each section is dedicated to controlling the functions of that section and interfacing to the communications bus. As a result, test software modules can be written to command each section to execute various self-tests. The system must monitor and control loading and execution of these tests. The requirements of the scenario described here dictate the use of hard disk storage and fast execution time which would

indicate that an alternative to the CCU is necessary. The system ATE is such an alternate system using an HP 1000 with a PSI card whose ROMs are programmed to emulate CCU software. This retains the investment in knowledge of personnel already experienced in using the CCU, while allowing the potential for expansion and automation. The 1000 system used here replaces a CCU at 1/3 the cost. The system uses an A600+ with 15Mbyte winchester and 270kbyte floppy internal drives. Besides the PSI card itself, the only additional hardware requirement is a special cable to interface the PSI card to the AYK. Software requirements include a driver for the PSI card, a compiler to generate the test programs which will run in the AYK, and the monitor program. A test executive "interprets" the test programs in the AYK.

THE 12042B PSI CARD

The programmable serial interface card is nothing more than a High level Data Link Control (HDLC) interface card (for implementing a DS link over modems) without the HDLC firmware. It provides a 1000 with the ability to support a communications link based on a protocol residing in custom PROM firmware. The card has five 4Mhz Z80A microprocessors, two ROM sockets for custom code of up to 8K bytes per socket, plus 16Kbytes of RAM. HP recommends the 24602A PSI firmware development package for end-users designing their own communications products. This consists of a debug monitor, cable, and manuals. The monitor allows for loading of assembled code into PSI card RAM along with most of the other features you would expect a monitor to have. The code for this application was developed using an HP64000 development station. The absolute file created with the 64000 is transferred to the 1000 using an RS-232 link from the 64000's modem port to a mux port on the 1000. The 64000 has a terminal emulator capability which allows the user to log on to the 1000, open a file from EDIT, and initiate transfer with 'upload filename:absolute'. As an alternative, the code could be written on a 1000 and cross assembled to Z80 code. HP also markets a 'multiuse' PSI card (12043A) which uses RAM to store the user developed code rather than EPROMs. It comes with an 8K EPROM which contains a monitor and self test, loading, and executing code. This product is fully supported by HP as shipped since it is not modified by the user in any way. The 12042B can be supported but only as part of a system (at no cost) according to HP; hardware maintenance for the card alone is not available. The PSI code developed is basically a subset of the software used by the CCU to interface with the AYK-14. The size of the code has now grown to approximately 20Kbytes, greater than the 16Kbyte ROM limit mentioned earlier. This necessitated moving some of the code to RAM on the card, and writing a program to load the code from a file on the 1000 to RAM space on the PSI card. The welcome file has been modified to run this program each time the system is booted up to insure that all of the code resides in the PSI card at all times. Since the checksum resides in EPROM memory, modifications to any of the code requires burning new EPROMs. The cable from the AYK to the PSI card edge includes a 2.4Mhz oscillator to provide the send and receive timing required. PSI interface circuitry divides by 16 resulting in the required 150K clock. The data signals are interfaced to RS-422 differential line



drivers.

TEST PROGRAM COMPILER

Test programs which exercise various functions of the torpedo are created using a high level compiler specifically written for this application in FORTRAN. The compiler was developed to allow the user to create readable programs which can be easily modified. The resulting test programs simulate aspects of in-water demands on the communications bus, the subsystem interface, and each of the subsystems. The following example of test program source code is shown here to illustrate the specific means of implementing communication with the monitor program running on the 1000. This program moves the fins to 12 degrees and checks the commanded value against the analog position read by the fin potentiometer and converted to a digital word:

```
DEFINITIONS;
USER_INPUT      RESERVE 2 WORDS;      {1802 - 1803 operator input}
TESTID          RESERVE 1 WORD;       {1804 current test id number}
TEST_STATUS     RESERVE 1 WORD;       {1805 current pass/fail status}
DATATYPE        RESERVE 1 WORD;       {1806 data type}
MEAS_VALUE      RESERVE 2 WORDS;      {1807 - 1808 measured value}
HIGHLIMIT       RESERVE 2 WORDS;      {1809 - 180A upper limit}
LOWLIMIT        RESERVE 2 WORDS;      {180B - 180C lower limit}
CONDITION       RESERVE 1 WORD;       {180D overall pass/fail status}
CONTINUE        RESERVE 1 WORD;       {180E monitor control flag}
. . .
MAIN PROCESS;
. . .
IO USING FIN_COMMAND      {command fins to move to 12 degrees}
  FROM AYK BUFFER FIN_12DEGREES
  TO CME;
DELAY 3 SECS;              {wait until fins stabilize}

IO USING READ_A/D_DATA    {read the a/d data which will contain the}
  FROM CME TO AYK         {present position of the fins}

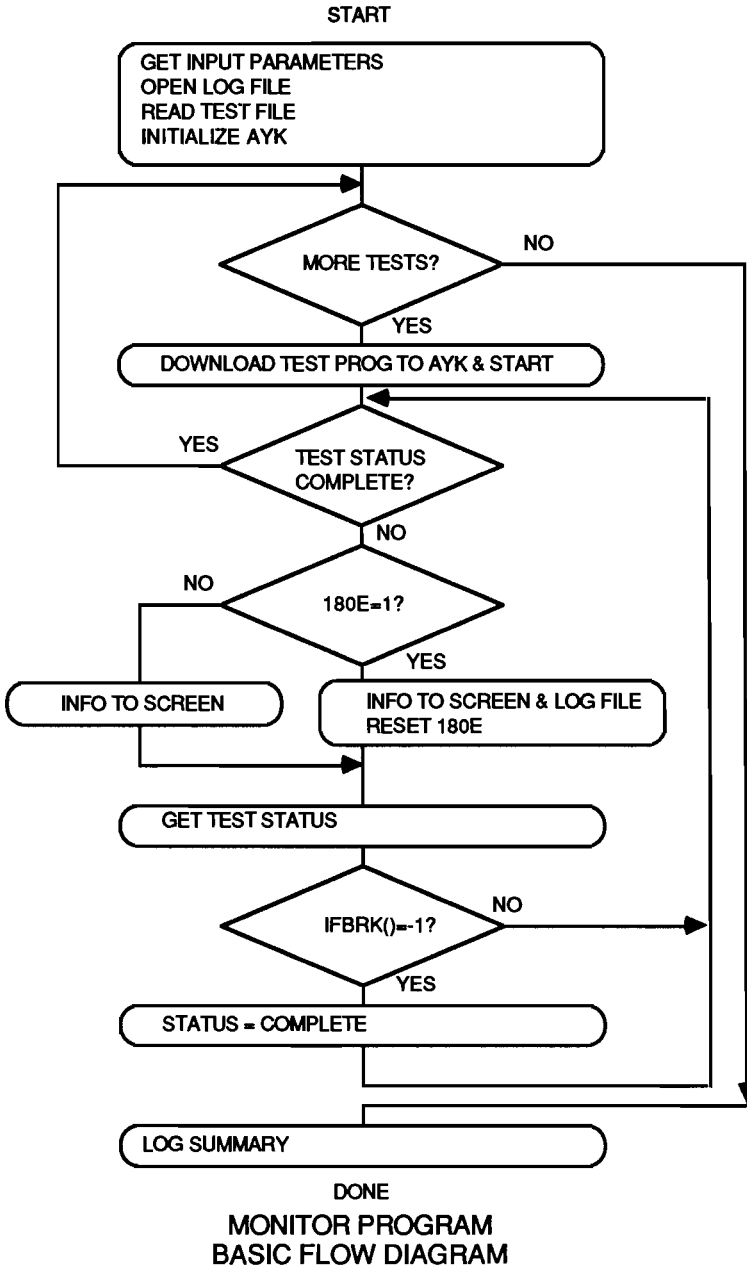
{the a/d data just obtained is compared against expected values here and the results
passed to the monitor using status words reserved in the above definitions section}

CALL WAIT_TO_CONTINUE;    {loop until monitor resets 180E}
. . .
SUBROUTINE WAIT_TO_CONTINUE;
  CONTINUE = 1;
DONT_CONTINUE:
  IF CONTINUE = 1 THEN
    GOTO DONT_CONTINUE;
  END IF;
END SUBROUTINE WAIT_TO_CONTINUE;
. . .
END MAIN PROCESS
```

Note that the definitions section allocates specific memory locations for user defined variables beginning at 1802h. These are the locations that are displayed by the monitor and return pass/fail status information. Of special interest is CONTINUE which is used to 'pause' execution of the test program until the monitor can process the information it has and 'catch up'.

MONITOR PROGRAM

The following flowchart illustrates the basic functions of the monitor program:



The program begins by checking for valid run string parameters, opening a new log file with a unique name, reading the testname file, and verifying PSI card communications while initializing the AYK. The testnames file is a type 4 file created with EDIT that provides the monitor with a listing of all previously compiled tests which are to be run. They will be executed in the sequence in which they appear in this file. A sample file is shown below:

* testname	on pass run...	on fail run...
TEST1	TEST2	DIAG1
TEST2	TEST3	DIAG2
TEST3	TEST4	DIAG1
TEST4	STOP	DIAG1
DIAG1	DIAG2	DIAG2
DIAG2	STOP	STOP

In addition to the testname, the monitor looks for two more fields: the first is the test to be executed if the test passes, the second the test to run if a failure occurs.

The program then sequences through the testname file, loading and running each until it has completed. While a test program is executing, the monitor continually reads PSI RAM and updates the crt display. Whenever 180E=1, indicating that the test has completed a section and is looping, the monitor logs the data and resets the variable to allow execution to continue.

The monitor program relies on just a few simple subroutines to handle communications with the AYK. CCUCOM accepts as parameters the CCU command you wish to send (CH_STRING) and it's length in chars (LENGTH). This subroutine emulates the operator typing in a command on the CCU console. A check for the ABort command is made before passing the string to the AYK via the EXEC call.

```
SUBROUTINE CCUCOM (CH_STRING, LENGTH)
```

```
 . . .
```

```
*space fill command string buffer so that no nulls are passed to CCU
WRITE(COMSTRING, '(50x)')
```

```
*pass the string to CCU
```

```
*IWRITE specifies write to PSI card
```

```
*ISELECT specifies type of EXEC call and LU of card
```

```
*ICOM contains the string to be sent
```

```
*ILEN is the number of chars on the string
```

```
*INUM is the number of the PSI command sent
```

```
*IWORDS is the number of words in the PSI command
```

```
IWRITE = 100002b
```

```
ISELECT = 110011b
```

```
NUM = 3
```

```

IWORDS = 1

IF (COMSTRING(1:1) .EQ. 'A' .AND. COMSTRING(2:2) .EQ. 'B') THEN
  ILEN = 0 !length is zero
  INUM = 2 !abort command
END IF

CALL EXEC (IWRITE, ISELECT, ICOM, ILEN, INUM, IWORDS)

END ! of ccucom

```

Typical command strings to the AYK include MC-. (master clear) ST-1. (execute self-test) and MD-1802. (display 16 memory locations starting with 1802h). The period is required as the command terminator. The following would be the resulting CCU screen for the command string MD-1802. (recall that 1802h is the starting address of user defined variables):

						079 000 CN1 P EC5	
				STOP			
P	0000	<00>		CPU	MEMORY		
(P)	857F	41 1 07 17	8100	1802	857F	8100	
AD	0001	<00>	8100	1804	8280	00C0	
S1	0000		8100 40 1 00 00	1806	8102	0000	
S2	0000			1808	080C	0000	
B-P	0000			180A	81F9	FC14	
B-O	0000			180C	016C	E616	
LLJ	0000			180E	0158	0177	
RTC	0000			1810	080C	080E	

CCU MEMORY DISPLAY

In other words, a CALL CCUCOM('MD-1802.'8) will result in the MD-1802. command being passed to the AYK just as if it originated from a CCU. The AYK will then act on the command and send the corresponding display to the PSI card, where it is buffered in RAM. The AYK continually updates the information, as if it were refreshing the screen of a CCU. The memory locations displayed correspond to the user variables declared in the definitions section of the test program, as was shown earlier. As the test program modifies these variables, the resulting value in PSI RAM is also modified. The monitor program then gets the data from PSI RAM as needed via the subroutine PAINT. The subroutine PAINT contains the EXEC READ call (equivalent to the EXEC WRITE in CCUCOM) which retrieves the CCU screen from PSI card RAM. PAINT returns the contents of PSI RAM into a memory buffer where it can now be examined and tested against limits and used to update screen values. The memory buffer was declared as local common in an include file.

```

SUBROUTINE PAINT

```

```

. . .

```

```

* space fill the buffer
  WRITE (BUFFER<'(1024x)')
*

```

```

* IREAD specifies read from PSI card
* ISELECT specifies type of EXEC call and LU of card
* IBUF buffer to contain the returned text
* ILEN max size of the buffer
* INUM number of the PSI command sent
* IWORDS number of words in the PSI command
*
      IREAD = 000001b + 140000b  !140000b requests no-abort no-suspend error handling
      ISELECT = 110011b
      ILEN = 512
      INUM = 4
      IWORDS = 1
      CALL EXEC(IREAD, ISELECT, IBUF, ILEN, INUM, IWORDS, *777)

      GOTO 1000
777  CALL ABREG(IA, IB)
      WRITE(1, ' (A, A2, A2, A) ' )  'PAINT:  Error ', IA, IB, ' ; problem with PSI card'
1000  END !of paint

```

Now that we have the CCU screen in a 1024 character array, we can sort through it and pick out the information we need. Since we just executed a MD command, we would normally want the memory values found in the right two columns of the CCU display. The following subroutine GET_MEM picks out the 16 hex values and fills two arrays - one with the hex values and one with converted integer values.

```

SUBROUTINE GET_MEM
      . . .
* set initial conditions and find first three carriage returns
      CR=CHAR(13)
      IERR=0
      IPOS=0
      I=0
      J=1

      DO WHILE (I .LT. 3 .AND. J .LE. 82)
          IPOS=IPOS+1
          J=1
          DO WHILE (BUF(IPOS:IPOS) .NE. CR .AND. J .LE. 82)
              IPOS=IPOS+1
              J=J+1
          END DO ! while buf
          I=I+1
      END DO ! while i

*
* Now that third CR found, first memory value must be 52 positions to right
*
      IF (J.GT. 82) THEN
          IERR = -175
      ELSE
          IPOS = IPOS-5
          DO I = 1, 15, 2
              IPOS = IPOS+57 ! At beginning of left column of values
              HEX_ADDR(I) = BUF(IPOS:IPOS+3)
              CALL C_TO_I (HEX_ADDR(I), VALUES(I))
          *
          IPOS = IPOS+12 ! At beginning of right column of values

```



```

HEX_ADDR(S(I+1)) = BUF(IPOS:IPOS+3)
CALL C_TO_I (HEX_ADDR(S(I+1)),VALUES(I+1))
END DO ! i
END IF ! j
END ! OF GET_MEM

```

Also available from this CCU display is the status of the software currently executing - (STOP in this case). This can be obtained in a similar subroutine called STATUS. Instead of looking at the CCU screen locations containing memory values, we look specifically for that location where the CCU returns status information.

SYMBOLIC DEBUG / PROFILING

The monitor program appeared to be the slow link in the test process. The test program invariably completes the present section and is looping by the time the monitor returns to check the results. We would like the screen refreshed as fast as possible to track variables within a section. In addition, there is a time constraint due to heat buildup in the torpedo, so work has begun to speed execution of the monitor. Already relied upon for debugging purposes, the symbolic debugger's program profiler feature was used to try to pinpoint problem areas. Profiling is initiated from Debug with the Overview command. At this point, the program runs to completion while Debug samples the program counter every 10 milliseconds and logs the data it collects to a file. Naturally the system must be quiet while profiling to insure that the resulting histogram will represent only your program's execution time. The test sequence must be representative of a typical test; a series of many short programs would cause more data points to be taken in subroutines which load programs, for example. After profiling, the data points obtained are plotted with the Histogram command. The following is the histogram resulting from profiling the monitor program:

```

Debug Rev.2540 <870706.1513>
DEBUG> O MONITOR1.OVR
Overview> H
Processing profile data...

```

Routine	Amount	Histogram
-----	-----	-----
.FFCL/seg. 0	7%	*****
GETMEM/seg. 0	4%	**
.FFOP/seg. 0	2%	*
Other (known code)	3%	**
Other (unknown code)	11%	*****
I/O suspend	51%	*****
Waiting for a program	16%	*****

Clearly, the program is spending the majority of time in the I/O suspend state. In fact, only one of the monitor subroutines (GETMEM) consumed more than .5% of the total execution time. This would appear to leave little room for

improvement and certainly suggests that our assumption that a inefficient monitor program was the problem was wrong. Since it took a few seconds for the operator to respond to the terminal read, the program was modified to eliminate this then profiled again with the following results:

```
Debug Rev.2540 <870706.1513>
DEBUG> O MONITOR2.OVR
Overview> H
Processing profile data...
```

Routine	Amount	Histogram
-----	-----	-----
GETERR/seg.	3%	**
.FFOP/seg. 0	2%	*
Other (known code)	5%	***
Other (unknown code)	19%	*****
I/O suspend	51%	*****
Waiting for a program	16%	*****

The histogram above shows no significant change from the original, confirming what we expected and again only one monitor program shows up. If time spent waiting for user input were significant, it would have shown up on the first histogram as "class I/O suspend". The I/O suspend we see indicates I/O to an unbuffered device, such as a disk. We also know that although much time appears to be consumed writing data to the screen, it is insignificant since it would appear as "Buffer limits suspend". To attempt to determine were all of this I/O suspend was occurring, we again modified the program, this time eliminating all writes to the log file leaving no disk I/O. The following histogram resulted:

```
Debug Rev.2540 <870706.1513>
DEBUG> O MONITOR3.OVR
Overview> H
Processing profile data...
```

Routine	Amount	Histogram
-----	-----	-----
DISPLAY_INFO/seg. 0	5%	***
Other (known code)	3%	*
Other (unknown code)	20%	*****
I/O suspend	53%	*****
Waiting for a program	15%	*****

Time spent in the I/O suspend state actually *increased* slightly. Obviously, disk I/O was not a problem, leaving only the EXEC reads and writes to the PSI card itself as the cause of the I/O suspend state. Rather than monitor code that slowly searched through and modified arrays causing the problem, as it first appeared, the program actually spends more than half of the time waiting for the PSI card. For the present the speed will be satisfactory but it is still possible to optimize the PSI firmware or monitor code to speed execution. The Histogram defaults to plot all the subroutines in the program, as done above. Once this is done, another histogram can be plotted, this time using only the data points collected in the subroutine identified to be CPU bound in the first plot. As an example, we

ran a histogram of the only monitor subroutine which showed up in the above histogram (DISPLAY_INFO). This routine updates the screen and log file:

```
Overview> H DISPLAY_INFO
Profile for module DISPLAY_INFO
5% of total time spent here
Line #      Amount  Histogram
-----
702         13%   *****
737         5%     *****
741         3%     *****
743         2%     **
752         1%     **
762         13%   *****
770         23%   *****
771         9%     *****
774         3%     *****
775         10%  *****
777         6%     *****
```

Looking at the list file for DISPLAY_INFO reveals that lines 762, 770, and 775 are all logical IF statements and take 46% of the time spent in DISPLAY_INFO. However, since in this case only 5% of the total time is spent here there would be no point in optimizing this code.

MONITOR OUTPUT

Below is a typical display screen while the monitor is running:

```
BURT REVISION 1.1          <870324.1407>
Log File name: BURT0000.LOG      Parameters: None

BURT Started at: 12:21 PM THU., 26 MAR., 1987
Torpedo powered up at: 12:32 PM THU., 26 MAR., 1987

*****
Testname: TEST1.TST          Update time: Sat Jan 24, 1987
*****

Testname: TEST1.TST
Test ID: 0000 Test status: RUNNING
AYK MEMORY:
1802: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
180F: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
181C: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Elapsed torpedo time: 5 minutes 8 seconds
```

The header provides the operator with the name of the log file used for this test, the version of the monitor, the run string parameters, and the start time. Following the header is the name of the test currently executing in the AYK and the time stamp from the source code. Below this are the 39 memory locations tracked by the monitor program and the time elapsed from power-up. The corresponding log file resulting from a test consists of a series of screens, each one representing the state of test software following completion of a section (180Eh=1):

BURT REVISION 1.1 <870324.1407>
Log File name: BURT0000.LOG Parameters: None

BURT Started at: 12:21 PM THU., 26 MAR., 1987
Torpedo powered up at: 12:32 PM THU., 26 MAR., 1987

Testname: TEST1.TST Update time: Sat Jan 24, 1987

Testname: TEST1.TST
Test ID: 0000 Test status: RUNNING
AYK MEMORY:
1802: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
180F: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
181C: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Elapsed torpedo time: 5 minutes 8 seconds

FINAL AYK MEMORY:
1802: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
180F: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
181C: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Elapsed torpedo time: 5 minutes 11 seconds
** WARNING 10; Test terminated abnormally

Testname: TEST2.TST Update time: Mon Jan 26, 1987

Testname: TEST2.TST
Test ID: 0000 Test status: RUNNING
AYK MEMORY:
1802: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
180F: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
181C: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Elapsed torpedo time: 9 minutes 11 seconds

Testname: TEST2.TST
Test ID: 0000 Test status: RUNNING
AYK MEMORY:
1802: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
180F: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
181C: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Elapsed torpedo time: 10 minutes 27 seconds

FINAL AYK MEMORY:
1802: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
180F: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
181C: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Elapsed torpedo time: 10 minutes 29 seconds
** WARNING 10; Test terminated abnormally

BURT Finished at 12:28 PM THU., 26 MAR., 1987
TOTAL ELAPSED TORPEDO TIME: 0 Hours 6 Minutes 49 Seconds
** BURT FAILED **

TYPICAL LOG FILE

SUMMARY

The PSI card provides a relatively fast way to create a customized 1000 based system to meet almost any requirement. By designing firmware unique to a particular application, the user can create an almost unlimited number of customized products. As was seen, the program which utilizes the PSI card for I/O can be quite simple and, by customizing the PSI card firmware, can retain a protocol already familiar to programmers, reducing development time. Any application requiring serial communications and where speed and flexibility become a limitation is a candidate for replacement by a similar system. As seen here, cost can actually be significantly reduced while realizing the advantages an HP1000 computer environment can provide. The expertise required by a company to develop such a customized system can be considerable, however. The ability to write Z80A assembly code for the PSI firmware, HP1000 assembly code for the PSI driver, code for the monitor program (FORTRAN), and, of course, technical knowledge of the target system are all required.

There are many possibilities for future enhancement of this application. Experienced engineers might attempt to create tests which diagnose specific areas and structure the order to isolate faults when possible, in effect building-in simple expert system capabilities. Additionally, by adding a few HP1B instruments, the scope of the test could be expanded.

I would like to acknowledge the work of Craig Gibson who wrote the first version of the monitor including all of the CCU routines.



A Modular Multiprocessor Simulation Philosophy

A. D. Gregg and C. M. H. Kimpke
City Computing Ltd.
Surrey House
Surrey Street
Croydon, U.K.

P. J. Webb
Behavioral Science Division
Admiralty Research Establishment
Queens Road
Teddington, U.K.

Introduction

The prime function of the Behavioural Science Division of ARE is to identify and attempt to solve the human factors related problems of future naval computer based systems. This covers a wide range of operational requirements from Command and Control systems to Sonar detection and classification problems. The application work is therefore focused on the development of hardware and software tools to give improved man-machine interaction.

The method used to investigate and assess the likely performance (in MMI terms) of these new systems has been to build simulators of proposed systems using advanced forms of interactive display technology and data processing techniques. Royal Navy personnel have then been used, in a series of controlled experiments, to assess their performance on the new system.

Design of any experimental simulator using advanced forms of technology is, at best, a speculative task. Researchers are often unwilling, indeed unable, to provide a detailed specification of the overall requirement. Thus much of the final design is dependent on the results of pilot studies. This means these simulators must be able to accommodate many design changes and provide a high degree of flexibility in interfacing to other systems.

The Behavioural Science Division, like most research organisations, has limited funds and must maximise its use of available resources. Development and manufacture of large purpose built simulators is unrealistic. So general purpose commercial processing and display equipment, which can be employed for a wide range of applications, are used where possible. By adopting a modular approach it has been possible to develop a series of building blocks with which to attack most simulation problems.

Hardware

The computers chosen to perform the number crunching and control of these simulators is the Hewlett-Packard 1000 A-Series, and particularly the A900 (or micro29) for their processing power. The A900 CPU's performance is 3 MIPS, almost 1 MFLOPS, and can support 24 Mbytes of memory. It can also be microprogrammed if necessary.

These machines are arranged in a point-to-point network. Links can be put between any computers. Usually these links are high performance Parallel Interface Cards (PIC), running at about 0.7 Mbytes per second. Slower links can be used if throughput is low. These loosely coupled processor networks provide much flexibility and expandability.

High performance detailed graphics is provided by a SIGMEX ARGS. These machines are connected to the A900's by PIC's. Lower performance graphics is provided by HP Graphics cards, which are plugged into the A900's card cage.

Software Design

The design should be seen as two levels of structuring. Firstly, the simulation is composed of many separately identifiable program modules which communicate with each other. Secondly, the individual program modules are themselves of a highly structured design and implementation.

A program would, for example, draw a display or log operator interactions. Typically there would be a dozen programs in a system, all running concurrently. Each program has one (and only one) message queue. Tasks can suspend while waiting for a message, so that they use no CPU time. Because messages are queued, there is no need for a program sending a message to 'meet' the receiver - it sends the messages then continues with the rest of its assignment.

In most programs messages are picked up at a single point (or procedure). Each message consists of a 'Key' and a data buffer. The key and buffer, and possibly the program's internal state determine the purpose of the message, and its resulting actions and/or messages to other programs. One should never assume that a set of messages will be contiguous. (Arguably, if they must be then they are really one large message). A message will cause a program to do something eg. update a display or access a database, and possibly issue messages to other programs.

This setup has many advantages:

(a) programs are truly 'black boxes' - if they receive the right messages, they will behave appropriately. So they can often be tested independently of the rest of the system.

(b) it is easier to trace bugs. Actions are always caused by messages, so the flow of control always comes from the pick up point.

(c) there are none of the mysterious interference effects typical of system common communication.

(d) messages are queued, so a program does not have to 'meet' another to send it a message.

(e) usually it is possible to add new messages from other programs without having to modify existing message sequences.

(f) it is easy to replace and move programs - all that matters is that they correctly interpret messages they receive, send out messages expected, and access their program-specific peripherals (eg graphics computer).

(g) programs can easily be designed to survive bad messages. And if they do not, it is often possible to

restart them. Robustness and recoverability are very important to AXB because simulators must run for long periods to get useful results.

There are also shared data areas, called pools. These have very strict access procedures to avoid the problems typical of system common communication. It must be emphasized that there are usually very few pools in a system - most communication is done by message. Also, updating a pool cannot cause programs to perform actions, so it is essentially a static element.

Source code in the simulators is written mainly in Pascal. Pascal code is highly structured, and has extensive type, parameter list, and range checking. More importantly, Pascal allows you to build and easily access highly sophisticated data structures. Initial code production usually takes longer with Pascal than other non-structured languages, but debugging time is much reduced, and the software produced is much more maintainable and alterable. Whenever possible standard graphics languages such as GKS or HP DGL are used.

When absolutely necessary assembler or microcode might be used - it is very easy to interface HP Pascal/10000 to these languages.

Tasks perform message transfers by making calls to library routines. These libraries were written as HP Pascal/1000 'Modules' - using these Pascal/1000 'Modules', we make the library routines and data types available to Pascal/1000 source code, so that they look like an extension of the language. This gives all the time saving benefits of Pascal type, parameter list, and range checking. (The libraries can also be used from Fortran and Assembler, but not so conveniently.)

Software Environment

The Program Communication System (PCS) sets up the programs for the simulation, and establishes message queues. Tasks address other programs by name, so do not need to know on which node of the network they are. Assignment of programs to nodes is read from a file at start up, so can be changed easily and quickly. Note that the PCS user does not need to know the structure of the network (though this will obviously have some bearing on performance). No software changes are needed to change the node on which a program runs.

As part of the PCS environment, there is an Interactive debugger Program (IP). This allows the user to interact with the system from keyboard or command file. He can inject messages, inspect message queues, reschedule crashed

programs, and perform other control operations. IP has facilities for building and inspecting user defined message structures, and for reading these messages from file. There is also a simple program which will wait for messages and print them out.

PCS acts through a custom built networking package, which makes the network look like a single virtual machine. This provides creation of message queues; transparent message passing from node to node (through intermediary nodes if necessary); and remote program scheduling. The network configuration is read from a text file, so it is easy to add or remove machines and links. The network package accesses the link drivers through standard EXEC calls, so it does not matter what types of links are used (eg PIC, serial etc).

In order to give access to the full speed of the links, it provides the capability to lock the lines for fast large transfers. This feature, not normally used by a PCS user, is available through the network library. The Network Library is a HP1000 Pascal Module like the PCS Library.

The network also provides remote file transfer & listing, and virtual terminal facilities.

Curtis TADS Simulator

The Curtis Towed Array Display Simulator is a real time simulation of a passive narrowband sonar system. A realistic model of an ocean environment containing many vessels is used to generate sonar signals representative of real vessels in a real life scenario. A mathematical model of the ship borne sonar processing systems is used to simulate the attributes of the real life sonar processing system and its array of hydrophones. The sonar signals are displayed using high resolution raster graphics.

The sonar operators are able to interact, in real time, with the various displays using a number of interactive devices. eg Tracker Ball for cursor and harmonic divider measurements, and switches for the selection of sonar processing options and display configuration.

The scenario is under the control of an experimenter, who may if he wishes take full control of the scenario in real time. Normally, however, the simulation runs a 'Log' of scenario events which determine the operating characteristics of various vessels in the scenario. Changes in the operating characteristics (eg. speed change) will cause changes in the sonar signals, thereby influencing the sonar picture.

As the simulator can produce controlled and repeatable sonar environments, it is ideal for studying operator target detection and classification performance. Special scenarios of interest can be fabricated where equivalent real life situations are either impossible to achieve or very expensive; for example, operations within a large task force would require massive organisation and the cost could be prohibitive.

Another role of this simulator is to investigate new ideas in data display and operator aids. The advantage of performing as much work as possible with the help of simulators is again cost. Changes to a simulator are much less expensive than changes to many in service sonar sets.

Implementation of TADS

The programs and data flows in TADS are indicated on figure 1. Most of the data flows from left to right, but there are some messages going the opposite way.

The simulation program follows the scenario selected by the controller. It keeps the Scenario Pool updated with the latest positions of vessels in the scenario. Every few seconds it sends a tick to the Broadband to start the sonar processing.

The kernel of TADS is the Broadband, Display and Narrowband programs. Broadband does the initial processing of the scenario information and passes the processed information to the Narrowbands. The Narrowbands produce pictures ready for display and send them to Display. Display stores display data, updates displays, and draws displays on operator request. The Narrowbands are exceptionally heavy on processing, so there are two identical Narrowband programs to distribute the load through the network.

Operators use TADS through switch panels monitored by the Tote program. Tote sends messages to Display to select display options, and to the Narrowbands to select processing options.

The controller interacts through the controller's station using touch sensitive interaction on a series of tote pages. The controller program sends messages to the simulation program to control and modify the scenario.

The simulation used to run on three HP A900s and a SIGMA ARGS, but has recently been upgraded to four A900s and two ARGS. The distribution of programs is shown in figure 2.

Additions to TADS

Using these design considerations, it is very easy to add programs, or even new subsystems. AXB have recently added a Direct Voice Input system to TADS. This was developed on one machine in isolation, but using PCS. It contained three programs, and issued messages to a 'dummy' program corresponding to the Tote program. When testing was complete, it took very little effort to add DVI to TADS, despite the fact that the DVI programs had to be re-distributed over separate computers in the final system.

Conclusions

Simulators built at AXB have shown that commercially available general purpose hardware is suitable for the production of very sophisticated simulators. Processing speed and the ability to link computers together and interface them to peripheral equipment are very important in the choice of hardware.

By following the software design philosophy using the concept of the Virtual machine under the PCS, we have been able to build systems / simulators which are very modular. This high level of modularity gives good fault tracing ability and enhances the maintainability. Structured languages such as Pascal, of which PCS looks like an extension, has numerous advantages over more traditional simulation languages. Only essential parts, required to achieve the required performance, need be written in assembler or even microcode.

c Controller, Her Majesty's Stationery Office, London,
1986.

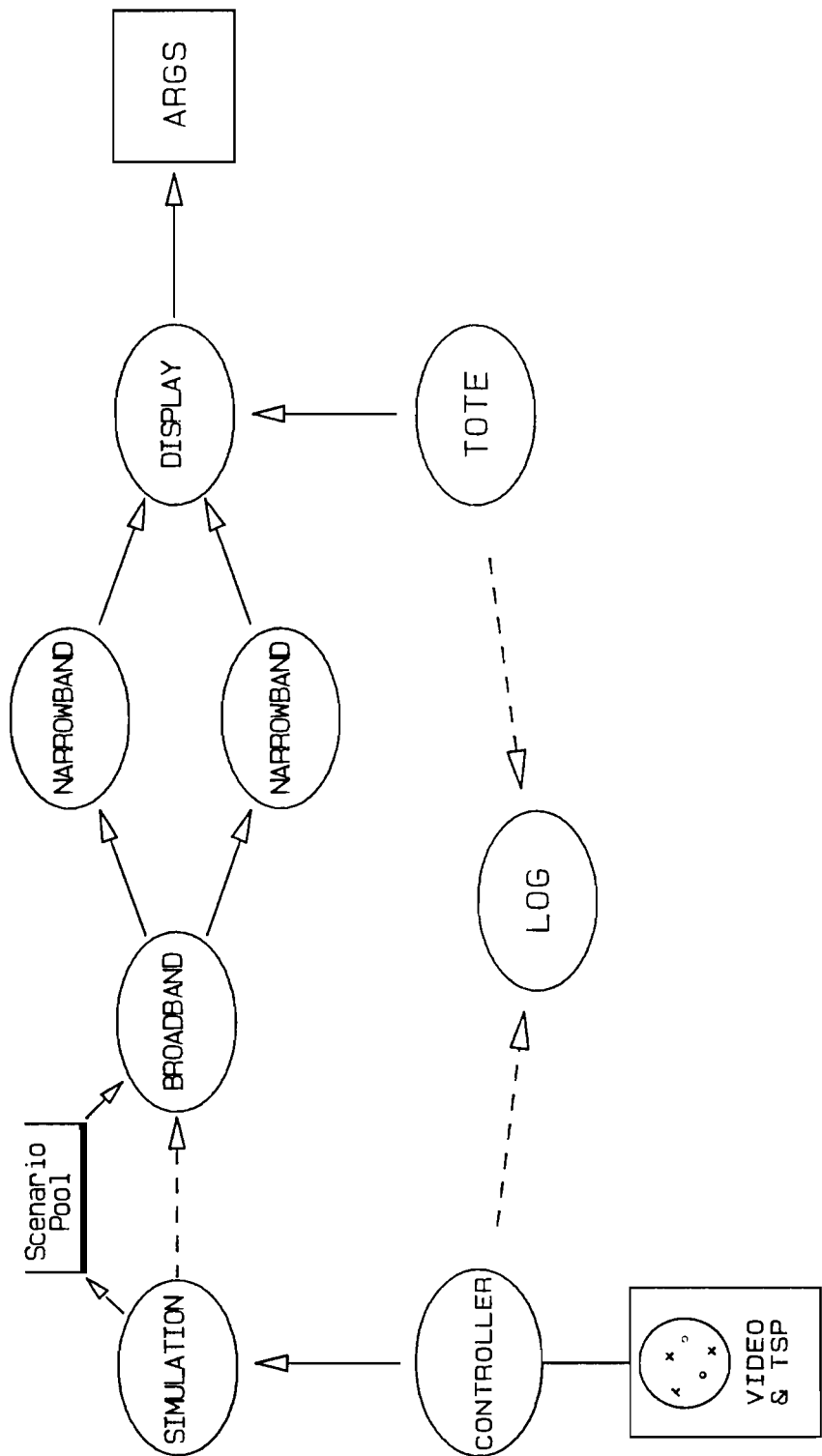


Fig. 1 : AN EXAMPLE CURTIS TADS SIMULATOR

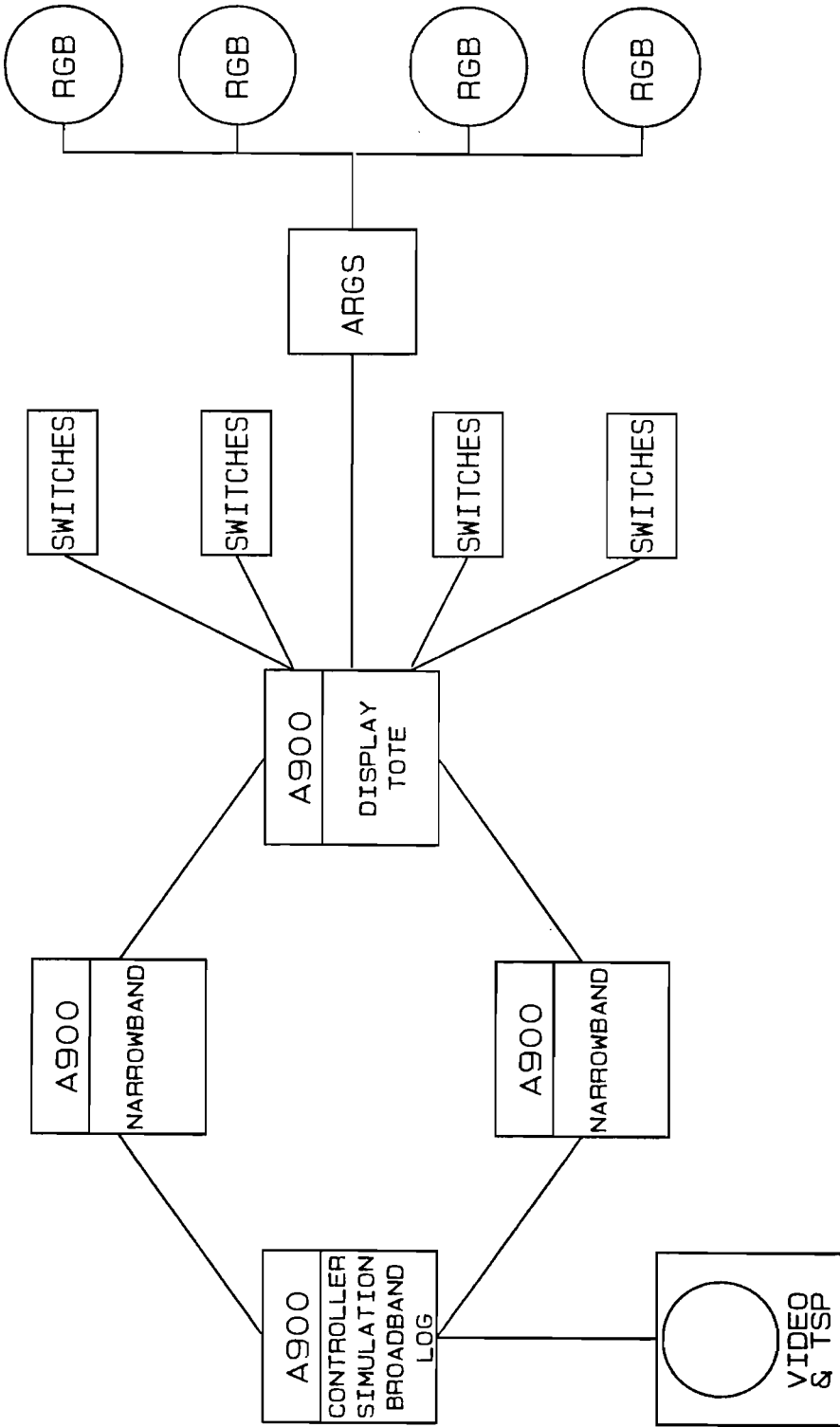


Fig.2 : PROGRAM DISTRIBUTION ON THE HARDWARE





DISC INTERFACES FOR HP SYSTEMS

Gary Vogelsberg
Hewlett-Packard Company
Disc Memory Division
P.O. Box 39
Boise, Idaho 83707

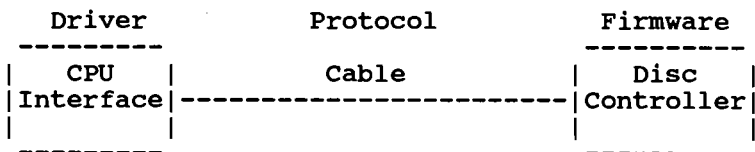
INTRODUCTION

One important, and often misunderstood, factor in the performance of an HP system is the disc channel. This paper is intended to give the HP computer user greater insight into the benefits of disc interfaces and the impact of the interfaces on HP system performance. The paper will also give some insight into future disc interfaces for HP systems.

WHAT IS A DISC INTERFACE?

The disc drive is a storage device, used to store data so it can be accessed by a system. Internal to the CPU, data is stored in memory. There are various hierarchies of memory within the CPU, allowing the CPU to be designed for optimum price/performance. Because CPU memory is expensive and volatile, disc drives are used to complement CPU memory. They provide a place where large amounts of data can be stored in a non-volatile form, and be easily accessible to the system.

The disc interface is the connection between the disc drive and the CPU. The interface provides a means of moving data between disc mechanisms and memory within the CPU. Examples of disc interfaces on HP systems include the parallel differential interface on the old 7906/20/25 disc drives, and HP-IB, which is used on many current products. The components of a typical disc interface are pictured below:



Driver/Interface Card

Within the CPU, hardware and software work together to manage disc I/O's. The disc driver is the software portion, while the interface card is the hardware portion. The interface card plugs into the internal CPU bus. The purpose

of the interface card is to receive I/O requests issued across the CPU bus by the driver and to communicate those requests to the disc controllers. The interface then manages the communications necessary to complete the I/O request. This includes managing the transmission or reception of the data as packets across the cable.

The intelligence of the interface varies. Microprocessors can be placed on the interface, effectively downloading some of the functions off the main CPU. This can result in less CPU overhead and improved performance.

Cable/Protocol

The cable is the physical connection between the interface card and the controller. The cable usually consists of multiple wires. It is through the cable that commands and data are sent.

Commands are sent across the cable using a protocol. The protocol can be defined as the language used by the interface card and the controller as they communicate across the cable. For HP-IB disc drives, Command Set 80 (CS-80) or Sub Set 80 (SS-80) are the protocols that are used.

Controller

The disc controller's function is to provide the intelligence required to execute commands issued across the interface. The three main functions of the disc controller are to decode commands, execute the commands, and report the execution status of commands back to the host. The controller can also be designed to provide diagnostics to aid in troubleshooting a disc drive.

INTELLIGENT VS. LOW-LEVEL INTERFACES

There are different levels of interfaces available on systems today. These interfaces can be generally classified into two groups: intelligent interfaces and low-level interfaces. Intelligent interfaces are able to accept and execute high-level commands, while low-level interfaces require very detailed instructions from the system. On HP systems, the HP-IB interface for disc drives is an example of an intelligent interface, while the older parallel differential interface is an example of a low-level interface.

When an intelligent interface is used, the system does not require a great deal of knowledge about the disc drive. The disc drive is viewed as a device made up of a series of logical blocks. The system requests a read or write to the

logical blocks in question, and then lets the controller execute the command.

When the intelligent controller receives a request from the system, it takes responsibility for completing the request. It issues the command to the disc mechanism to move the head to the correct location. Once the head is positioned correctly, the controller requests the channel to transmit/receive the data, and the data is sent. If the data is coming from the disc, the controller will verify that the data is good (using embedded error correction code) before sending it.

The key to intelligent controllers is delegation, allowing tasks to be off-loaded from the CPU. These controllers are capable of owning commands once they are issued from the CPU.

Low-level interfaces, on the other hand, do not provide a great deal of functionality to the system. The system has to provide separate serial commands to have the disc heads moved to the proper location via three-vector addressing (cylinder, head, and sector), tell the disc to read or write, and tell the disc how much to read or write. Little buffering is provided, and error correction has to happen at the host level.

THE TYPICAL INTELLIGENT I/O

Now that the pieces of a disc interface have been defined, let's take a look at a typical disc I/O across an intelligent interface. The I/O request is initiated by the CPU when it discovers that it needs data that does not reside in memory, or that some data needs to be posted to disc. The CPU decides what data it needs to transmit/receive, and then issues a command across the CPU bus. This portion of the I/O is referred to as CPU overhead.

Upon receipt of the I/O request, the interface card looks for idle time on the bus and sends the command down to the disc controller. The disc controller then acknowledges receipt of the I/O request. The time it takes the interface to receive and then issue these commands is referred to as interface overhead.

After the controller receives the command, it decodes the command and issues a command to the disc to seek to the correct location. The time required to do this is referred to as controller overhead.

The disc mechanism must then execute the requested command. It does this by means of a mechanical movement of the actuator, placing the heads in the disc drive over the correct track (the time required to execute this movement is referred to as seek time).

Once the heads are over the correct track, additional time is required for the disc surface to rotate so that the correct data is under the head. The time spent waiting for the correct data to rotate under the head is called latency.

Once all of this is complete, data can be transferred. Data is sent to or from the disc mechanism through the interface, cable, and controller. Once the data transmission is complete, the controller sends a message that the transaction has completed properly. The time spent transmitting data to/from disc is known as transfer time.

In a normal disc transaction on current HP systems, the majority of the time is spent in seek and latency. CPU overhead, interface overhead, and controller overhead are generally small components of the I/O execution time. Transfer time is variable, and is dependent on the amount of data being transferred.

INTERFACES ON HP SYSTEMS

HP-IB

HP-IB disc drives have now been available on HP systems for six years. This interface, offering one of the first intelligent controllers, has provided a number of advantages on HP systems.

Because of the intelligent controller used on the interface, integration time for new peripherals was greatly reduced. Since the system does not need to know the specific attributes of the disc drive (ie. number of sectors per track, heads, and platters), special new drivers do not need to be done for each new mechanism. Since less systems effort is required, the result is that HP has been able to turn out a large number of new peripherals with less systems effort required.

The common interface has also provided an easy inexpensive method of attaching a wide variety of peripherals and instruments to HP systems. On HP systems, the same HP-IB interface could be used to attach discs, tapes, printers, and plotters to the system.

The common interface on systems and peripherals also provides an easy migration path. Since HP-IB is supported

on systems from the HP 9816 engineering workstation to the new Model 850 Precision Architecture system, there is a great deal of flexibility in moving peripherals upward as computing needs grow.

The HP-IB interface has served HP systems very well. The transfer rate of the interface, roughly 1 megabyte per second, matches the requirements of many HP systems and peripherals very well.

HP-IB PERFORMANCE ISSUES

We at HP receive questions about HP-IB performance; it is often mentioned as a perceived bottleneck to system performance. The following paragraphs will discuss the channel, the way it is utilized, and its impact on system performance. The summary will indicate the impact of the HP-IB interface on performance, and discuss future alternatives.

Sharing the Channel with HP-IB

Intelligent interfaces are designed to share the channel, maximizing its utilization. Channel sharing is accomplished via several means across the HP-IB interface.

First, data is buffered on intelligent CS-80 drives. When a read command is being executed, data will be read into the buffer in the controller if the channel is being used by another device. As soon as the channel is available, the data can then be transmitted to the CPU from the controller. If buffering were not provided, data could not be transmitted until the data came under the head again. And the channel could be busy again the next time around.

Second, with intelligent interfaces, the controller accepts responsibility for executing commands at a high level. The result is that fewer commands are sent across the channel.

Third, rotational positioning sensing (RPS) is available with some of HP's intelligent controllers. RPS allows optimum channel utilization by reducing the amount of time that the channel is tied-up while the disc drive is incurring latency. The controller knows where the heads are relative to the desired data, and the controller waits and requests the channel just in time to transfer the data off the disc. The result is that the channel is not tied-up unnecessarily, and is available for other purposes.

Care needs to be taken in implementing RPS. RPS is designed to reduce channel contention in multiple drive configurations; it gives no improvement in single drive

configurations. RPS can actually have a negative impact on performance if the drive with RPS is put on the same channel with devices that do not use RPS. This is because devices that do not use RPS will tend to tie up the channel, making it less likely that the channel will be available when data is approaching the head. The result could be multiple latencies to complete an I/O.

HP-IB Impact on Disc I/O's

Earlier in this paper, the components of an intelligent I/O were outlined. They include interface overhead, controller overhead, seek, latency, and transfer time. The transfer rate of the interface can affect the transfer time, but does not affect other portions of the typical I/O.

It should be noted that the transfer time is usually a small portion of the total transaction time for a typical I/O. As an example, a 1 kilobyte random I/O to a 7957 disc drive has the following I/O components:

Average controller overhead	3.0 ms
Average seek time	29.0 ms
Average latency	8.3 ms
Average transfer time (at 853 kbytes/sec)	1.2 ms

Total average transaction time	41.5 ms

In this example, the transfer time accounts for only 3% of the average transaction, so transfer rate has very little impact on performance. The impact of transfer time on the total transaction will vary depending on the size of the I/O, with transfer time becoming more significant as the size of the I/O increases.

It should also be noted that in the above example, improving the speed of the interface would not change the transaction time. This is because the average transfer rate of the 7957 is very well matched to the speed of the HP-IB interface; HP-IB allows a 7957 to transfer data at its expected rate. A similar analysis can be done for other disc drives.

The above example demonstrates some of the issues in considering the impact of the interface on system performance. The level of performance gain from a faster interface will therefore be very system and application specific.

Performance gains from a faster interface will not be significant unless there is channel contention. The amount of channel contention will impact the amount of performance. The main factors that will impact the amount of contention

on a channel: 1) the number of discs on the channel, 2) the number of I/O's that are being processed, and 3) the size of the I/O's.

SCSI

The Small Computer System Interface (SCSI) is currently being worked on as an interface for HP technical systems, and will be available on some HP technical systems in the future. SCSI is an industry standard interface.

There are several implementations of SCSI. The transmission of data can be done across either single-ended or differential cables. Data can be sent either synchronous or asynchronous. The highest transfer rate currently available, which requires synchronous transmission, is up to 4 megabytes per second. The SCSI interface supports total cable lengths of up to 25 metres with the differential cable.

There are many similarities between HP-IB and SCSI. As a result, it offers many of the same advantages that the HP-IB interface offers. The higher data transfer rate is one advantage that SCSI has over HP-IB.

Fiber-optics

HP is also working on a new fiber-optic interface for HP systems. This interface will provide a fiber-optic connection between a system and a group of disc drives.

There are several advantages that fiber-optics offers:

- Fiber-optic technology can be used to provide very fast data transfer rates
- Data can be transmitted over much longer distances using fiber-optic cables
- Signals sent across fiber-optic cable are not susceptible to radio frequency or electro-magnetic interference.
- Interference is not emitted by fiber-optic cables.

Fiber-optic technology will provide a high-performance connection to HP systems. With the HP Precision Architecture systems, larger transfers will occur and the need for a high band-width channel will be acute.

More powerful systems will also require larger disc configurations. These will be able to be built while minimizing channel contention. Long fiber-optic cables will also provide a great deal of flexibility in the placement of

equipment. And little concern will need to be given to environmental concerns.

Today, Hewlett-Packard is in the position to offer a variety of interfaces to meet the requirements of HP systems. Understanding the implications of disc interfaces on systems will make the choice easier.

DISC PERFORMANCE
Carol Hubecka Bergman
Hewlett-Packard Company
Disc Memory Division
P.O. Box 39
Boise, Idaho 83707

INTRODUCTION

There are many factors that affect the performance of a disc drive, which in turn can have a direct effect on overall system performance. The information presented herein is intended to provide an understanding of disc performance and the associated variables. It will examine the specific performance of several new products recently introduced by Hewlett-Packard Company, including the HP 7936 and 7937 disc drives, the HP 7957 and 7958 disc drives, and disc controller cache.

While this document addresses specific performance on HP 9000 Series 300 HP-UX and HP 1000 RTE-A systems, most of the concepts will be of general interest to all users of Hewlett-Packard's technical computers. The first section is devoted to the basic components of a disc transaction. The significance of each component in relation to disc performance is examined. The second section introduces system variables such as locality, channel contention, transfer size, etc., and their impact on disc performance. The third section discusses disc controller cache as a means to improve performance. Throughout the document examples using Hewlett-Packard's latest products will be provided. These examples should clarify the effects of the performance variables as well as assist in characterizing the performance of these specific products.

DISC PERFORMANCE BASICS

Any discussion of disc drives naturally begins with an understanding of the disc and its operation. In discussing disc performance it is necessary to first understand the means by which a transaction occurs on the disc.

Disc Transaction:

A disc transaction is comprised of four components: disc controller overhead, a physical seek, a physical latency, and the actual transfer of information. Traditionally, transfer

time has been the smallest component of the transaction, with controller overhead being next. The HP 7936 and 7937 drives are the exception with controller overhead being the smallest component of the transaction. For all drives, the mechanical seek and latency generally comprise the largest part of the transaction.

The following table compares the individual transaction components of the current Hewlett-Packard disc drives (all measurements are in milliseconds):

Disc Drive	Controller Overhead	Seek	Latency	Transfer	Cache Impact
HP 7914	4.0	28.1	8.3	1.2	
HP 7933/35	3.5	24.0	11.1	1.0	
HP 7933/35XP	4.5	24.0	11.1	1.0	-16.6*
HP 7936/37	<1.0	20.5	8.3	1.0	
HP 7936/37XP	1.2	20.5	8.3	1.0	-15.9*
HP 7941/45	10.1	30.0	8.3	2.0	
HP 7942/46	10.0	30.0	8.0	2.0	
HP 795X	3.0	29.0	8.3	1.2	

* The reduction in time is due to the reduced number of seeks required for a drive with controller cache. The actual performance improvement on a drive will vary depending on the read hit rate and the read percentage; the listed values are based on a read hit rate of 70 percent and a read percentage of 70-75 percent.

1. Controller Overhead

The disc controller provides the intelligence of a transaction electronically by:

- * Decoding the disc command sent by the host computer,
- * Executing the command,
- * And reporting the execution status back to the host computer.

The amount of time required for the controller to process instructions, translate data, check for transmission errors, etc. is known as controller overhead.

In the past, disc controllers were very hardware intensive, which resulted in high speed command executions. This speed, however, came at a high price. The development time was considerably long and internal and on-line diagnostics from the host were not available.

Consequently, adding intelligence to the controller added overhead in the disc transaction. Experience, however has allowed us to make our controllers efficient in doing the greatest amount of work in the smallest amount of time. This is evidenced by the low controller overhead of the HP 7936 and 7937 drives.

2. Seek Time

Once the controller has decoded the command, the disc must perform some mechanical positioning to prepare for the execution of the command. The drive must first find the desired disc location by moving its head to the correct media track. This action is defined as the seek. Seek times have been significantly reduced in the past five years, but they still represent the largest component of the transaction.

The exact amount of time required to find the desired track for a given transaction will be dependent upon the location of the track on the media and the current position of the head. Therefore, a more accurate estimate of seek time is the average random seek, or the time to do all possible seeks divided by the total number of seeks possible. This assumes that the actuator is moved around with equal probability of landing at a given cylinder. The average time will be the average random seek.

3. Latency (or Rotational Delay)

Once the drive has found the correct track it must then find the desired sector on that track. The media continues to rotate beneath the head as the track is searched. The time required for one rotation of the disc is defined as the latency time, and is directly related to the rotational speed of the disc. The rotation, like the seek, is mechanical.

This definition of latency is certainly "worst case" time since the head may be considerably closer to the desired sector than one full rotation. A more accurate measure of rotational delay is the average latency. It is defined as the time to complete one half of a rotation.

4. Data Transfer

The actual movement of data from the CPU to the disc drive or vice versa is known as the data transfer. The amount of time required to complete this function is dependent upon the rate at which the data moves between the system and the disc drive, as well as the size of the transfer. The individual transfer rates of each device that is involved in the

transfer of data (the system I/O channel, the bus, and the disc drive) determine the maximum achievable data transfer rate. In fact, the rate of transfer will vary among these devices, making the maximum attainable transfer rate only as fast as the slowest device. In other words, the chain is only as strong as the weakest link.

Since the amount of time devoted to the data transfer is dependent upon the size of the transfer and the achievable transfer rate, there will be variations from transaction to transaction. Therefore, an accurate measure of transfer time would be average transfer time. Hewlett-Packard uses an average transfer size of 1 Kbyte to measure average transfer time. Using this average size, the average transfer time is then calculated for each drive.

Transaction Time:

Each of the components of a disc transaction contribute to the total time to complete that transaction. The summation of the average time it takes to complete each component is a good measure of the total average time to perform a disc transaction. The total average transaction time is specific to the product in question and does not take into account individual host system attributes.

The following figures represent the various average transaction times for Hewlett-Packard disc products:

Disc Drive	Average Transaction Time
HP 7911/12	40.6 ms
HP 7914	41.6 ms
HP 7933/35	39.6 ms
HP 7933/35XP	24.0 ms *
HP 7936/37	30.8 ms
HP 7936/37XP	15.1 ms *
HP 7941/45	50.4 ms
HP 7942/46	50.0 ms
HP 795X	41.5 ms

* The reduction in time is due to the reduced number of seeks required for a drive with controller cache.

Performance Metric:

Hewlett-Packard uses the metric of I/Os per second to measure disc performance. I/Os per second is defined as the maximum number of disc transactions per second that a specific drive can perform at a given transfer size (usually 1 Kbyte). This

measure can be calculated by taking the inverse of the total average transaction time that was just described. Once again, the measurement represents raw disc performance and does not take into account any system specifics. Actual performance will vary with system and application.

NOTE: I/Os per second as defined is a Hewlett-Packard metric and not necessarily an industry standard.

The following table lists the average transaction time for each of the Hewlett-Packard disc products and the resultant number of I/Os processed per second based on 1 Kbyte transfers.

Disc Drive	Average Transaction Time	I/Os per Second
HP 7911/12	40.6 ms	24.6 I/Os
HP 7914	41.6 ms	24.0 I/Os
HP 7933/35	39.6 ms	25.3 I/Os
HP 7933/35XP	24.0 ms	41.7 I/Os
HP 7936/37	30.8 ms	32.5 I/Os
HP 7936/37XP	15.1 ms	66.2 I/Os
HP 7941/45	50.4 ms	20.0 I/Os
HP 7942/46	50.0 ms	20.0 I/Os
HP 795X	41.5 ms	24.1 I/Os

SYSTEM CONSIDERATIONS

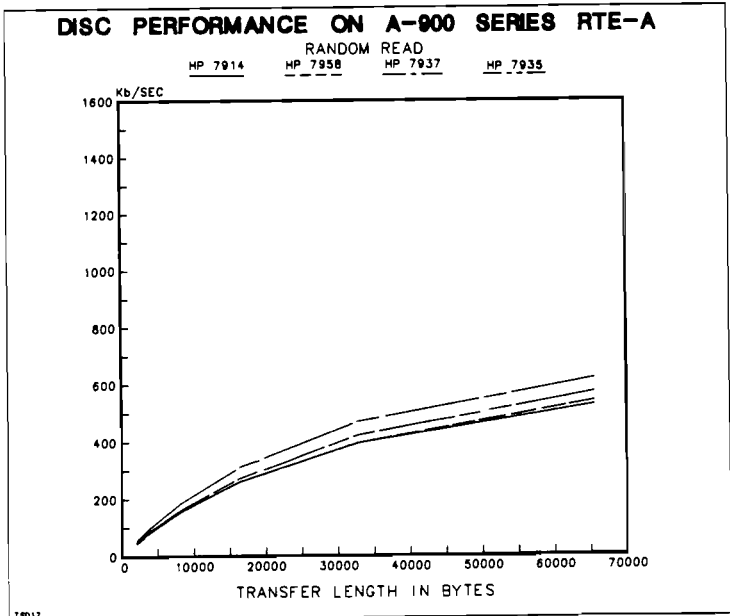
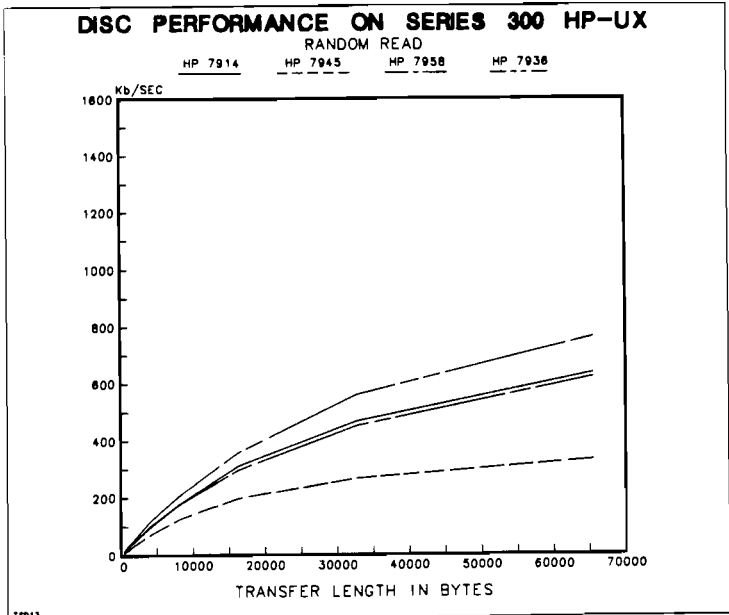
Understanding the basic components of the disc transaction provides a starting point for examining various host system factors that influence disc performance. The following information discusses the impact these factors have on the basic components of a disc transaction, as well as steps that can be taken to improve performance.

Transfer Size:

As mentioned earlier, Hewlett-Packard defines disc performance based on 1 Kbyte transfers. Since no system uses exclusively 1 Kbyte transfers, it is useful to examine disc transfer rates at varying transfer sizes. Understanding how your system is used will help you determine what disc drive best fits your performance needs.

The following graphs show, in Kbytes per second, the raw transfer rate of HP's discs on HP-UX and RTE-A. The graphs show that the size of the transfer has a direct impact on

performance. Because absolute performance will vary with system and application, the data should only be used to compare relative drive performance.



Locality:

All of the information presented thus far is representative of a strictly random access environment. In a random environment, the disc I/O requests are satisfied by disc accesses that are not necessarily within the same proximity. This is not a realistic model for a disc drive operating on a typical system. For most systems there is a principle known as locality. Locality refers to the situation where the disc I/O requests are satisfied by disc accesses within the same proximity. Locality is normally measured as a percentage of the total requests. This percentage is commonly referred to as the locality factor.

For the purposes of our testing, locality is defined as follows:

Random Read - Each read to the disc is to a random location. (0% locality)

Local Read - Each read following a random read is a sequential read. Therefore, one half of the read accesses are random and the other half are sequential. (50% locality)

Sequential Read - All read locations are accessed sequentially. (100% locality)

The following graphs demonstrate the impact locality has on the HP 7958 disc drive on HP-UX and RTE-A. Localized requests require less mechanical movement, specifically less actuator movement. Localized requests will be satisfied by data within or near the same track. This reduces and often eliminates the distance the drive has to seek. This reduction in positioning time greatly reduces the overall transaction time, which provides a substantial performance improvement.

HP 7958 PERFORMANCE ON SERIES 300 HP-UX

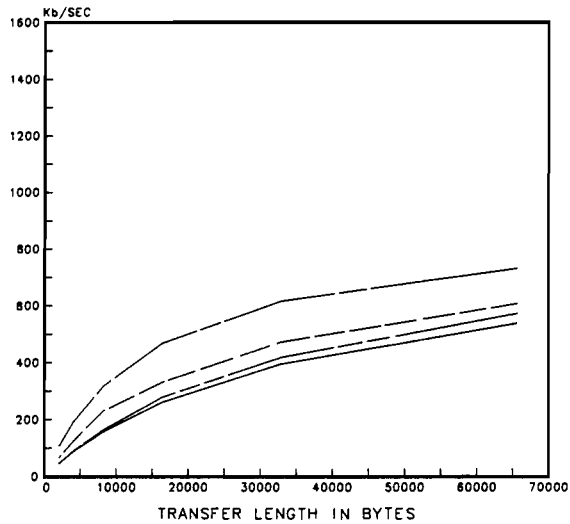
LOCALITY

RANREAD

LOCREAD

SEQREAD

RANWRITE



TSD10

HP 7958 PERFORMANCE ON A-900 SERIES RTE-A

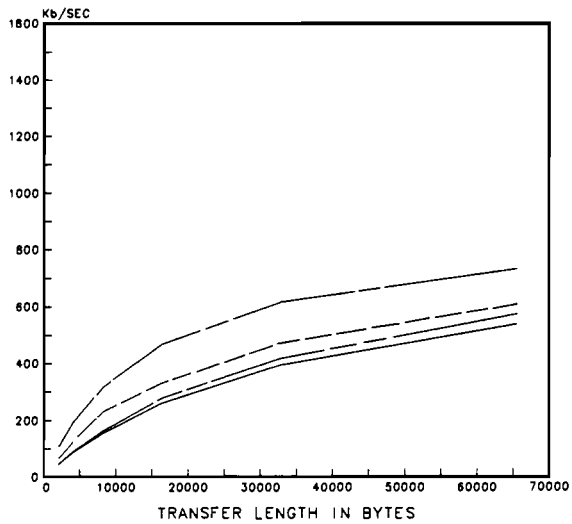
LOCALITY

RANREAD

LOCREAD

SEQREAD

RANWRITE



TSD09

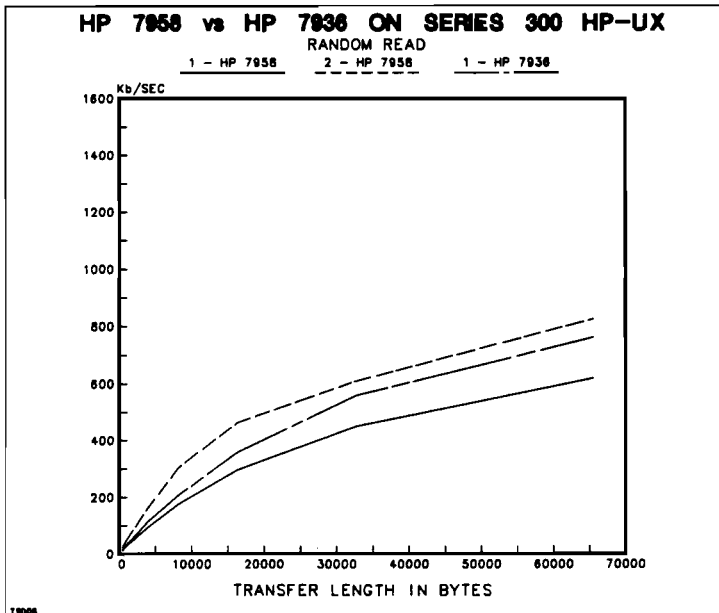
System Configuration:

The manner in which a system is configured, including both hardware and software, also impacts disc performance. When initially configuring a system it is important to understand what the system will be used for and the mass storage requirements that will meet those needs. In evaluating the disc products available, the number of megabytes offered, the footprint, the performance, the power requirements, as well as the price, should be considerations.

1. Number of Spindles

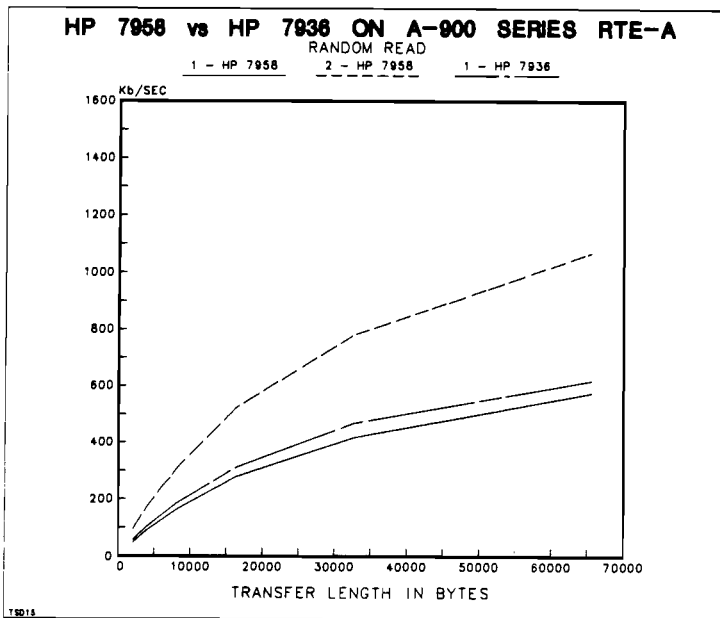
With the multitude of disc drive capacities to select from, it often becomes unclear which combination of drives optimizes performance. Of specific interest is the question of two smaller drives versus one larger one. The answer is very application dependent. For a system that can keep both drives busy, two drives is generally the best answer. On the other hand, a single faster drive is the better solution for a very localized system with little multiple processing occurring.

The following graph shows the difference in performance of two HP 7958 drives versus one HP 7936 on HP-UX.



It should be stressed that the HP-UX user MUST spread his files across the multiple discs to actually see the benefit of multiple spindles. Both drives must be accessed frequently to see the performance gain. One such configuration is to split the root and the virtual memory devices.

The following graph shows two drives can also perform better than a single drive on RTE-A. Again, both drives must be accessed frequently and files spread across the devices.

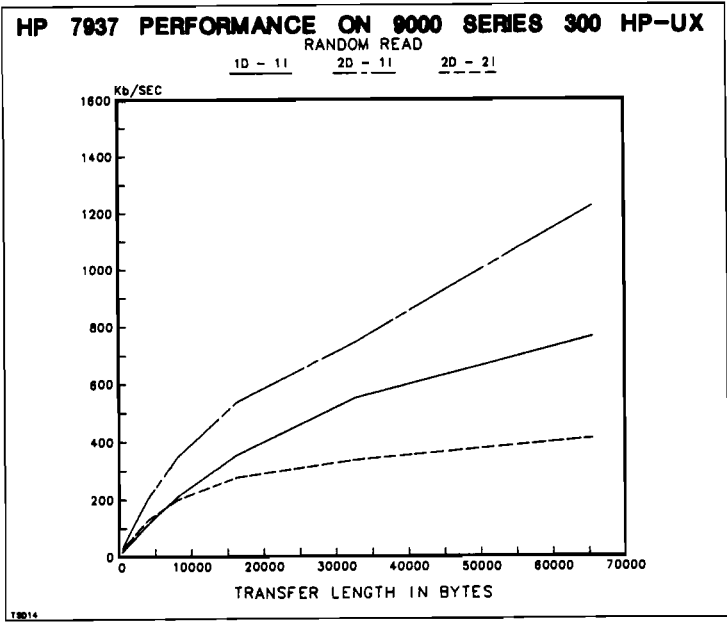


2. Channel bottleneck

Channel bottlenecks can occur when multiple disc drives share a channel in an I/O intensive environment where file access is spread relatively evenly across the discs on a channel. Different systems handle channel usage in different ways.

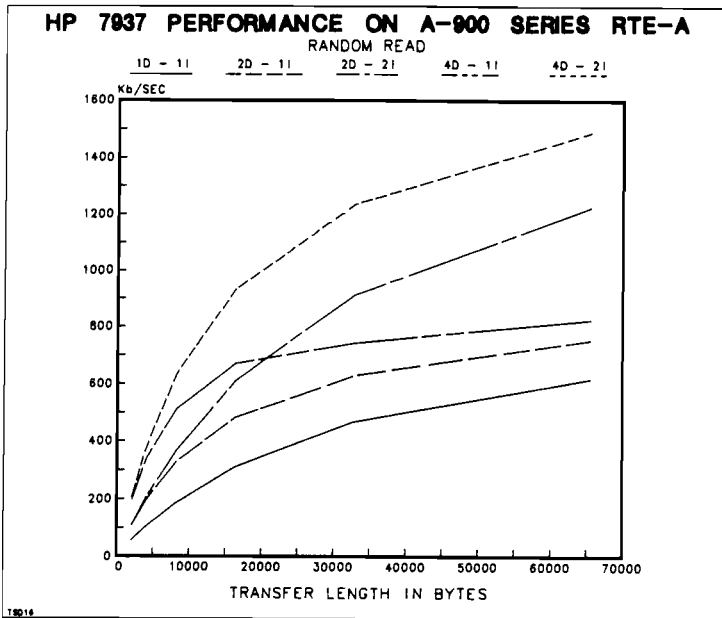
Series 300 HP-UX, for example, handles multiple drives on a shared channel very well as the next graph shows. The driver interleaves the components of the transaction request quite efficiently to increase channel utilization. For instance, while drive one is seeking, drive two can report its status back to the host. Two drives on separate channels will

provide performance comparable to the same two drives sharing a channel as long as both interfaces are high speed. The following graph shows how performance decreases when one of the two interfaces is low-speed.



(Note: "D" denotes disc(s) and "I" denotes interface(s).)

RTE-A handles multiple drives on a shared channel efficiently because the driver interleaves the components of a transaction request. Like Series 300, drive two can report its status back to the host while drive one is seeking. Multiple drives on multiple channels, however, provide even better performance. This is because, in addition to interleaving requests on a single channel, requests can be interleaved across multiple channels. This means that more requests can be processed in the same amount of time as the next graph shows.



(Note: "D" denotes disc(s) and "I" denotes interface(s).)

3. Dedicated System Disc:

Another factor of system configuration with a major impact on disc performance is the location of the system disc. By definition, the system disc, or root device, is the disc on which all system files reside. Since the contents of the system disc, i.e. directories, are accessed again and again, there will be a great deal of I/O activity directed at this device. As a result, it often makes sense from a performance standpoint to distribute the remaining user disc I/O activity among the remaining discs. This allows the system disc to be dedicated for system level I/Os. This will help the performance of the other discs on the system, as well as the system or root disc.

4. Spread of Peripherals across the Channel:

In conjunction with the number of available channels, it is also important to manage the spread of the existing peripherals across these channels. In fact, the manner in which peripherals are spread across the channels has more of

an effect on performance than the quantity of available channels. In order to spread the peripherals in a manner that will provide the most benefit to the system, it is important to understand how each peripheral is utilized.

The importance of spreading peripherals is evident in situations where disc drives share a channel with tape drives or printers and all are utilized simultaneously. The tape drives and printers have a tendency to transfer larger blocks of data which require longer periods of channel time, thus making it increasingly difficult for the disc to connect to the channel when necessary. By placing tape drives and printers on channels separate from disc drives, an environment is created that promotes improved performance.

5. File Management:

Another method of managing the I/O activity across multiple discs is through effective file placement and management. By physically spreading frequently accessed files among several discs, the amount of contention per disc can be reduced. Since the objective is to reduce both drive and channel contention, special attention needs to be given to spreading files across channels as well as discs. A good means of distributing files on HP-UX is to first split the virtual memory disc and the root device. Since both of these discs are accessed so frequently, configuring them on separate discs will reduce contention. The remaining user files can then be spread between the two, or more, discs.

RTE-A performance can be improved by distributing files according to use. Spreading frequently used files across discs and channels will positively affect performance.

6. Virtual Memory Allocation:

Virtual memory is a memory management scheme which, in addition to main memory, uses disc storage as secondary memory. Virtual memory allows the system's logical address space to be larger than its actual physical address space.

Because the virtual memory device and the system disc are accessed so frequently with HP-UX, it make sense to locate them on different physical discs. This will reduce the number of collisions on the discs and improve performance.

If your RTE-A system uses lots of virtual memory, then the same advice applies; split the virtual memory device from the system disc.

7. Load Management:

The various types of applications and software may still have the greatest impact on system performance. It is important to know the types of applications that run on a system. Are they CPU intensive, I/O intensive or both? Armed with this information the best mix of jobs can be scheduled, utilizing the CPU resources and disc drives for optimum performance. For instance, I/O intensive programs should not be mixed with other I/O intensive programs. A better solution would be to mix a CPU intensive program with an I/O intensive program.

CONTROLLER CACHE

In the past, disc drive designers have designed and tested disc products as if all sectors of the disc had an equal probability of being accessed by the host. This is not always an accurate assumption due to the locality factor. In fact, the access patterns of the host are usually not random. Some areas of the disc are accessed over and over (system directories, for example) while other areas may not be accessed for long periods of time. With this in mind, Hewlett-Packard has designed high performance disc drives that keep these frequently accessed areas in random access memory, which is faster to access than the disc surface. This RAM, more commonly known as controller cache, has been designed for use with Hewlett-Packard's high-end disc drives, the HP 7933, 7935, 7936, and 7937.

Controller cache on the HP 7933XP and 7935XP products consists of one megabyte of dynamic ram added to each drive. This cache stores the last 255 4096-byte active areas of the disc so future accesses of these same areas can be provided to the host without a physical disc read actually occurring. This provides a higher level of disc performance over the standard HP 7933 and 7935 drives and can contribute to greater overall system efficiency in the right application.

Controller cache on the HP 7936XP and 7937XP products provides two megabytes of dynamic ram for each drive. This cache stores the last 440 4096-byte active areas of the disc. 4096 bytes of write cache enables immediate response for a write transaction. The combination of a read and write cache provides greater overall system efficiency in the form of system throughput and response time. It provides the additional benefit of a broader range of applications than the HP 7933XP and 7935XP.

Cache Performance:

The physical performance of a disc drive with controller cache is the same as a non-cached drive in that both have controller overhead, a seek time, a latency time, and a transfer rate. The difference is that with a cache drive the seek and latency are often omitted. To compensate for this occurrence, the controller cache factor is subtracted from the average transaction calculation.

1. Controller Cache Overhead

It is important to keep in mind that any cache system imposes some overhead. This overhead is incurred when cache is searched and the user request is not found so the data has to be read from disc anyway. Also, reading more data than the user requests adds additional overhead if it is never accessed. The cache overhead of the HP 7936XP and 7937XP is significantly lower than that of the HP 7933XP and 7935XP, while both are only a minute portion of the whole transaction.

2. Seek Time

The physical seek characteristics of the disc drive will not be affected by controller cache. However, the purpose of the cache is to reduce physical seeks by eliminating many physical accesses of the disc for reads. Therefore the effective seek time seen by the host will be improved.

3. Latency Time

Physical latency will not be directly affected by controller cache either. However, any host read which is answered through the cache will not induce a latency. Therefore, the effective latency is improved.

4. Transfer Rate

The maximum data transfer rate of 1.0 Mbyte/second will apply to all data transfers which are supplied from the cache memory or from a physical access of the disc drive over HP-IB.

5. Transaction Time

Disc transaction time is defined as the amount of time required for a disc drive to complete an I/O. It is the sum of the following components:

- * Controller Overhead Time
- * Seek Time
- * Rotational Latency
- * Time to Transfer 1024 bytes
- * Controller Cache Impact

The primary effect of controller cache is to reduce the disc transaction time. The average disc transaction time on a disc with controller cache is dependent on the specific application. In particular, the read hit percentage and read/write mix have a significant impact on the disc drive's performance.

Cache Positioning:

Controller cache and RTE-A can be a good match. When properly positioned, controller cache can provide a significant performance improvement to the HP 1000. The HP 7933XP and 7935XP are supported on RTE-A 4.0 and later, while the HP 7936XP and 7937XP are supported on RTE-A 4.1 and later.

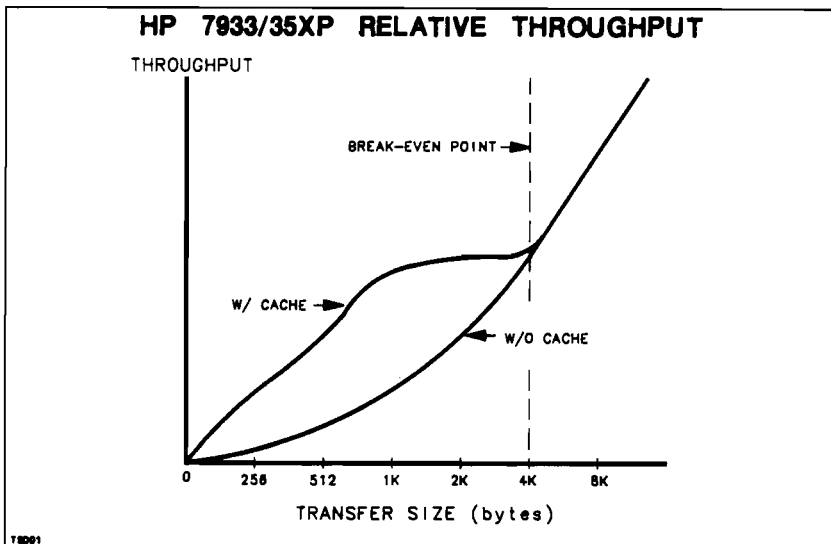
Controller cache, as currently defined, is not a good fit for HP-UX and, thus, is not supported on any HP-UX systems. The 4096-byte cache domain size is not optimal for HP-UX. Better performance is generally seen by increasing the HP-UX block size to 8192 bytes.

1. HP 7933XP and 7935XP

HP 1000 users who are I/O bound and concerned with performance should consider controller cache according to the following guidelines:

- * Locality of primary applications - The greater the locality of the disc requests, the greater the performance improvement from controller cache. Because reads are cached in blocks of 4096 bytes, locality improves the chance of the data already being in cache when the current transfer is small.

- * Transfer size - Throughput with controller cache is increased for read operations when transfer sizes are under 4096 bytes as the following graph shows.



* Read/write ratio - The greater the ratio of reads to writes, the greater the performance improvements from controller cache. Controller cache does not speed up write operations because it "writes through" cache, meaning that the disc drive will not acknowledge to the CPU that the write is complete until the data has been written to the disc surface.

If the above guidelines are followed, an average HP 1000 user can expect a 5-20% performance improvement from controller cache on the HP 7933XP and 7935XP. One customer doing exclusively small sequential reads saw throughput improvement of two to three times!

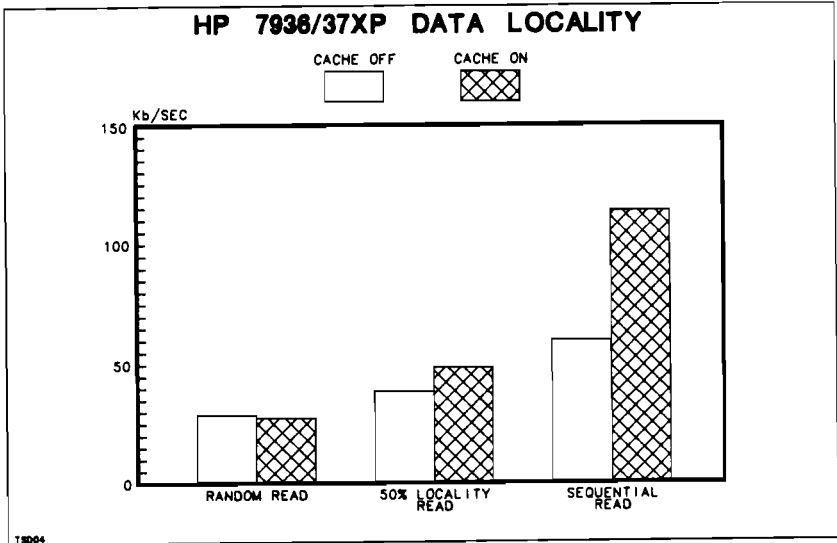
HP 1000 users doing primarily write operations (i.e. data logging) will not benefit from controller cache on the HP 7933XP and 7935XP. In fact, they may experience a slight (3%) degradation in performance.

2. HP 7936XP and 7937XP

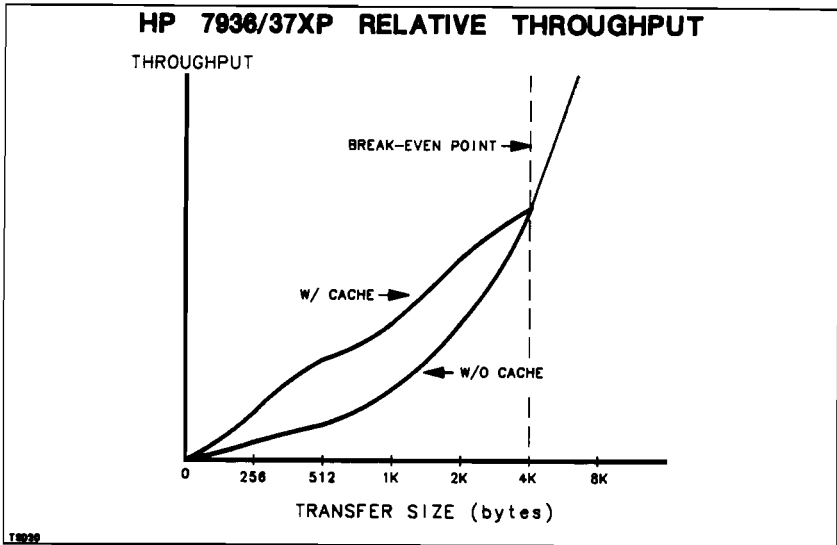
The implementation of write cache on the HP 7936XP and 7937XP broadens the number of HP 1000 applications that will benefit from the use of controller cache. As a result, a read intensive environment is no longer required to see benefits

from the use of cache. The following guidelines, however, do still apply.

* Locality of primary applications - The greater the locality of the disc requests, the greater the performance improvement from controller cache. Because reads are cached in blocks of 4096 bytes, locality improves the chance of the data already being in cache when the current transfer is small as the following graph shows.



* Transfer size - Throughput with controller cache is increased for read operations when transfer sizes are under 4096 bytes as the following graph shows. Write operations must be under 4096 bytes to utilize the write cache.



If the above guidelines are followed, an average HP1000 user can expect a 10-30% performance benefit from controller cache on the HP 7936XP and 7937XP. Actual performance gain will depend on application.

Controller cache provides more than just higher levels of disc performance over standard drives. The combination of faster disc transactions, reduction of disc mechanical movements and higher disc reliability contribute to greater overall system efficiency in the form of system throughput. (System throughput being defined as more transactions completed in a given time period.) The availability of write cache on the HP 7936XP and 7937XP disc drives and the resultant overlapping of reads and writes further contributes to improvements in response time.

SUMMARY

The disc drive is an integral part of any computer system. The system utilizes the disc continuously to read in data and write out data. The speed at which the disc performs these functions plays a crucial role in the overall performance of the system.

There are many variables associated with the performance of the disc. A knowledge of these variables and their effects is essential for ensuring efficient use of the disc. The first step in gaining this knowledge is understanding the following basic components involved in a disc transaction:

- * Controller Overhead
- * Seek
- * Latency
- * Data Transfer

The amount of time the disc spends completing each of the above functions determines the total amount of time involved in processing the disc transaction. Naturally, the physical limitations of the specific disc play an important part in determining the amount of time spent on each component. This is an area that Hewlett-Packard is continually improving to provide maximum performance. This commitment is evident with the introduction of the new HP 7936 and 7937 drives, the HP 7957 and 7958 drives, and especially the introduction of controller cache. In addition to the physical characteristics of the disc, there are many external factors that can influence the amount of time devoted for each component of the disc transaction. These include:

- * Transfer Size
- * Data Locality
- * Channel Contention
- * System Configuration

It is the goal of Hewlett-Packard to maintain the performance of Hewlett-Packard systems through the development of new disc technologies and further refinement of existing ones.

Random Vibration Control Program

Minoru Suzuki
and
Masataka Kozuka
Nippon System Gijutu Co.
1-2-8 Toranomon, Minato-ku
Tokyo Japan

1. Introduction

Conventional vibration tests have to be done in the field to provide an actual operating environment for a specimen.

The Random Vibration Control Program (RVC), however, enables vibration tests to be performed with a vibrator as many times as required under conditions very close to the actual vibration state, D/A and A/D converter.

RVC runs on a HP-1000 A-Series computer with a special signal processor.

RVC measures frequency characteristics of the specimen, jigs and vibrator quickly, corrects the measured data automatically, and controls vibration via a signal having the proper spectrum. In addition, RVC can analyze the vibration of the specimen.

RVC can be used with the electro-hydraulic or electro-magnetic vibrator controller.

2. Theory of Random Vibration Control

First, the target power spectral density is entered to the computer (the spectrum used here is called a standard spectrum). Figure 2-1 shows the standard spectrum.

The input standard spectrum is randomized, then inverse Fourier transformation is performed on it, and the results are converted to a time-base signal (drive signal) which is sent to the vibrator.

The vibrator then generates vibration according to the frequency characteristics of the test system (vibrator, jigs, and specimen), that is, the relationship between excitation and vibration.

Usually, the generated vibration does not correspond with the input standard spectrum.

Next, the resultant vibration status of the specimen is detected as an electrical signal by the sensor for recording (this feedback signal is called a control signal).

The control signal is converted to a power spectral density (control spectrum) to analyze the power spectrum by comparing it with the standard spectrum.

Compensating for the difference between the control and standard spectrum, the computer generates a new drive signal and sends it to the vibrator. The new drive signal can be obtained as follows:

$$\text{New drive signal} = \frac{\text{Standard spectrum}}{\text{Control spectrum}} \times \text{previous drive signal}$$

Figure 2-2 shows the data flow under random vibration control.

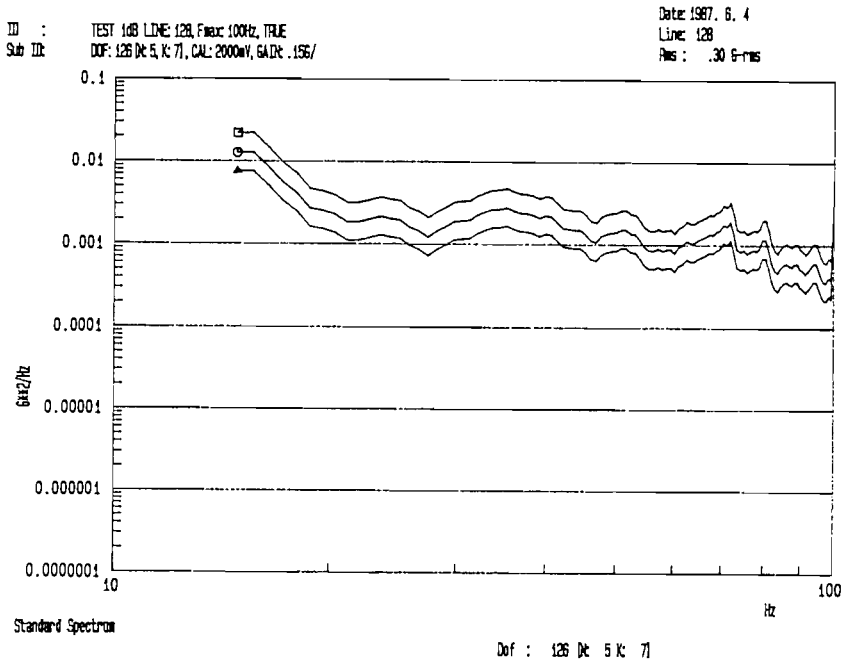


Figure 2-1 Example of standard spectrum

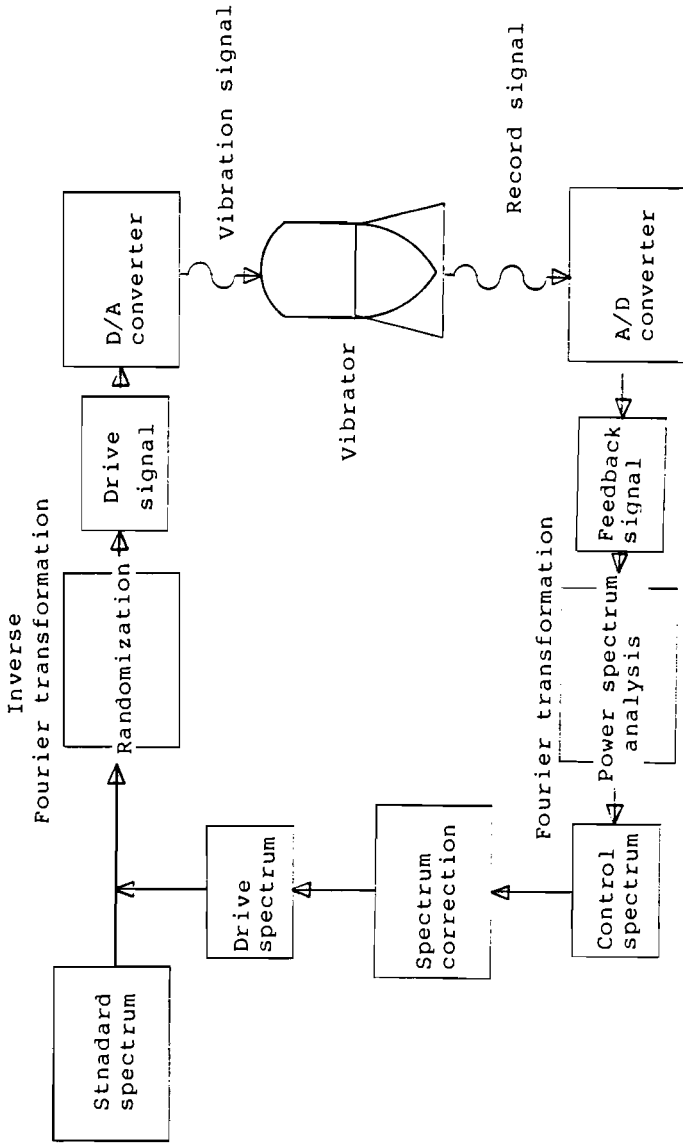


Figure 2-2 Random vibration control flow

3. Operating Environment

This section describes the RVC operating environments. Figure 3-1 shows the hardware configuration required for execution of RVC.

(1) Hardware environment

- (a) Computer and operating system
 - (i) HP-1000 A Series computer
 - memory: 3M bytes or greater
 - (ii) Graphic terminal: 2393, 2397...
 - (iii) Magnetic disk drive unit:
 - 7957, 7958, 7936, 7937...
 - (iv) Operating system: RTE-A
- (b) Analog subsystem
 - NSG MD-5000 CS (one ADC and one DAC)
- (c) High-speed signal processor (NSG SP-8000)

(2) Software environment

- (a) NSA basic part
 - NSA: NSG Signal Analyzer Program
- (b) Graphic library
 - HP-1000 DGL-II
 - NSG graphic library II

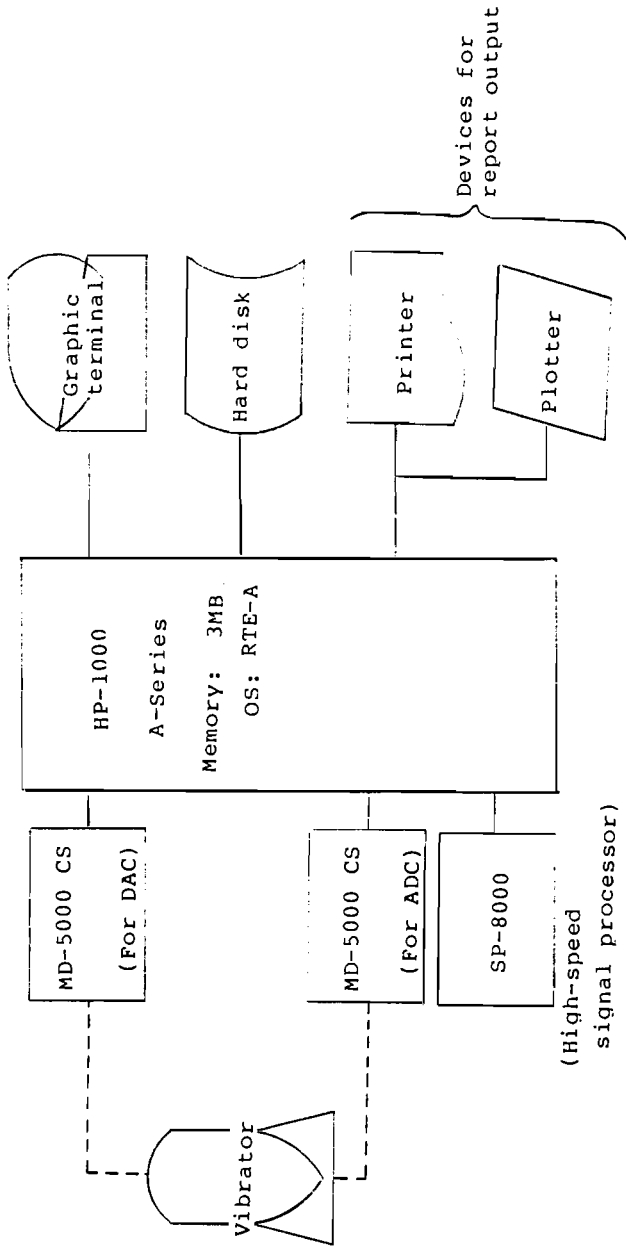


Figure 3-1 Hardware configuration

4. Outline

4.1 Role of Random Vibration Control Program

The Random Vibration Control Program (RVC) operating under the NSG Signal Analyzer Program (NSA) controls the vibrator.

NSA performs various types of measurement, control, and analysis with a high-speed A/D or D/A converter. It transfers information on the HP-1000 minicomputer system via programs and classe I/O, each of which is created to perform a specific function.

Figure 4-1 shows the relationship between NSA and RVC.

4.2 Outline of Functions

- (1) RVC controls vibration according to the vibration conditions set by the user. It can also output data on vibration to the graphic terminal for monitoring, and save such data in disk file.
- (2) Conditions of vibration control can be set by the user easily in an interactive manner. Their default values are also provided.
- (3) Vibration control is divided into pretest and main vibration phases. Pretest is enabled on up to level four, and the results of the pretest are used in the main vibration phase.
- (4) The monitor enables a check on the status of vibration control. Data can also be saved during vibration control.
- (5) The report output function plots data which was saved during vibration control on the graph. It is useful for a check on whether proper vibration was applied.

The report output function can output not only a single data item, but also multiple data items for comparison. In addition, it enables data to be overlaid.

4.2.1 Input of vibration conditions

Conditions for vibration (gain, input channels, calibration value, and so forth) are entered successively in a guidance mode.

There are two methods of entering the standard PSD. One method is to enter the PSD on the screen; the other is to use actual vibration data sampled by the A/D converter.

The entered PSD is plotted on the graph. When entered through the A/D converter, the PSD can be edited by the graphic editor.

The data may be either new data or modified data of previous one.

RANDOM VIBRATION CONTROL PROGRAM

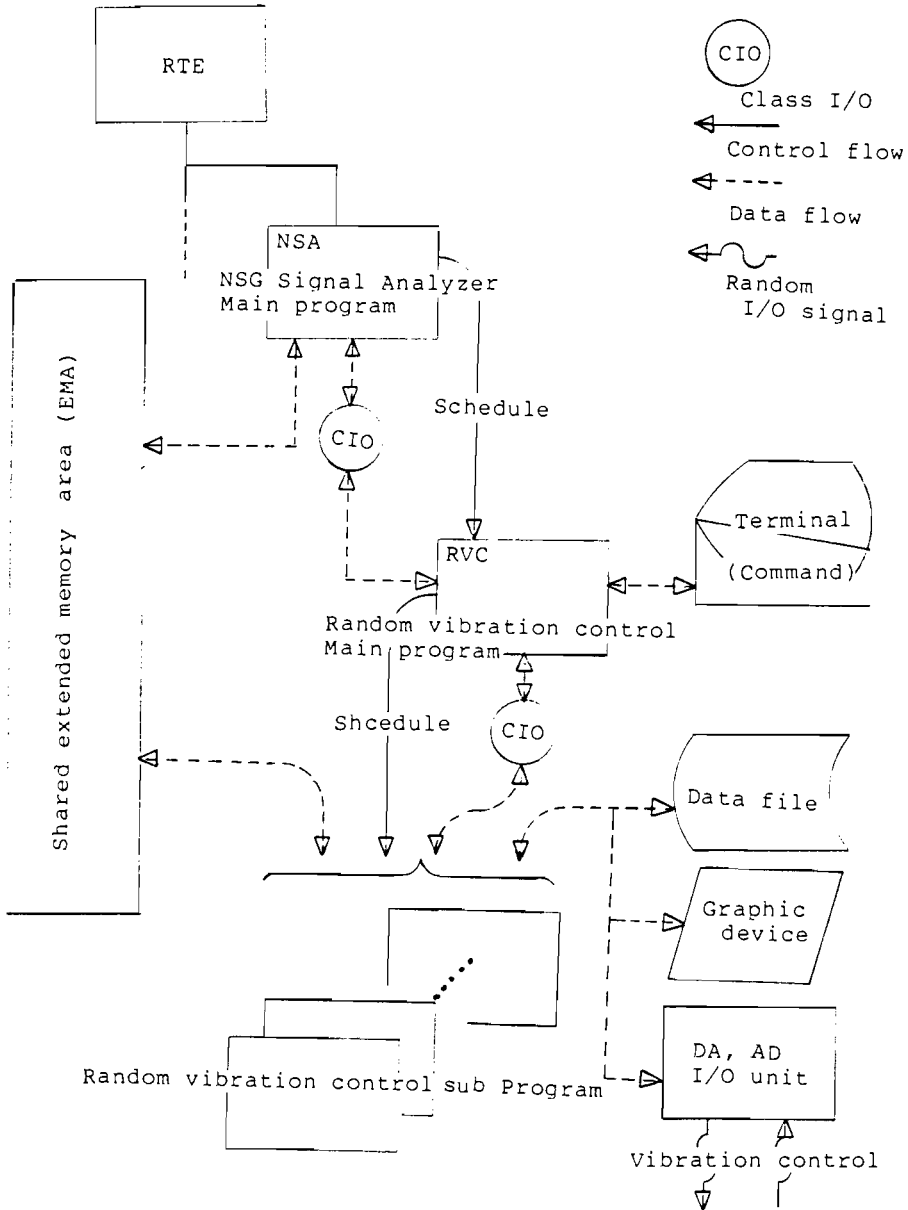


Figure 4-1 RVC and related components

RANDOM VIBRATION CONTROL PROGRAM

4.2.2 Vibration control

Vibration control starts with creation of a standard PSD based on the vibration conditions defined in advance.

The standard PSD is created according to the level scheduling (managed by level and time). This data is called a drive PSD.

After randomization and inverse Fourier transformation are performed on the drive PSD, it is sent to the vibrator through the D/A converter.

When the drive PSD is sent to the vibrator, smoothing processing, 3 σ clipping processing, D/A converter output range check are performed to prevent sudden change on the vibrator.

Exponential smoothing processing is performed on this drive PSD so that it can be saved for later comparison with feedback data.

Vibration caused by the vibrator is recorded through the A/D converter.

For data recorded on each channel, Fourier transformation, smoothing processing, then exponential smoothing processing are performed. The resultant data is called a control PSD.

The control PSD is checked with the alarm range defined by the standard PSD, and the RMS value in the vibration conditions. If the control PSD exceeds these limits, processing is stopped.

The data that successfully passed these checks is then compared with the saved drive PSD to obtain the transmissibility.

The transmissibility is reflected to the next drive PSD when it is generated, thus allowing proper vibration control.

All PSD data is managed in an extended memory area (EMA). Transfer of data is posted via class communication between function-based programs.

Figure 4-2 is a block chart illustrating the above description.

The vibration status can be checked using the monitor function.

Information from RVC is successively directed to a CRT (soft panel) allowing the operator to check the current state on the CRT.

RANDOM VIBRATION CONTROL PROGRAM

The minimum required measurement data is automatically saved when the last level starts. The operator can also save measurement data whenever he wants to do so. The saved data can be used for later report output.

Vibration control is divided into pretest and main vibration phases. The pretest is defined as a phase where a gain value is determined and the expected value of the transmissibility is obtained for the main vibration phase.

Note:

Some of the channels for recording vibration data of the vibrator can be defined as measurement channels, rather than feedback data channels; so they can be used for checking. The data recorded on these measurement channels is called a measurement PSD.

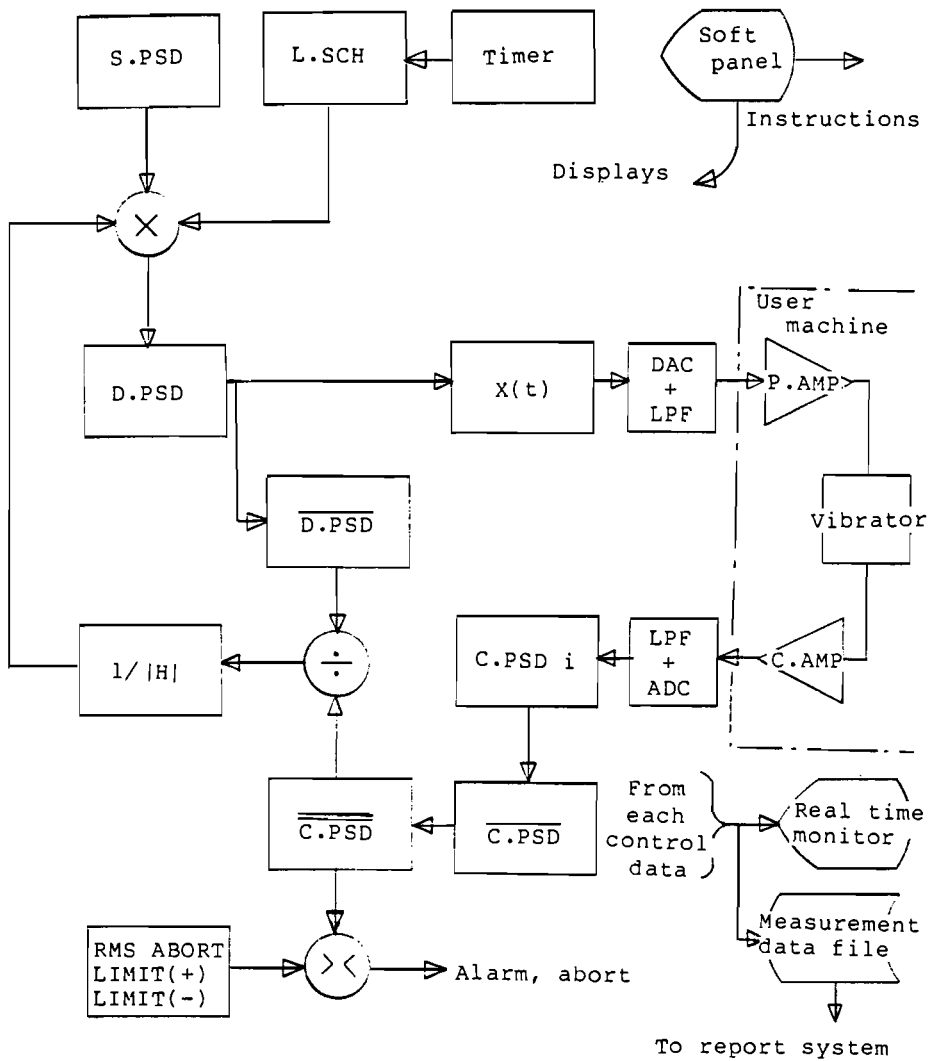


Figure 4-2 Random vibration block chart

RANDOM VIBRATION CONTROL PROGRAM

4.2.3 Report output

RVC provides a vibration data saving function so that vibration data can be used for later test evaluation.

The saved data can be plotted on the graph by the report output function.

The following reports can be output as graphs:

- (1) Standard PSD
- (2) Drive PSD
- (3) Control PSD
- (4) Measurement PSD
- (5) Time-domain signal data (drive signal)
- (6) Transmissibility ($1/|H|$)
- (7) Transmissibility (measurement PSD/standard PSD)
- (8) Transmissibility (measurement PSD/drive PSD)
- (9) Error ratio (control PSD/standard PSD)

The formats of these reports are given on the following pages.

Some other RVC functions are: index listing for file management, listing of test conditions, data file output, and data file retrieval by test ID.

ID : TEST 148 L128: 128, Fmax: 100Hz, TRUE
Sub ID: DDF: 128 (k 5, k 7), CAL: 2000mV, 6ADR: .156/

Date: 1987. 6. 4
Line: 128
Pns : .30 G-rms

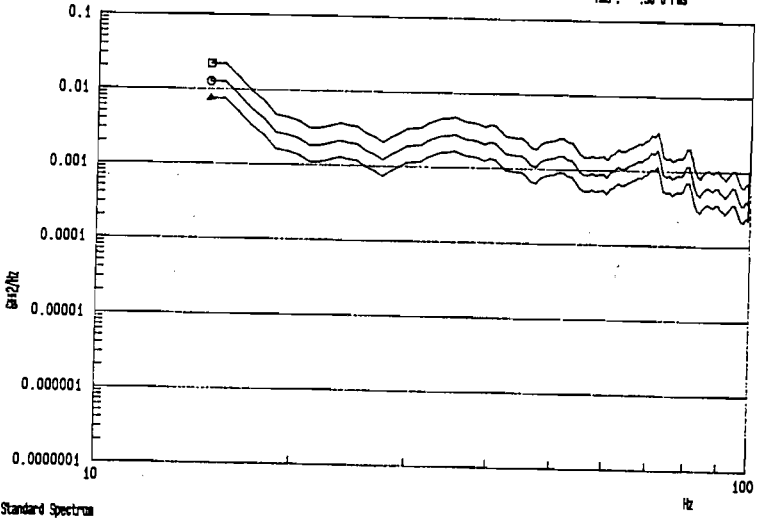
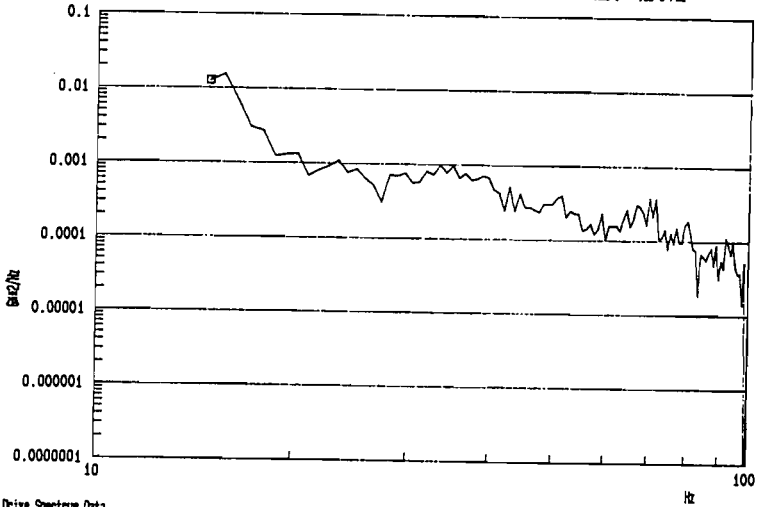


Figure 4-3 Standard PSD report format

Def : 128 (k 5 k 7)

ID : TEST 148 L128: 128, Fmax: 100Hz, TRUE
Sub ID: DDF: 128 (k 5, k 7), CAL: 2000mV, 6ADR: .156/

Date: 1987. 6. 4
Line: 128
Pns : .19 G-rms



Drive Spectrum Data
Level: -6.00 dB

Time : 739.86sec

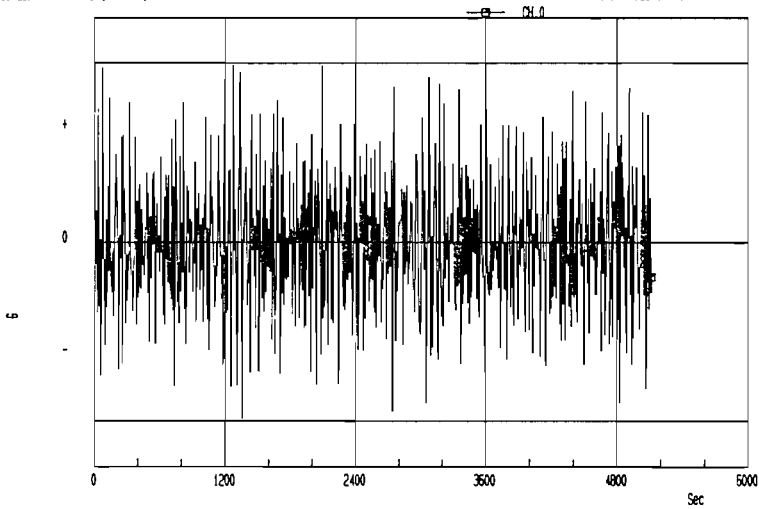
Def : 128 (k 5 k 7)

Figure 4-4 Drive spectrum report format



ID : TEST#3 LINE: 512 DOF=30 (R: 3, K: 3)
Sub ID: TRUE (-6.0dB)

Date: 1987. 2.26
Line: 512
Res: .26 6-rms

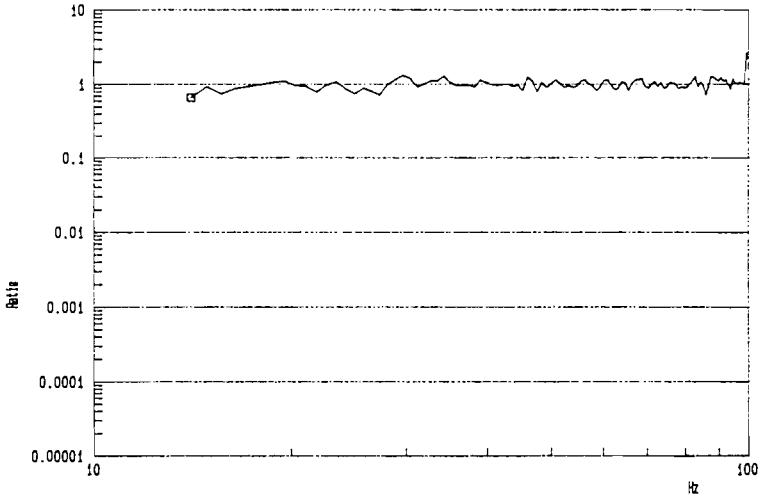


Time-Domain Signal
Level: -5.00 dB Time: 230.67sec Def: 30 (R: 3, K: 3)

Figure 4-5 Drive signal waveform vs. time report format

ID : TEST 108 LINE: 128 Fmax: 100Hz, TRUE
Sub ID: DOF: 126 (R: 5, K: 7), CAL: 2000W, GAUR: .156/

Date: 1987. 6. 4
Line: 128

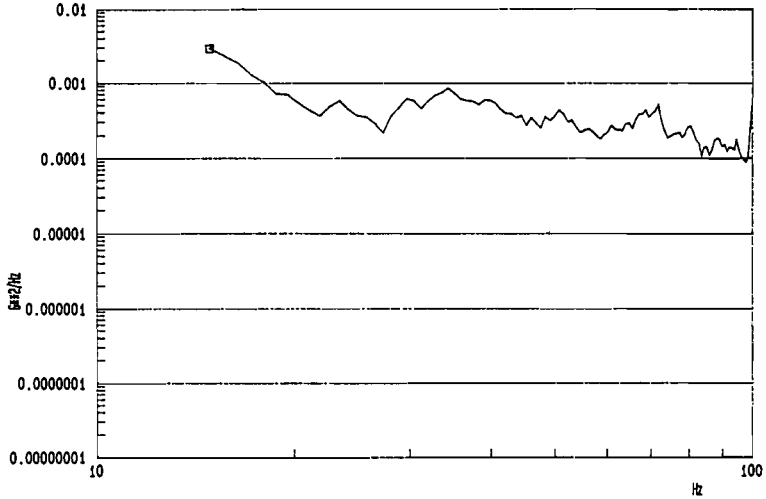


Error Spectrum (Control/S.PSD)
Level: -6.00 dB Time: 882.51sec Def: 126 (R: 5, K: 7)

Figure 4-6 Error spectrum report format

ID : TEST 148 L128 Fmax: 100Hz, TRUE
Sub ID: DEF: 126 DE 5, K 7, CAL: 2000mV, GADR: .156/

Date: 1987. 6. 4
L128
Res: .13 G-rms



Control Spectrum Data
Level: -8.00 dB

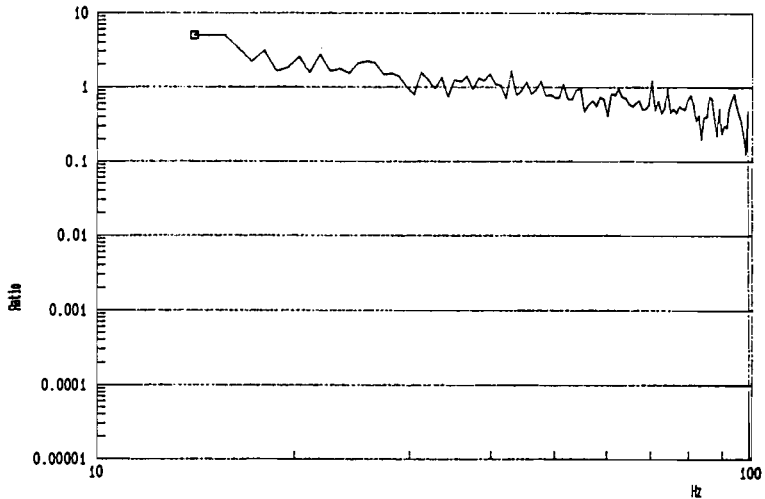
Time: 852.51sec

Def: 126 DE 5 K 7

Figure 4-7 Control spectrum report format

ID : TEST 148 L128 Fmax: 100Hz, TRUE
Sub ID: DEF: 126 DE 5, K 7, CAL: 2000mV, GADR: .156/

Date: 1987. 6. 4
L128



Transmissibility (1/1H)
Level: -8.00 dB

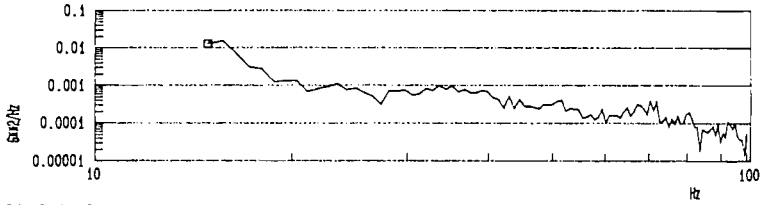
Time: 888.53sec

Def: 126 DE 5 K 7

Figure 4-8 Transmissibility report format

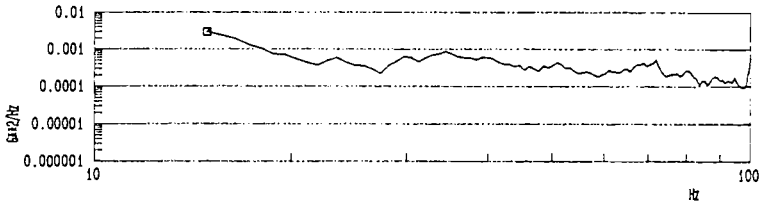
RANDOM VIBRATION CONTROL PROGRAM

ID : TEST 1dB LINE: 128, Fmax: 100Hz, TRUE Date: 1987. 6. 4
 Sub ID : DDF: 128 (K 5, K 7), CAL: 2000mV, GADR: .156/ Line: 128
 Res : .13 G-rms



Drive Spectrum Data
 Level: -6.00 dB Time: 739.06sec Def: 128 (K 5 K 7)

ID : TEST 1dB LINE: 128, Fmax: 100Hz, TRUE Date: 1987. 6. 4
 Sub ID : DDF: 128 (K 5, K 7), CAL: 2000mV, GADR: .156/ Line: 128
 Res : .13 G-rms



Control Spectrum Data
 Level: -6.00 dB Time: 852.51sec Def: 128 (K 5 K 7)

Figure 4-9 2-charts-per-page output report format

5. Specifications

Item	Specifications
Frequency ranges	100, 200, 300, 400, 500, 1 k, 2 k, 3 k, 4 k, 5 k, 6 k (Hz)
Number of lines	64, 128, 256, 512, 1024
Control signal	True random/pseudo random
3 Σ /clipping	Available
Loop time	seconds
Saving of standard spectrum	100 types/file
Creation of standard spectrum	Keyboard (level/slope specification) A/D converter input (editing allowed)
Setting of vibration conditions	Interactive mode
Saving of vibration conditions	99 types/file
Real time monitor	Enabled (on graphic terminal)
Vibration control	Close/open Mean value/peak hold value
Control input	± 10 V (12-bit resolution)
Control output	± 10 V (16-bit resolution)
Number of controlled vibrators	1 (2, optional)
Number of control input channels	8 max.
Number of measurement input channels	7 max. (up to 8 channels together with control input channels)

(Con'd)

RANDOM VIBRATION CONTROL PROGRAM

Item	Specifications
Report output Report type	Graphic terminal, X-Y plotter Standard spectrum, abort limit (+, -), control spectrum (exponential mean, loop mean, composite, channel-based), drive spectrum, drive waveform vs.time, transmissibility spectrum, error spectrum
Report form	1 chart per screen (overlay enabled), 2 charts per screen (overlay enabled)
Y-axis scale	Automatic/specification, Linear/logarithm/dB

RANDOM VIBRATION CONTROL PROGRAM

An Implementation Of LISP
Dan Felman
Hewlett-Packard Company
2 Choke Cherry Rd.
Rockville MD 20850

Introduction

Preface

This paper will attempt to describe my experiences while working with the LISP interpreter obtained from Interex's CSL swap tape.

The main reason for tackling a study of the interpreter and eventual enhancements was in the availability of its source code, written in PASCAL. This was beneficial in many ways. For instance, if a required feature did not exist, I could add it in. If a function did not work, I could attempt to fix it. If I did not like how it operated, I could change it. And best of all, I could learn a new language from the inside out.

Although this paper is relatively brief, I will try to convey what was like working with different PASCAL compilers and how I ported the code across several machines.

In addition, I will describe the operation of the interpreter in some detail for those readers who, like myself, enjoy the challenge of tackling a change here and there.

The code for the VECTRA, and HP 1000 implementations should be submitted at this conference.

1.1 How it all begun.

As I registered for my first AI class, I began to wonder how and where would I get a hold of a LISP system to do my homework and experiment with. Well, the answer quickly came in the form of the swap tape at the INTEREX's conference in Washington D.C., where I worked with the organizing committee representing local Hewlett-Packard needs.

The swap tape contained a couple of LISP programs. One of them was the start of my LISP learning.

The LISP interpreter was written in PASCAL and fitted neatly into a 32 page RTE partition. Since then, the code has grown to about 3 times its original size, taking 3 code segments in RTE-A and SGMTR with MLLDR in RTE-6. I must observe that I have not try to compile LISP under RTE-6 for quite some time, so I am not aware of how well it would port right now.

1.2 Why PASCAL.

The reasons for writing a LISP interpreter in PASCAL, as opposed to FORTRAN, BASIC, etc, are due to the facilities inherent in the language itself. LISP is a language with heavy use of pointers. Atomic symbols, such as variables, names, strings, etc, are stored and managed by the LISP run-time interpreter. These items are very dynamic and their data types may change often. Since one of PASCAL's fortes is pointers, it becomes an easy choice, due to the use of record structures that may be manipulated in the (run-time) heap. Another suitable language is C, but my knowledge of C is rather feeble, and I could not takle translating 1500 lines of code to C without a deep understanding of the language.

Since the implementation submitted to Interex was for RTE, I ended up working many a weekends on the machine I had available at my office, pouring over listings while trying to understand how the interpreter worked.

As the code grew, I had to switch to CDS (on the A900), and when compile times became increasingly large, I ended up augmenting PASCAL's workspace to minimize VMA swaps. By the beginning of 1987, this workspace was up to about 400 pages of EMA, and even then a compile would take over 5 minutes. Also, the PASCAL compiler I had to used was the non-CDS version, as the CDS version could not handle the number of routines and recursion with its limited stack. In

retrospect, this should simplify porting to RTE-6 system, since it is the only kind of compiler available in that operating system.

Now back to school. By the end of the first course, I seriously begun to think about a PC. If I could have one at home, I could spend more time on the interpreter "and" with the family. After a lot of thought, and a good size loan, I bought my own VECTRA to which I added the Turbo PASCAL compiler. It did not take long to move my LISP code from the A900 to the VECTRA, and what a difference it made. A compile with floppies takes well less than a minute!, and this is basically the same amount of code compiled on the A900. Oh, well, this is what technology is all about, isn't it?.

Translating the code from HP PASCAL to Turbo PASCAL was not trivial since anything but simple "read" or "write" statements is really implementation dependant. To begin with, the code had to be split into several "include" files so that Turbo would accept it. Turbo PASCAL's editor can only handle files as large as around 62 kbytes, and the LISP interpreter was much larger. Luckily, (and so far) the code produced by this compiler still fits within a 64 kbytes segment (another Turbo limitation). On the other hand, the heap can be as large as the maximun memory recognized by MS-DOS, or abour 640 kbytes. Since the LISP user code and data are maintained in the heap, relatively large programs can be written under Turbo. In any case, include files, at the end, helped manage the overall structure the code. It does seem that further work on this compiler will not be possible unless Turbo PASCAL is enhanced with a new "large memory model" version.

2.1 Some Differences

In order to move the PASCAL code between the HP 1000 and my VECTRA, changes had to be made to conform to the specific compiler implementations. I will now list some of these differences and how I partially worked around them.

- ⊕ Opening a file for input with the HP compilers is as follows:

```
{ 'f' is a variable of type "text" }
reset ( f, 'file_name');
```

whereas in Turbo one has to assign first

```
assign ( f, 'file_name');
reset ( f );
```

- ⊕ In LISP, one must allocate memory for atomic symbols with a variable amount of bytes, because allocating a fixed maximum amount is rather wasteful of memory resources. PASCAL does not allow this, once a record structure has been setup to be used by the procedure "new". Therefore, operating system dependent calls must be made, such as these on the HP 1000:

```
{for allocating bytes in the heap (ema)}
{ 'p' is a record pointer of any type }
{ 'number' is the amount to allocate }
pas.new2 ( p ,number );
```

```
{for releasing bytes to the heap (ema) }
pas.dispose2 ( p, number );
```

for the HP-UX machines, these calls become:

```
{for allocating bytes in the heap }
p := malloc ( number );
```

```
{ for releasing bytes back to the heap}
free ( p );
```

and for Turbo they are:

```
{ for allocating bytes in the heap }
getmem ( p, bytes )
```

```
{ for releasing bytes back to the heap }
freemem ( p )
```

- ◆ Another difficulty is access to run-time parameters.

In the HP 1000 and the HP9000/500 series HP-UX, one can use

```
{on the 500, the case is important and must be
 spelled as shown here}
Pas.SParameters
```

on the HP9000/800 series, one must use two calls:

```
{ sets up the run time parms }
argc

{ retrieves single items from the list }
argn
```

and on Turbo, it is of course different:

```
{ retrieves single item from the list }
getparm
```

- ◆ Where possible, the standard procedure "new" is used. All HP implementations (I believe following the standard) require the variant part of the record to be specified with the "new" call. In Turbo, the variant can not be declared with "new".
- ◆ Also, declaring the Include files is different between the HP compilers and Turbo PASCAL.

2.2 System dependent include files

To avoid massive changes when moving the interpreter between environments, I created separate Include files as follows:

```
hp1000.inc
hp800.inc
hp500.inc
turbo.inc
```

which only requires a simple edit in the main to bring the corresponding (and proper) file as needed.

For more detailed information on the internals of this LISP interpreter, refer to the following section.

3.1 Interpreter files

The code for the interpreter is broken up into files. Each file more or less corresponds to a chapter as defined in Steele's book.

The following files exist today:

lisp.g	contains global declarations for the PASCAL program
lisp.pas	with the main program plus auxiliary routines
predicates.pas	with LISP predicates
control.pas	has APPLY plus other control functions
symbols.pas	contains symbol manipulation functions
numbers.pas	
sequences.pas	with sequence functions
lists.pas	lists functions are defined here
evaluator.pas	contains EVAL plus some other functions
streams.pas	has stream i/o functions
io.pas	standards LISP i/o functions
fileio.pas	contains file i/o access functions
errors.pas	has the error functions

Each of the ".pas" files also has an associated ".inc" file containing the forward declarations for PASCAL.

3.2 Data structures

In order to make some sense out of this implementation and to try to understand the listings, a review of the data structures used must be done. Knowing the data structures will make further modifications somewhat easier.

A node is the basic data structure used by LISP, and is addressed by a pointer type nodep. Node is a Pascal record structure consisting of a mark flag (used by the Garbage Collector), and a variant part whose components mirror LISP data types and are described as follows:

- conscell a LISP cons cell consisting of a CAR part and a CDR part,
- symbol a LISP symbol with a print name of up to 80 characters, a value slot, and a property list slot. A trace flag is also in the object's structure, as well as a STATIC flag for constant symbols, Figure 1 shows a Symbol data structure.
- sstring a LISP string node (a constant), each also of up to 80 characters,
- stream a LISP stream for input/output, not fully implemented, with a pointer to 1 of 4 allowed text files,

Figure 2 shows a Symbol data structure.

- fltnode a LISP floating point number with precision limited by the Pascal compiler being used,
- intnode a LISP integer node, also limited in its precision by the Pascal compiler

A listing of this basic data type is shown in Figure 3.

A second data structure is used to link all nodes in the heap for garbage collection purposes. This structure consists of a two pointer record named stacknode, and accessed by a pointer type stackpoint. The first pointer is used to link other stacknodes while the remaining one links nodes.

The Garbage Collector (GC) sweeps through LISP memory by way of stacknodes. The linked list is headed by TOP and its end is pointed to by BOTTOM, meaning that TOP points to the first node allocated by the interpreter, and BOTTOM points to the last one.

At startup, LISP knows of many symbols, including function names and global variables. File "lisp.g" contains a listing of the symbols known to LISP and setup during the initialization phase. These LISP symbols can be easily flagged because of the suffix SY. For example, the CAR

```

SYMP      = ^SYMBOLNODE;
SYMBOLNODE= packed record
            VALUE: NODEP;
            PLIST: NODEP;
            TRC,STATIC: boolean
        end;

```

Figure 1. The data structure for a Symbol.

```

STMP      = ^STREAMNODE; (* for stream i/o *)
STREAMNODE = packed record (* input buffer *)
            FNUMBR: INT; (* file number used *)
            CPOS: INT; (* current line pos *)
            CLINE: LINETYPE (* line buffer *)
        end;

```

Figure 2. The data structure for a Symbol.

```

OBJTYPES = (CONSCCELL, SYMBOL, SSTRING,
            STREAM, FLTNODE, INTNODE);

NODEP     = ^NODE; (* ptr to node *)

NODE      = packed record (* node record *)
            MARK: boolean;
            case OBJECT: OBJTYPES of
                CONSCCELL: (CARP: NODEP;
                            CDRP: NODEP);
                SYMBOL: (NAMEP: STRNGP;
                          SYM: SYMP);
                SSTRING: (SNAMEP: STRNGP);
                STREAM: (TNAMEP: STRNGP;
                          STM: STMP);
                FLTNODE: (RVALUE: real);
                INTNODE: (IVALUE: integer)
            end; (* node record *)

```

Figure 3. A LISP 'node' type

symbol's name is CARSY, and the NULL symbol's name is NULLSY.

3.3 The environment

Only one environment is maintained. This environment is akin to the package referenced in Common LISP. If it were called a package it would probably be the 'LISP' package, since all the LISP primitives exist in it. Packages is a possible addition.

All symbols are linked in this LISP environment and headed by a global variable named LISPLIST. This list is really a LISP list with cons cells allocated to link the symbols. Therefore, if one were to print LISPLIST, it would display a list consisting of all known symbols, such as function names, variables, constants, etc.

As I explained above, all nodes, whether conses, symbols, strings, or whatever, are globally linked by stacknodes. In essence this linked list is similar to what Pascal maintains in the heap, and allows data structures to be deallocated by the garbage collector.

Because all objects are linked in via stacknodes, it follows that cons space is mixed with symbol space in PASCAL's heap. This has some pros and cons. On the pros side, one does not have to limit the number of cons cells, or string space a program may use. This limit is imposed only by the amount of memory existing in the machine (or allowed to the current process in a multi-tasking implementation such as UNIX) and that Pascal may manage as heap. What this means, is that the programmer is free from worrying about string space usage, versus how much cons space is allowed. In other implementations, even on LISP 1.5, memory space is split into cons space, and symbol (and string) space.

On the other side of the coin, this management scheme leaves all heap maintenance to the whim of the Pascal compiler. If one considers that allocation and deallocation of heap space is not defined in the Pascal standard, then whether space one deallocates may be reused (some Pascals hold these in a free list, other do not) becomes implementation dependent and portability of the code is drastically reduced. More about this, in the Analysis section.

Several routines are involved in creating nodes and linking them into the workspace (environment). MAKEINT and MAKEFLT create numeric nodes, MAKESYM creates symbol nodes, MAKESTR deals with string nodes, MAKESTM handles streams, and CONS allocates cons cells. All of these internal functions use GETNODE to allocate space for type node.

GETNODE is responsible for allocating heap space and linking the acquired node into the stacknode space at BOTTOM.

Additionally, INTERN enters a symbol node into the LISP symbol table headed by LISPLIST. Again, LISPLIST could just as easily be renamed LISP and be declared a package. Of course, implementation of a package system is more involved than this, but the similarity of the structures remain.

Symbols, once INTERNed, are never deallocated. Common LISP provides a means of deallocating symbols with the function UNINTERN, but it is unsupported in the current version. If the symbol becomes unbound, it is merely marked as such, and remains linked into LISPLIST.

In order to understand how LISP maintains the environment, let's dissect 'intern'.

3.3.1 intern

When a symbol is declared into LISP, INTERN is called to link it in the symbol list. INTERN does one of two things. If it already exists, a pointer to that node is returned. Otherwise, it uses MAKESYM to create a symbol, links the new symbol into LISPLIST with a call to CONS, and a pointer to the new symbol is returned as INTERN's value.

To get a better grasp on the data structure linkages, let's see what the environment looks like after the first symbols are entered into the LISP system.

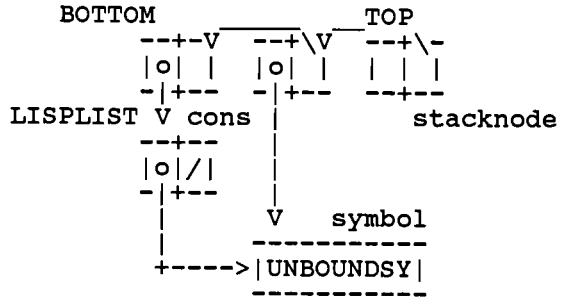
The initial linking is done by the internal routine INIT, which pre-initializes the environment as follows:

```

                                BOTTOM
                                TOP
                                ---+---
                                | | / |
                                ---+---
LISPLIST -> NIL

```

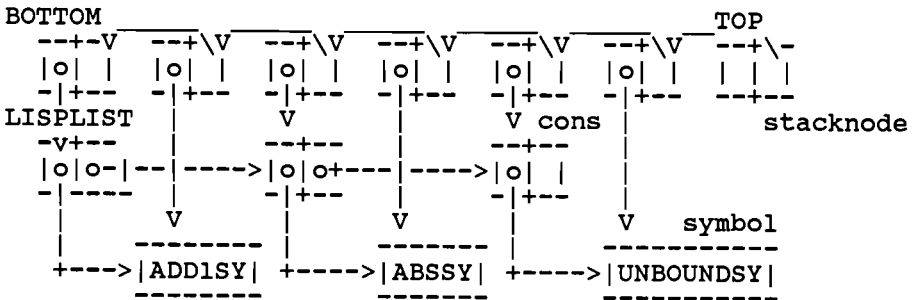
The first symbol declared is UNBOUNDSY (with a print name of "***unbound***"). After INTERN links UNBOUNDSY, the environment looks like this



where the top row has nodes of type stacknode and everything else is of type node. One can see that starting with TOP, at the right, the first usable node has a pointer to UNBOUND, but since INTERN adds to the symbol table with a CONS call (LISPLIST is the second argument), a second node is allocated (by CONS) and added to LISPLIST. A print of LISPLIST would now yield

(UNBOUND)

After two more entries into the symbol table, the linked list looks as follows



As shown, stacknodes are used to hold LISP nodes, and are needed for Garbage Collection sweeps. If we were to print the list headed by LISPLIST at this point we would see

(l+ ABS **unbound**)

where 'l+' is the print name of ADD1SY, and '**unbound**' is the print name of UNBOUND.

Let's now take a closer look at GETNODE, a very important routine.

3.3.2 getnode

GETNODE's only parameter is the object type to create. Based on this type it allocates a node in the heap (currently using Pascal's new procedure), and sets the right values in the appropriate object slots.

For example, if CONS were to call GETNODE to allocate a cons cell, GETNODE would first allocate a cons type node out of Pascal's heap and would then initialize the pointers (CAR and CDR) to nil. CONS is responsible for linking both pointers to the proper values.

Before returning to CONS, GETNODE allocates a stacknode, links the new node into the global environment headed by TOP, as shown graphically above, and points BOTTOM to the new tail of the list. The markers TOP and BOTTOM then point to the first and last nodes, in the LISP system.

3.4 LISP Startup

Before the top level loop is entered, the initialization of the environment must be done. This is accomplished with a call to the internal procedure INIT, which sets the symbol list head, LISPLIST, to 'nil'. This is graphically represented in the first figure drawn above. INIT then enters, via INTERN, all known LISP symbols into the main (and only) package. Or should I say environment. I will later on describe how a Common LISP 'package' structure may be implemented.

Constants, such as T and NIL, are set to STATIC. INIT also initializes global variables needed by the interpreter. For example, a stacknode node is allocated, TOP, to head the global link list.

Additionally, an error message array is filled in to be used by error reporting routines. Other global variables are also initialized, and some input/output symbols are set to point to the user console.

INIT finally prints the "Welcome to LISP ..." message seen at startup.

3.5 The Top Level

As most LISPs, the interpreter itself follows the READ/EVAL/PRINT loop cycle. Therefore the Pascal main consists of a do-forever loop which reads an expression, evaluates it and prints any resultant value.

The terminating condition is to evaluate the function (EXIT).

After INIT finishes the environment initialization, the interpreter checks for arguments passed to this run. If a name was passed, it attempts to load this file (only one allowed) with the LOAD function. At the end of the load, and if no terminating condition was encountered, the input stream is reset back to the user console, and a prompt is issued. No prompts are displayed when inputting from a file.

LISP then initiates the top level READ/EVAL/PRINT loop.

Let's now examine these the pieces of the loop individually.

3.6 The LISP reader

The reader is headed by the function READ. READ takes an optional input stream, and if present, it resets the input to the argument and issues a read from that file (or device). Otherwise, a prompt is issued by a supporting routine.

READ assumes that if an argument is passed, OPEN was used (maybe via LOAD) to open the text file for read access. It is an error if this is not the case.

Function INPUTEXP is actually used by READ to read a LISP expression from the input stream. Let's see how INPUTEXP works next.

3.6.1 inputexp

INPUTEXP is a recursive descent parser which understands LISP forms as objects and lists. It understands the backquote syntax, as well as the ' (quote), # (pound), and , (comma) macros. It also handles single escape () characters.

This routine is really a subset of the full Common LISP reader, which is not surprising since the Common LISP

description is a very sophisticated program in its own right.

The token parsing is done by INPUTATOM, and the physical reads are handled by NEXTCHAR and NEXTLINE.

3.6.2 inputatom

This function is also a recursive descent parser that finds tokens in the input string, and creates LISP nodes accordingly. It can parse integers, floating point numbers with a decimal point, symbols and strings (delimited by the double-quote character). If the token being read is a symbol, in enters it into the current package (remember, there is only one in the current implementation) by passing the name to INTERN.

3.6.3 nextchar

NEXTCHAR returns the next available character in the input buffer. If the buffer is at the end, NEXTLINE is called to get a new buffer (a line of input in this case). NEXTCHAR then upshifts the character just read in, unless the escape character was previously seen.

NEXTCHAR also avoids reading past the comment character (;). If one is found in the input buffer, a new line is requested from NEXTLINE.

3.6.4 nextline

NEXTLINE reads a line of data from the current input. If the variable and handles the read accordingly. Otherwise, it prompts the user for input with the character '>'.
'>'

3.7 The Evaluator

EVAL is the primary evaluation function and is called by the main to return a value from the evaluation of a form. EVAL is passed the expression read by READ.

EVAL does the following:

- if the argument form is a
- . constant: it is returned

```

. ')': an error message is displayed and a break loop is entered
. atom: a value is found among
    the current environment    (dynamic in nature)
    the global environment    (by its 'value' slot)
if neither is found, an 'unbound error' message is issued
and a error loop is entered
. list: then
    if the CAR is an atom then
        if trace is on for the function name, the TOUT function
        is called to output a trace
        if EVAL recognizes the name, then
            it handles it directly,
        else
            it checks for a user defined function definition, and
            for a user defined macro definition,
        if no success is achieved so far, APPLY is called with
        the function name, and all the (evaluated) arguments.
        If trace was in effect, an appropriate output takes place,
        again, by calling TOUT.
    else
        NIL is returned. No error checking is currently done on a
        a form such as ((setq n 5))

```

The environment used by the top level is initialized to NIL. This means that LISP has not yet seen any user variables being used.

Originally, EVAL was called with two arguments, the form to evaluate, and the environment (set to NIL by the top level loop). EVAL then passed this environment to all functions that needed access to it. APLLY was one of them.

In the current implementation, the environment is maintained globally in a stack-like data structure headed by the symbol STACKF. The top of the stack is the current environment and is pointed to by the symbol ENV. Therefore, any function in need to access elements of the current evaluation stack would look at ENV for values. I believe

this is more efficient, less time consuming and requires less Pascal stack space. PUSHENV and POPENV are used to add and take off from the environment stack. They are described below.

When EVAL finds a form calling for a user defined function, LAMBDA is used to obtain the value of the function call. The arguments of the LAMBDA call are evaluated a priori. By the same token a user defined macro call is handled by MACROEXPAND, which gets the unevaluated form as an argument.

3.7.1 lambda

LAMBDA gets the evaluated arguments from EVAL and calls BIND to add them to the front of the current environment. It then asks PUSHENV to enter the new environment as the current one. We will take a look at this in a moment.

After the new environment is in effect, LAMBDA executes an implicit PROGN with the body of the function, and on return from PROGN, the old environment is restored with POPENV.

LAMBDA returns to EVAL the result of the PROGN call.

3.7.2 macroexpand

MACROEXPAND gets entered with the form as typed by the caller, and uses APPLY to expand the macro. Eventually, backquote, comma, and/or commaat templates are handled by EVAL.

3.7.3 pushenv

PUSHENV is called with the environment to make current. It first sets a global variable (ENV) to it so that all other functions may use it. This new environment is then linked on top of a stack-like data structure headed by STACKF. STACKF is set NIL by the top level loop.

PUSHENV returns the previous environment, which may be used by POPENV on a successive call.

3.7.4 popenv

The argument to POPENV is (and should be) the environment to make the current one. It should naturally follow PUSHENV after some evaluation has taken place. POPENV would normally be called with the linked environment returned by PUSHENV.

It sets ENV (the current environment) to the restored environment, and unlinks the top from STACKF.

POPENV does not return a value.

3.8 The LISP Printer

PRINT is a relatively short section of the interpreter, as it only works with a subset of the full implementation. FORMAT is not currently available. PRINT is, so far, the only way to output both to the console or to a stream file.

In essence, PRINT recursively explores a form (if not an atomic value), while printing the representation of the values.

A pretty-print mechanism is built-in within PRINT, and its effect may be turned on and off by setting the variable *print-pretty* to T or NIL, as desired

3.9 The Garbage Collector

Garbage Collection is handled by GC. GC calls MARKIT to mark all the cells pointed to by the environment stack, headed by STACKF, and the global symbol list, headed by LISPLIST. Once all known nodes are marked, FREEUP is called to sweep every node, from TOP to BOTTOM. Any node not marked is returned to the heap, while marked nodes are unmarked.

4.1 ANALYSIS

What is interesting about this interpreter is that there is a great deal left open for expansion. There are many things that not only do not exist as described in Steele's Common LISP book, but in a few cases what does exist does not fully conform to the standard. In the paragraphs that follow I will describe the work that needs to be done.

On the other hand this is a useful program, if nothing else, at least for the learning potential. Since October of 1985 I have added about 1600 lines of code (doubled its size), and learned a great deal about LISP. Let's now take a look at the major tasks.

1. Lexical vs. Dynamic environment. I do not yet fully understand the methodology needed to implement a Lexical LISP. The version running today has a dynamic environment. The effort needed to create a Lexical environment is unknown to me at this time.
2. Modify the internal data structures to support separate cons and symbol table spaces. This should help number 3. The effort required should not be great. What this means is that the maintenance of the heap could be taken away from the Pascal compiler and put into the interpreter. In this case, all cons space would be allocated a priori, and put into a free list. As needed, these cells would be unlinked from the list and used. The garbage collection phase could then relink unused cells into the free list. The few routines that access Pascal's heap would be modified as well as the garbage collector.
3. Fix garbage collection. It seems to only work part time. I believe that by working on number 2 (above), should go a long ways towards clearing the collector. The problem seems to be that an environment is not properly saved during function calling, so that the collector deallocates a link it is not supposed to. Of course, the problem may not be with the garbage collector whatsoever, but may lie somewhere else. I need to study this in some more depth.
4. Create a package system. Although it may seem an awesome task, I believe it should not be such. In principle, the linked list headed by LISPLIST is

really the 'LISP' package. New lists could be created and maintained without a lot of difficulty. The reader and the evaluator would have to be modified to accept packages as new data structures. To start this enhancement, one could create a 'KEYWORD' package to handle LISP keywords. The capability needed to support just one more package would give a good idea as to the amount of work a whole package system would require.

5. Optimize the lookup of intrinsic functions by possibly using hash tables, or other suitable techniques. This would require some study, and a need to create new data structures. How much effort would be required is somewhat difficult to discern, but it should be less than coming up with a Lexical LISP. Since this is a performance issue, although useful, it should be left for last.
6. Fix the interpreter so that strings can be manipulated as such, are declared as constants (so that they can not be assigned to), and other sundry fixes. Effort for the fix is not great, but all the additional functions and operators for string manipulation may take some time.
7. Expand the LISP reader and printers, and add FORMAT. It will also require quite a bit of effort just to add FORMAT. Steele dedicates multiple pages in his book to this function.
8. Fully implement the numeric functionality (chapter 12, Steele) A lot of work, if one considers all the mathematical operators Common LISP is honored with.
9. Support Hash Tables (chapter 16), Arrays (chapter 17), characters (chapter 13), structures (chapter 19), and streams (chapter 21). This is major work.
10. Add a full file system interface (chapter 23). This becomes machine dependent and would require a lot of sweat and tears to implement in a portable manner.
11. Fully support all data types. Again, lots of work.

12. Add all the lacking control structures, such as BLOCK, CATCH, THROW, etc. This is relatively important, and with some study, it should not take an awfull lot of time. The effort put into this task should be well rewarded.
13. Clean up the error detection and correction features. Some effort will be required as most functions and predicates have to detect user errors.
14. Add trace feature beyond those already provided, such as STEP and others. This should not take a great deal of effort, although the implementation of a full backtrace mechanism would not be simple.
15. Implement all functions not described by 1. to 14. above as required by Common LISP. Effort unknown.

Effective Usage of EMA/VMA in FORTRAN Programs

Wayne R. Asp
Hewlett-Packard Company
2025 West Larpenteur Avenue
St. Paul, Minnesota 55113

INTRODUCTION

The ability to handle large amounts of data is an important feature of the HP1000 computer family. The management of large data arrays in RTE is implemented as Virtual Memory Area (VMA) and allows data to reside on disc. When needed, the data is swapped into memory. Additionally, a subset of VMA, Extended Memory Area (EMA) allows the data to reside in memory only. EMA/VMA is important to the HP1000 family as it extends a program's data address range well beyond the 64Kbyte limit inherent in 16 bit architectures. Most applications use EMA/VMA for manipulation of large amounts of data and/or for storage of data readings taken from some measuring device.

EMA allows addressing of large data arrays, up to 2 megabytes per user program. The data resides in memory at all times and is never swapped to disc. This is important for time critical applications as the data arrays are always available. Also, EMA data can be shared between multiple user programs.

VMA employs a demand paged virtual data scheme to address up to 128 megabytes of user data arrays. The user configures the size of the overall virtual data area on disc and the amount of virtual data that is to reside in memory at any given time. VMA is mainly used for extremely large data arrays.

To use EMA/VMA in FORTRAN, the user must declare the arrays using the \$EMA compiler directive. This informs the compiler to generate the appropriate instructions to map EMA/VMA data into logical memory before addressing the data. While totally transparent to the user, EMA/VMA data access incurs a performance penalty of five to ten times that of standard non-EMA/VMA data access.

EMA/VMA: A User's Perspective

Where does this performance penalty come from? How can the user minimize the impact of this penalty on FORTRAN programs? Is it possible to control the EMA/VMA area programmatically?

To address these questions, the user must have a clear understanding of how EMA/VMA functions, from a general technical viewpoint. The example given here shows a typical FORTRAN program fragment using EMA/VMA.

```
FTN7X,L
$EMA /XXX/
```

```
PROGRAM EXAMPLE
COMMON /XXX/ I(1024),J(1024,1024)
.
DO n=1 ,1024
  I(n)=n
  DO k=1 ,1024
    J(k,n)=n
  ENDDO
ENDDO
.
.
```

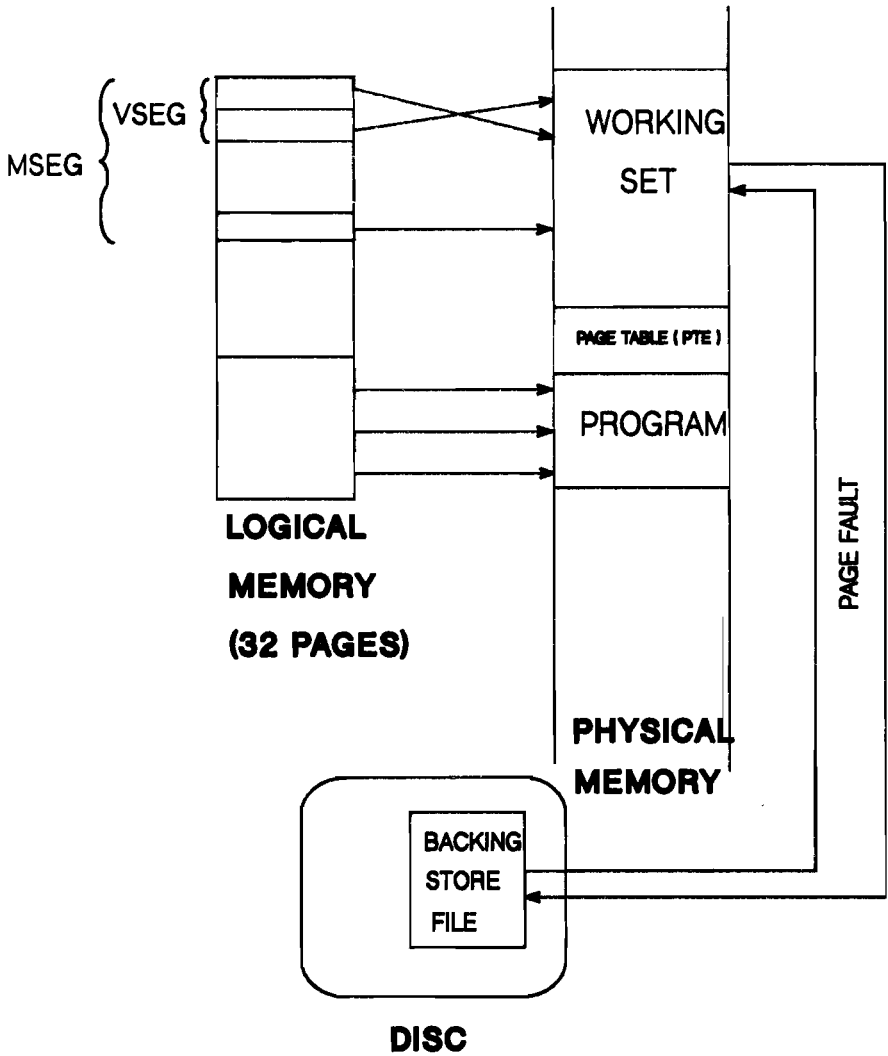
Since EMA is implemented as a subset of VMA, this discussion will focus on the VMA implementation. In the program fragment given, what happens when the arrays I and J are accessed in the DO loops?

Figure 1 shows the VMA memory mapping scheme from a user's viewpoint. The user program resides somewhere within physical memory, along with the VMA working set, whose size was specified at Link time. Also included in the program's partition is the Page Table (PTE). The page table is added unconditionally by Link to any EMA/VMA program. The PTE describes what is contained in the working set at any given point in time.

The user's logical memory is normally set up to point to the physical page numbers where the program code resides. The VSEG mapping registers 30 and 31, are used by the EMA/VMA firmware to map pages of data from the working set into logical memory. The MSEG is only used if the user is calling EMA/VMA mapping subroutines directly, or using VIS subroutines.

When the statement "I(n)=n" is encountered by the FORTRAN compiler, it generates a call to an EMA/VMA mapping subroutine (in microcode) to map the page containing the requested data element into logical memory. This will be referred to from now on as "the microcode". The microcode uses the PTE to determine if the page containing the element is currently resident in the working set. If not a VMA fault has occurred and a call is made to RTE to bring the needed page into the working set from the backing store file.

Once the page containing the element is resident in the working set, it is mapped into page 30 of logical memory. Now the entire process is repeated to map in the "spillover" page, or the page following the page just mapped into register 30. This is done to assure that the entire data element is mapped into logical memory. For instance, the data element might be of type REAL*8, with the first word located at 1023 on one page, and the other 3 words located at 0, 1 and 2 on the following page. The spillover page is mapped into page 31 of logical memory.



VMA MEMORY MAPPING

Figure 1

Once the microcode has completed the mapping process, it returns to the program the logical address of the requested data element. The program then accesses the data.

After looking at the basic EMA/VMA scheme, there are four important points that should be noted:

- 1) The mapping scheme is totally transparent to the user. No additional programming is required to use it.
- 2) Because no additional code is required, "standard" FORTRAN can be written, making the code more portable to other machines.
- 3) FORTRAN calls the EMA/VMA microcode once for every reference to an EMA/VMA data element. It is faster to map (or re-map) the pages containing an element than to check if the pages are already mapped in to logical memory.
- 4) The EMA/VMA microcode called from FORTRAN will always automatically map the spillover page into logical memory.

In general, the performance penalty associated with EMA/VMA usage comes directly from the two latter points, though the microcode and disc accesses to the backing store file also contribute.

Passing EMA/VMA Arguments to Subroutines

Many programmers are confused and frustrated by trying to pass EMA/VMA parameters to and from subroutines. Since EMA/VMA addresses are 32 bit and all other addresses are 16 bit, strange things happen in a program that inadvertently mixes the two together. There are three methods that the programmer can use to pass EMA/VMA parameters:

- 1) The subroutine expects a 16 bit normal address for a parameter, but the programmer wants to pass a single data element from EMA/VMA. In this case the EMA/VMA data element can be enclosed in an extra level of parentheses, for example (I(n)). This instructs the FORTRAN compiler to make a temporary local copy of the EMA/VMA data element and pass the copy to the subroutine as a 16 bit address. The drawback with this method is twofold. First, only one data element can be passed this way, no arrays. Second, the subroutine cannot modify the data element, as the copy is actually passed, not the original data element.
- 2) The subroutine uses the EMA statement to declare arguments and/or local variables as 32 bit addresses. The EMA/VMA data element or array can be passed by reference directly to the subroutine. The subroutine can directly modify the data element(s). The drawback with this method is that data elements with normal 16 bit addresses cannot be passed as an argument to a subroutine expecting a 32 bit address declared by the EMA statement.
- 3) The program uses EMA transparency. Using this option, all arguments are passed using 32 bit addresses. 16 bit addresses are converted automatically to 32 bit addresses before they are passed to a subroutine.

Using EMA transparency, a subroutine does not have to presuppose whether 16 bit or 32 bit addresses are passed as arguments. The drawback to this method is that all calls to HP Library routines, utility subroutines, RTE calls, etc. must be explicitly declared as non-EMA/VMA so that 16 bit addresses are generated in all references to them. Otherwise the compiler will generate a 32 bit address reference when the routine is expecting a 16 bit address. Unspecified results will undoubtedly occur should this happen.

Each version of parameter passing has its advantages and disadvantages in unique situations. The programmer should choose the method easiest to work with and least likely to cause problems later on and then debug for that particular application.

EMA/VMA Implementation: An Overview

As with any product, once the user knows something about how it is put together and operates, certain advantages can be realized. Such is the case with EMA/VMA operations.

Figure 2 illustrates the concept of a suit. A suit is defined as 1024 pages of VMA. The suit applies to the entire VMA space, not the working set. As we shall see later, the working set can contain pages from many suits at the same time. Suits are numbered from 0 to 63, or 64 total.

The interaction of the the essential VMA tables and their addressing is illustrated in figure 3. As described in the program fragment presented previously, suppose that data element I(n) is to be accessed. When the EMA/VMA microcode is called, how does the microcode determine where the data element is, and what pages to map into logical memory?

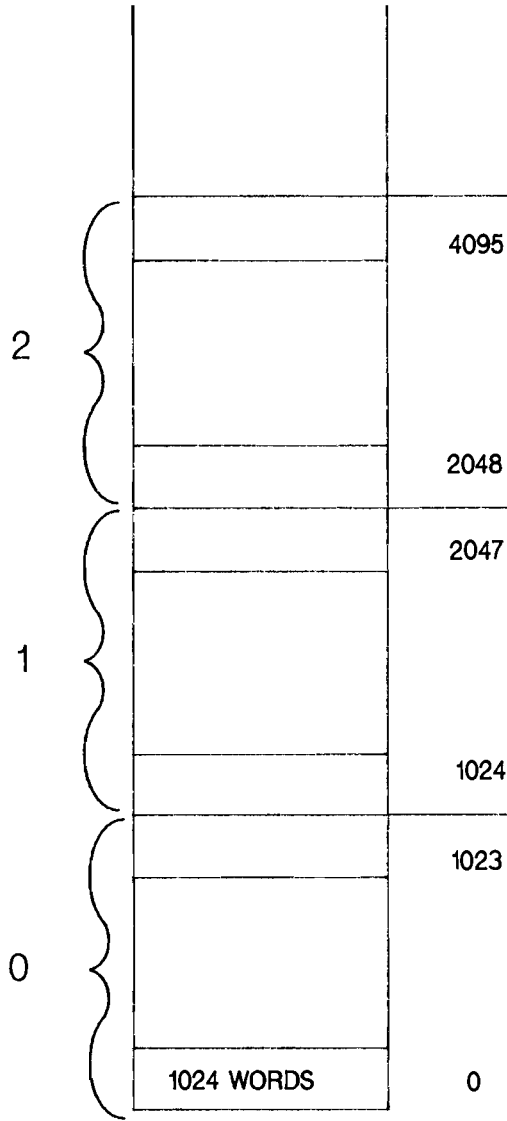
The entire scheme revolves around the PTE. The PTE contains 1024 entries, and has a one to one correspondence with the working set. Each entry in the PTE describes the current contents of exactly one page in the working set.

Further, each entry in the PTE corresponds to exactly one page from each suit. For instance, PTE entry 0 is used by suit 0 page 0, suit 5 page 0, suit 100 page 0, etc. PTE entry 511 is used by suit 0 page 511, suit 5 page 511, suit 100 page 511 etc. Because of the way that this correspondence is defined, only one entry in the PTE needs to be checked to see if the required page and suit is in the working set or not.

The 32 bit VMA address contains the suit number, PTE index, and page offset for the given data element. The microcode begins by examining the PTE at the PTE index. If the suit number of this entry matches that in the 32 bit address, the page is already resident in the working set. The physical page number of the data in the working set is then entered into the logical memory map.

If the suit numbers do not match, then the page required is not currently in the working set (VMA fault). RTE is then called to bring the required page from the backing store file into the working set. It is possible that the working set is full, in which case a randomly selected page will be written to the backing store file and its entry in the PTE is set to nil. The required page is then brought into the working set and overlays the page just written out. The PTE entry for the requested page is then updated and the physical page number is entered into the logical memory map.

SUIT NUMBER

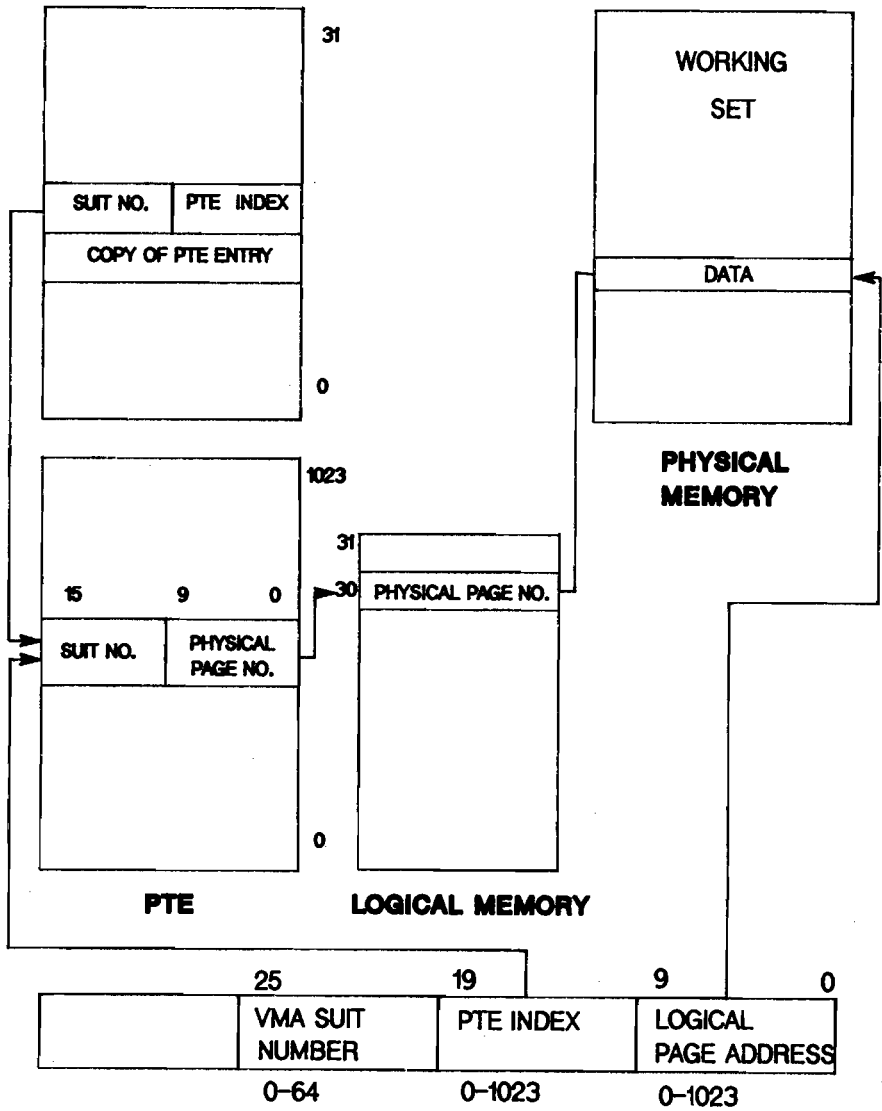


PAGE NUMBER

SUIT DEFINITION

Figure 2

SYNONYM TABLE (STE)



VMA TABLES/ADDRESSING

Figure 3

When the PTE entry for the required page is already occupied, a synonym collision is said to have occurred. A synonym collision will occur whenever the same relative page numbers within different suits are accessed and the pages are still resident in the working set.

When a synonym collision occurs, RTE is called upon to take action. The Synonym Table (STE) is used to contain those pages still resident in the working set, but not described in the PTE. The philosophy here is to save pages in the working set as long as possible, in case they are referenced again (locality of data). RTE moves the current PTE entry to the STE, and then brings in the requested page as before, updating the now empty PTE entry. If the page now stored in the STE is requested again, the STE entry and PTE entry will be essentially swapped. If the STE table becomes full, a randomly chosen entry (page) of the working set will be chosen from the STE only and written to the backing store file.

Once the requested page has been mapped into logical map register 30, the entire process is repeated to map the spillover page to register 31.

Naturally, there are many special cases that occur in the EMA/VMA implementation, too numerous to mention here. There are two additional tables. The Fault Exclusion Table (FET) notes pages which may not be flushed to the backing store file, essentially locking them in the working set. The User Map Table (UPT) contains a copy of the user's logical map and is updated on every VMA fault. This is used for programs which do their own mapping.

Upon close examination of the VMA scheme, it becomes apparent that EMA is simply VMA with only suit 0 active and no backing store file present. In this case the number of pages is limited due to the size of the PTE (1024), less one page for a corner case, or 1023 pages maximum.

General Purpose EMA/VMA Subroutines

Functionally, there are four groups of general purpose subroutines that are user callable. The groups are Information, I/O Management, Sharable EMA, and VMA File Management. These calls are thoroughly documented in the Programmer's Reference Manual, and are presented here for completeness of discussion.

Two Information subroutines, EMAST and VMAST return information regarding the program's EMA/VMA configuration. If users plan to do their own EMA/VMA management, as will be discussed later, these subroutines are essential to determine a program's EMA/VMA configuration.

There are three I/O Management subroutines, VMAIO, EIOSZ, and LOCKVMA. VMAIO provides up to 32 page I/O transfers to/from the EMA/VMA data area. This call is provided since EXEC will not handle EMA/VMA 32 bit addresses directly. VMAIO will call the microcode to map the I/O buffer completely into logical memory before the transfer begins. The program cannot reference the I/O buffer in EMA/VMA while I/O is in progress. EIOSZ returns the maximum I/O transfer size for VMAIO. In RTE-A, this is always 32 pages. In RTE-6, generally this will be from 20 to 27 pages, due to the M/E/F logical mapping structure. LOCKVMA allows the user to lock selected pages of the VMA area into the working set. These pages will not be swapped out to

the backing store file until they are unlocked. LOCKVMA accepts either virtual page numbers or VMA buffer addresses and lengths. The I/O Management subroutines are useful for I/O of large amounts of data, as might be encountered in a data acquisition application.

LKEMA and ULEMA provide similar functionality to LOCKVMA for Sharable EMA partitions (SHEMA). If the SHEMA partition has been locked, then it cannot be reused or overlaid, even if no programs are actively using it. This provides functionality similar to System Common.

The last group of subroutines, VMA File Management, give the user control over the backing store file and some FMP functionality. VMAOPEN specifies the backing store file to be used by VMA. If the file is VMAOPEN'ed in update mode, then the file contains valid data to be updated by the program and pages from the file are brought in to the working set initially. If update mode is not specified, the VMA area is considered to be uninitialized. Update mode is useful for saving data from one program run to the next.

VMAPOST forces all pages present in the working set to be posted to the backing store file. The PTE and working set are unchanged. VMACLOSE performs a VMAPOST and then closes the backing store file. VMAPURGE purges the current backing store file.

VMAREAD and VMAWRITE provide similar functionality as FMPREAD and FMPWRITE, but for EMA/VMA data. FMPOPEN must be called to open the DCB before these routines are called.

EMA/VMA Mapping Management Subroutine

There are 11 user callable subroutines provided with RTE which allow for user control of mapping management. These subroutines are the core internals of the EMA/VMA implementation and allow the user direct access to the internal structure.

The mapping subroutines can be broken down into four functional groups. First, those that perform only mapping or offset mapping (.LBP, .LBPR, .LPX, .LPXR). Second, those that perform a subscript resolution only, or resolution and mapping (.IMAP, .IRES, .JMAP, .JRES). These first two groups map only to the VSEG. The third group (MMAP, .ESEG) allows the user to map EMA/VMA pages to the MSEG. The last group (.EMIO) takes a buffer address and length and maps the buffer into the MSEG, if possible.

.LBP and .LBPR are the simplest mapping subroutines. They are passed a 32 bit virtual address either directly through the A and B registers (.LBP) or via a DEF pointer in the calling sequence (.LBPR). .LBP/.LBPR then map the EMA/VMA page containing the virtual address into the VSEG and map the spillover page. The logical address is then returned in the B register. It is generally best to call these routine from a MACRO subprogram due to their calling sequences.

.LPX and .LPXR perform the same as .LBP and .LBPR except that a 2 word offset is added to the virtual address before it is mapped. This offset is passed to both subroutines as a DEF pointer. On return, the B register contains the logical address.

The second group of subroutines (.IMAP, .IRES, .JMAP, .JRES) are more sophisticated than the first group. They allow the user to specify an array element with subscripts, which is then converted to a virtual address, with an offset. The RES subroutines only resolve the addresses, while the MAP subroutines both resolve the addresses and map the virtual address into logical memory. The I type routines use one word integer subscripts; the J type use two word integer subscripts.

Since these four subroutines calculate array addresses from subscripts, the dimensions of the array must be specified. This is done in a table of the following form:

TABLE DEC n	* Number of dimensions
DEC d(n-1)	* # elements in (n-1) dimension
DEC d(n-2)	* # elements in (n-2) dimension
.	.
DEC d(1)	* # elements in first dimension
DEC w	* # words per element
DEC offset	* Offset word most significant
DEC offset	* Offset word least significant

The array must be stored in column major order. The two offset words at the end of the table allow the array to begin at some offset from the beginning of EMA/VMA. This means that more than one array can be defined in EMA/VMA, but the mapping routines need only concern themselves with one at a time. The offset is defined as from the start of EMA/VMA to element (0,0,...,0). This is a potential problem in FORTRAN, as most arrays there start at element (1,1,...,1). When using these routines with FORTRAN arrays, the offset from (0,0,...,0) to (1,1,...,1) must also be taken into account. This offset can be calculated as:

$$d(n-1)d(n-2)..d(1) + d(n-2)..d(1) + \dots + d(1) + 1$$

For instances, suppose the EMA/VMA area was configured as:

```
COMMON /xxx/ I(5,5),K(6,6,6),J(5,6,7)
```

then the actual offset from the beginning of EMA to J(1,1,1) is:

$$5 \times 5 + 6 \times 6 \times 6 = 241 \text{ words}$$

and the offset from J(0,0,0) to J(1,1,1) is:

$$6 \times 5 + 5 + 1 = 36 \text{ words}$$

so the offset in the table would be:

$$241 \text{ actual} - 36 \text{ address offset} = 205$$

and the table entry would look like:

TABLE DEC 3	* 3rd dimension
DEC 6	* 2nd dimension
DEC 5	* 1st dimension
DEC 1	* words/element
DEC 0	* offset
DEC 205	* offset

When calling these routines, the subscript values must also be passed so that the calculation can be made. For example, the calling sequence of the .IMAP call:

```

EXT .IMAP
JSB .IMAP
DEF TABLE * TABLE describes array bounds
DEF An      * Address of nth subscript value
DEF An-1    * Address of (n-1) subscript value
.
.
.
DEF A1      * Address of 1st subscript value
RTN        return point

```

On return, the B register gives the logical address of the array element referenced for the MAP routines; the A and B registers give the virtual address for the RES routines.

.IMAP, .IRES, .JMAP, and .JRES are used heavily by the FORTRAN compiler to access EMA/VMA data. It is interesting to produce a mixed listing and examine when and how these routines are called.

The third group of routines, MMAP and .ESEG, allow the user to map the entire MSEG. MMAP is callable from FORTRAN and maps only consecutive pages of the working set into the MSEG. .ESEG maps non-contiguous pages of the working set into the MSEG. It is callable only from MACRO. For both calls, the user must specify the EMA/VMA page number(s) to be mapped. The maximum number of pages which can be mapped is MSEG size + 1. These routines are very useful when working with portions of arrays which can be mapped in all at once, thus reducing mapping overhead.

The last group of routines, .EMIO, maps a subscripted array of user specified length into MSEG, if possible. It is provided for compatibility with RTE-6 systems and was used to perform I/O mapping before VMAIO. .EMIO can be useful, however, when dealing with multiple large arrays in a complex program. .EMIO must be passed the array subscripts and length, along with a table describing the array parameters. This table is similar to the one used for the routines in group two.

Increasing EMA/VMA Performance

As was discussed earlier, EMA/VMA data access by FORTRAN is five to ten times slower than normal data access. There are two main reasons for this access speed penalty. First, FORTRAN imposes mapping overhead each time an EMA/VMA data element is referenced. Secondly, the EMA/VMA microcode always maps the spillover page just in case the EMA/VMA data element is not fully contained on the first page.

Effective Usage of EMA/VMA in FORTRAN Programs

The tradeoff being made by FORTRAN is data access speed versus addressing transparency and portability. By mapping each data element on every access the locality of the data elements ceases to be a concern for FORTRAN. The programmer's code is unchanged (except for the \$EMA directive) thus making it more portable to other machines.

There are, of course, restrictions inherent in the EMA/VMA implementation itself. When accessing data with strong locality, the scheme does exceedingly well. When accessing data which is sparsely located, the scheme performs rather poorly. Simply being aware of these restrictions and organizing EMA/VMA arrays around them can improve performance somewhat. VIS routines, which use MSEG rather than VSEG, can also boost performance.

In many instances, users are not satisfied with EMA/VMA performance. The performance is just too slow to accomplish the task at hand. If this is the case, there are a series of suggestions and techniques which programmers can utilize to increase and tune the performance of programs using EMA/VMA. Under normal conditions, these suggestions can speed access to EMA/VMA data. Under some circumstances, these techniques can decrease the overhead of EMA/VMA data accesses to nearly that of standard data accesses.

The tradeoff the user makes for using these techniques is loss of transparent access and portability. If this is not an acceptable tradeoff there is very little that the user can do, other than assuring locality of data.

If the tradeoff is acceptable then the following suggestions should be followed when designing an application for maximum EMA/VMA performance:

- 1) It is best if the application is designed from the start with EMA/VMA access strongly considered. In most instances it is difficult, if not impossible, to redirect the data flow of a currently running application to EMA/VMA without a near total rewrite.
- 2) The application should have strong locality of data in EMA/VMA. The less local the data, the poorer EMA/VMA performance will be. Data should be organized such that commonly accessed areas are located together in memory.
- 3) The programmer must be able to detail when, how, and by whom, EMA/VMA data will be accessed.
- 4) The application must have additional mapping registers available in the logical map to increase the size of the MSEG.

No real weight can be placed on these suggestions, they are all equally important.

Four EMA/VMA Mapping Techniques

A very basic mapping technique uses the MMAP routine presented previously. Recall that this routine allows the user to map consecutive pages of EMA/VMA into logical memory. In this instance, the pages will be mapped into the VSEG. In FORTRAN, how does one access the EMA/VMA data once it is mapped in? This is accomplished

through use of the \$ALIAS directive to set up an absolute common block as shown in the following program fragment:

```
FTN7X,L
$EMA /DUMMY/
$ALIAS /mine/=74000B      ! VSEG address
.
.
COMMON /DUMMY/ Iarray(20000) ! size of EMA
COMMON /mine/ Myarray(10),Mydata(1000)
.
.
My_page=5      ! use EMA page 5
Call MMAP(My_page,1) ! Map in 2 pages
.
Mydata(50)=1   ! access data
Mydata(60)=2   ! no further mapping done
```

(Inline FORTRAN code is shown in these examples only for the sake of readability. A better approach in practice would involve writing a MACRO routine to perform the mapping functions and pass a logical address back to FORTRAN).

Using this technique, the programmer must know, or assign, on which pages the data resides. It is easiest to program if each common block is a multiple of 1024 words or one page. So, for large numbers of small data arrays in EMA/VMA this technique works just fine. Roughly two to four times performance increase can be achieved using this technique mapping only two pages. Mapping a larger number of pages will show an even larger performance increase.

The major difficulty with this technique is the possibility that VSEG will be remapped if another data element in EMA is accessed by the program. In this case, the MMAP call must be made again to map VSEG back to the pages desired. It is recommended that the programmer use only MMAP calls and no other EMA access from FORTRAN, or be very, very careful when using this technique.

A slightly more sophisticated technique can be built upon this flaw. Since the mapping routines which FORTRAN calls always map to the VSEG, why not use the other pages in MSEG for mapping other EMA data. In this way, if the VSEG is remapped, only the VSEG pages will be affected.

An example of this technique might be:

```
FTN7X,L
$EMA (DUMMY,3)      ! 4 page MSEG
$ALIAS /mine/=7000B ! MSEG address
.
COMMON /DUMMY/ Iarray(20000) ! size of EMA
COMMON /mine/ Myarray(10),Mydata(1000)
.
My_page=5           ! use EMA page 5 & 6
Call MMAP(My_page,1) ! Map in 2 pages
lockpage(1)=5
lockpage(2)=6
error = LOCKVMA(lockpage,2) ! lock the pages
.
Mydata(50)=1       ! access data
Mydata(60)=2       ! no further mapping done
```

Now, if the VSEG is remapped, pages 5 and 6 will still be mapped into the first 2 pages of the VSEG. Notice that LOCKVMA was also called to eliminate the possibility of pages 5 and 6 getting flushed from the working set to the backing store file if another mapping routine is called from FORTRAN.

Although this technique is an improvement, the programmer must still know on which pages the data arrays reside. Performance increase is roughly the same as the first technique.

The third technique involves the use of the .IRES/.JRES calls to resolve the addresses of array elements. Recall that these routines are passed the array element subscripts and return a virtual address, and are callable only from MACRO. From this virtual address, both the EMA/VMA page number (bits 25-10), and the offset on that page (bits 9-0) can be easily used.

In the following example of this technique, a MACRO routine RESOLVE is called which in turn calls .IRES to resolve the array subscripts. RESOLVE returns the page number and offset of the array element which were determined from the virtual address.

```

FTN7X,L
$EMA (DUMMY,3)      ! 4 page MSEG
$ALIAS /mine/=7000B ! MSEG address

```

```

COMMON /DUMMY/ Iarray(20000) ! size of EMA
COMMON /mine/ Mydata(2048)

```

c We want to map in Iarray(15879). Resolve it.

```

Call RESOLVE(15879,My_page,My_offset)

```

c Now map in the page where it resides

```

Call MMAP(My_page,1) ! Map in 2 pages

```

```

lockpage(1)=My_page
lockpage(2)=My_page+1
error = LOCKVMA(lockpage,2) ! lock the pages

```

```

Mydata(My_offset+1)=1 ! access data
Mydata(My_offset+2)=2! no further mapping done

```

This technique can be generalized to provide EMA/VMA mapping functionality for an entire program. Performance increase is roughly the same as the first technique.

All of the three techniques presented thus far can call MMAP to map in more than two pages, if the MSEG is large enough.

The final technique uses the .ESEG subroutine in place of MMAP. .ESEG allows non-contiguous pages of EMA/VMA to be mapped into the MSEG. This is useful for moving data from one area of EMA/VMA to another, or to speed access when several EMA/VMA elements might be involved in a calculation of some sort. .ESEG is callable only from MACRO.

An example of this technique might be:

```
FTN7X,L
$EMA (DUMMY,5)      ! 6 page MSEG
$ALIAS /pag1/=64000B ! MSEG address page 1
$ALIAS /pag2/=70000B ! MSEG address page 3
.
COMMON /DUMMY/ Iarray(20000),Jarray(10000)
COMMON /pag1/  Mydata1(1024)
COMMON /pag2/  Mydata2(1024)
```

c We want to map in Iarray(15879). Resolve it.

```
Call RESOLVE(Iarray,15879,Ipage,Ioffset)
```

c We want to map in Jarray(582). Resolve it.

```
Call RESOLVE(Jarray,582,Jpage,Joffset)
```

c Now map in the pages where they reside

```
Call MESEG(Ipage,Ipage+1,Jpage,Jpage+1)
```

```
lockpage(1)=Ipage
lockpage(2)=Ipage+1
lockpage(3)=Jpage
lockpage(4)=Jpage+1
error = LOCKVMA(lockpage,4) ! lock the pages
```

c Now perform the calculation on 500 elements

```
DO i=1,500
```

```
Mydata1(Ioffset+i)=Mydata2(Ioffset+i)*100
```

```
ENDDO
```

In this example, MESEG is a MACRO routine which was written to map in the pages of EMA/VMA passed to it using .ESEG. Notice that in this example, one .ESEG call replaces 1000 .IMAP calls which would have occurred in the DO loop had FORTRAN been allowed to perform the EMA/VMA mapping. Typical performance increases seen when using this technique range from 2 to 25 times.

Variations of any or all of these four techniques can be modified and molded to fit almost any application which has some locality of data. Since these techniques rely on a one mapping/multiple access philosophy to increase the EMA/VMA performance, they would not work well in some applications.

Conclusions

Large data arrays can be effectively managed through the use of EMA/VMA data area. FORTRAN provides a transparent, portable interface to EMA/VMA at the cost of data element access time.

In many applications, it is possible to greatly increase the EMA/VMA performance by performing the EMA/VMA mapping functions within the program itself. An example of four possible techniques are:

- 1) Use the MMAP routine to map EMA/VMA pages into VSEG.
- 2) Use the MMAP routine to map EMA/VMA pages into MSEG.
- 3) Use .IRES/.JRES and MMAP to resolve array subscripts and map their EMA/VMA pages in MSEG.
- 4) Use .IRES/.JRES and .ESEG to map non-consecutive EMA/VMA pages into MSEG for multiple simultaneous array accesses.

Other techniques can also be implemented using the mapping routines directly. Using the mapping routines directly to provide multiple accesses per mapping can result in very large performance increases over FORTRAN EMA/VMA access times.

Appendix A: Glossary of Terms

Logical Memory:	User program's 32 page address space. Points to physical memory.
Physical Memory:	Memory boards install in the computer.
VSEG:	Virtual Memory Mapping Segment. The last two pages of the user's logical memory.
MSEG:	Mapping Segment. The last (MSEG size + 1) pages of the users logical memory.
Working Set:	Virtual pages currently in physical memory. Size is set up at LINK time.
Page Table(PTE):	First page of EMA/VMA area, though EMA/VMA address zero does not begin until the next page. Indicates which pages of EMA/VMA are in the working set.
Page Fault:	Occurs when a request page is not in the working set and needs to be brought in from the backing store file.
Backing Store File:	Virtual data area on disc.

EMA And The C Programming Language

Tim Chase
Corporate Computer Systems, Inc.
33 West Main Street
Holmdel, New Jersey 07733

Introduction

Extended Memory Addressing (EMA) is an interesting solution to a difficult problem. The problem is that the HP/1000 computer using 16 bit addressing is unable to reference more than 32k words of data storage. In the late 1960's 32k undoubtedly seemed like a vast amount of memory, but by today's standards it is tiny.

EMA is often thought to be some form of "virtual" memory management but, in fact, EMA is actually the reverse of virtual memory as defined on most computers. Most processors have a great deal of address space, but don't have the real memory to reference. The 1000, has the opposite problem; it has a small address space with a great deal of real memory. EMA is a way to lengthen the addressing reach of the processor without greatly impacting upward compatibility.

The EMA enhancement to the HP/1000 architecture has enabled a number of large programs to be successfully ported to the RTE. This paper discusses the impact EMA has on the C programmer. In particular it provides information on using EMA in a portable way to enable easy migration between the HP/1000 and the HP/UX based Precision Architecture computers.

Included is an introduction to the operation of EMA, how the concept of EMA is dealt with in the C programming language and some programming examples demonstrating EMA's use in C as well as some of C's unique EMA features.

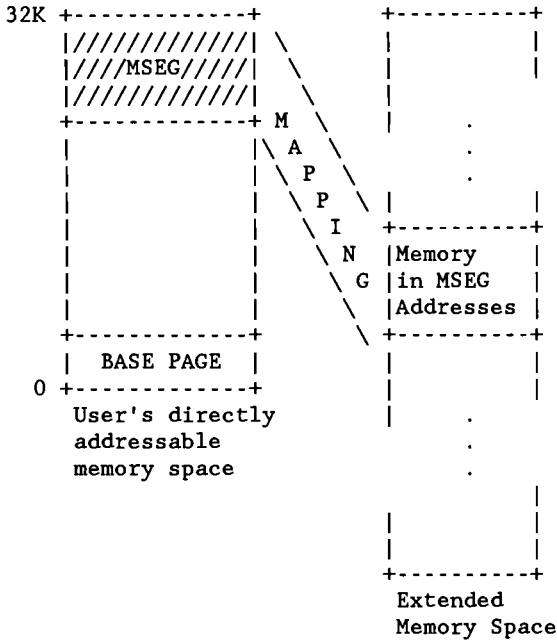
Because the C programming language is "close" to the hardware of any system it is implemented on, EMA presents some interesting problems to the C programmer. Other languages found on the 1000 attempt to hide the functioning of EMA as much as possible with some varying degrees of success. In taking this approach, some inefficiencies are introduced which the programmer has little control over. C, on the other hand, offers the programmer the option of either ignoring EMA's operation or paying attention to it. If the programmer explicitly pays attention to EMA's some interesting improvements in program performance may be obtained.

Although it would be nice to be generic in our presentation of C on the 1000, it is difficult. The reason is that there currently is only one C compiler for the HP/1000 which supports EMA. The compiler we will be using for discussion is CGS/C for the 1000 from Corporate Computer Systems, Inc.

EMA Basics

The HP/1000 is, for the most part, a 16 bit computer. Until the introduction of EMA, all addressing in the HP/1000 was done with 16 bits. As a result, address pointers are 15 bits in length for words and 16 bits in length for bytes. This makes the maximum data memory size be 2 to the 15 power or 32k. The memory of the 1000 is divided into 32 pages each of which is 1024 words in length. One of these pages is distinguished from the others and is called "base page." Because of the special definition of base page, the user is not normally allowed to directly use its space. This leaves 31 pages for program data storage assuming that the application is written using separate code and data spaces.

The designers of the 1000 were faced with a difficult problem. How to add additional addressing capability without redesigning the instruction set. One of the hallmarks of the HP/1000 has been its compatibility with previous versions of the processor. To solve the 1000's limited address space problem HP introduced EMA. It provides pointers which are greater than 16 bits in length and a mapping technique. This technique is perhaps best illustrated with the following figure.



With EMA, the user is given two memory spaces. One is the smaller (31k) directly addressable space and the other the larger EMA space. In actuality, the RTE provides either an EMA space where all memory in the space is real physical memory, or a VMA space (Virtual Memory Addressing) where the second (VMA) space is actually paged out to disc and only a user specified number of pages are actually kept in physical memory. The number of pages retained is called the "working set." Although EMA and VMA are quite different, we will treat them as different forms of the same feature. Unless otherwise noted EMA can be taken to mean VMA in this paper.

In order to address the extended memory without changing the HP/1000's instruction set, the user is given a region in memory called the MSEG area. The MSEG area is a collection of "n" pages (n>1) at the end of the 31k direct address space. MSEG forms a "window" which can be moved over any address in the EMA space. MSEG window movement is called "mapping" and is quite fast (On the HP/1000 900, a remap for EMA takes about 10 micro-seconds). Given the MSEG and some careful programming the user can, effectively, access much more data than the 31k limit imposed by the 16 bit instruction set. There is, however, only one MSEG area. This means that if the program is to move a word from one area in EMA to another area it must be done in several steps (assuming that no "trick" cross map instructions are used):

1. Move MSEG over the source
2. Load the source to a register
3. Move MSEG over the destination
4. Store the data into the destination

The reason for this is that step 2 moves the MSEG window which was set up by step 1. Data must be saved before the MSEG window is moved because EMA data can only be accessed when the window is over top of it.

Fortunately (for the programmer's sanity if nothing else) compilers which allow EMA access maintain the proper placement of the MSEG window. This is at once a good and a bad thing. Most HP/1000 compilers use a simple minded approach in MSEG management dictated by the rule "It couldn't hurt to move the window". Rather than attempt to remember where the MSEG window is located to avoid unnecessary moves, compilers just map, map and remap. In all fairness, though, it is virtually impossible to optimize window position automatically. HP has, instead, opted to make the movement of MSEG so fast that it is quicker to move the window than figure out if it should be moved. Still, moving the window and the extra code to effect the move does increase the program's overhead.

It would be useful, for example, to be able to move the MSEG window over a buffer in EMA and then directly access the various parts of the buffer without suffering a re-map each time an element is referenced. This is difficult, through, because EMA is so effectively hidden from the user. Until the introduction of EMA support by C, only assembly language gave the programmer the required power to implement truly high throughput applications. But assembly language suffers from profound non-portability.

C Meets The EMA Monster

C's approach to EMA is different from other languages supported on the 1000. This is due primarily to the fact that C has a well developed concept of an address. Addresses and pointers are easily manipulated by C and (usually) well understood by C programmers.

As a result, the C compiler provides the user with the choice of either being aware of the location of MSEG or of letting the compiler keep track of MSEG's location. Often, high speed applications need to control the mapping of the MSEG location on EMA. If the programmer does not do his own mapping control, then C will move MSEG on every reference to EMA regardless of whether or not the MSEG window is positioned properly and in so doing C becomes like any other compiler on the 1000.

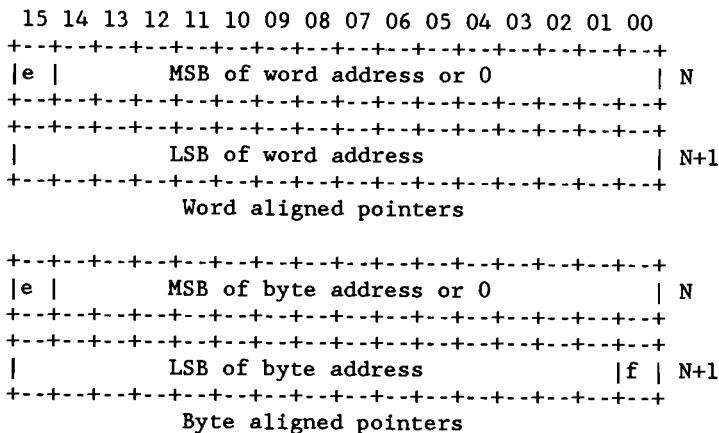
The MSEG window is always "page aligned". This means that the MSEG win-

dow always starts on a page boundary. Some data objects are multi-word and as a result may span a page boundary. This is why the MSEG window must be larger than one page in length. If the first words of an object lie in one page, it is mapped into the first page of the MSEG and the remaining words are mapped into the next sequential pages of MSEG.

In C the "normal" size pointer is 16 bits. This is consistent with the size of the HP/1000's address space. In order to access EMA, however, pointers must be made longer. This presents a problem, though, in that the HP/1000's "likes" to do 16 bit arithmetic and instruction set design is organized around the assumption that the predominate number of references will be to 16 bit objects. If the CCS/C compiler defined all pointers to be 32 bits in length, the code generated would be quite inefficient.

Instead, CCS/C gives the user control over the size of the pointers being used. This is done by adding the definition of long pointers to the standard C language. This enhancement must be used carefully because it is unique to CCS/C and SPECTRUM HP/C compilers. Code written using long pointers will port between the HP/1000, HP/3000 and new SPECTRUM systems (MPE/XL and HP/UX), but will not execute outside the HP world.

The long pointer in CCS/C is 32 bits in length and is capable of pointing to any memory location (EMA or directly addressable) which the user wishes to access. To remain consistent with the 16 bit pointer design, CCS/C provides two different formats for long pointers -- one for word aligned objects and one for byte aligned objects. The formats for each of the pointers is as follows:



In both formats the "e" flag indicates whether the pointer is pointing

to direct (non-EMA) memory or EMA memory. If "e" is set, then the pointer references direct memory and the LSB portion of the address is the address in direct memory which the pointer references. If cleared, the "e" flag indicates that the pointer references EMA. The address in EMA is composed of the MSB of the first word and the LSB of the second word. This gives an effective pointer space of 31 bits in length. (The current EMA implementation does not provide the full 31 bits of address space).

Character pointers are formatted in much the same way. The only difference is that the "f" flag is introduced. The f flag indicates whether the pointer points to the left most byte (f==0) or the right most byte (f==1). Note that on the HP/1000 the left most byte is the byte containing the most significant bits of an int (this is opposite to the byte assignment in a DEC PDP/11 or VAX). In addition to the f field, the character pointer's address bits (bits 0 through 14 in the first word and 1 through 15 in the second word, indicate which word is being addressed. Because the "e" flag is present in a byte pointer also, the effective number of bits available to indicate the word location is 30.

The important difference which C programmers must remember is that only long pointers may reference EMA addresses. Short pointers do not have enough bits to specify EMA references.

Long pointers are declared in C source programs by using the "^" character in the same way the "*" character would be used. For example, to declare a short pointer to an integer one would write

```
int *p;
```

To declare a long pointer to the same integer one would write

```
int ^lp;
```

Note that the ^ character is only used in declarations; it is not used to dereference long pointers to gain access to what they are pointing at. The "*", "[]" and "->" operators are used to dereference a pointer regardless of its size.

To store a value of 0 through the short pointer declared above one would write

```
*p = 0;
```

To store a 0 through the long pointer declared above one would write

```
*lp = 0;
```

The C compiler will emit run time code which will move the MSEG window over the EMA address pointed to by lp and then store a zero there. If lp pointed at a directly addressable location, then there would be no MSEG movement before the zero is stored.

Long pointer may point to any C data type except function. For example a structure:

```
struct T {
    int i;
    char a[10];
};
```

To define a short pointer to an instance of the structure T

```
struct T *sp;
```

To define a long pointer to an instance of the structure T

```
struct T ^lp;
```

Note that declaring a long pointer does not put the pointer, itself, into EMA. It only defines a pointer which is capable of referencing EMA or direct memory.

Because long pointers and short pointers may both reference direct memory with the same degree of accuracy, short pointers may be assigned to long pointers, providing both pointers point to the same type of object. For example, a short pointer to int may be assigned to a long pointer to int without an explicit type conversion (called a "cast" in C). A short pointer to float may not, however, be assigned to a long pointer to int because pointers to float may not be assigned to pointers to int. Further, because short pointers cannot point to EMA directly, long pointers may not be assigned to short pointers without explicit casting. Doing a long pointer to short pointer assignment with a cast does provide a very important feature -- user control of the MSEG mapping operation.

User control of EMA mapping

By using a cast from a long pointer to EMA to a short pointer the C programmer can directly control the positioning of the MSEG window. An example shows this. Suppose that a long pointer to type int is called "lp" and that a short pointer to int is called "sp". If lp is set up to point to an integer in EMA space and the program executes this statement:

```
sp = (int *)lp;
```

The following sequence of events take place

1. The MSEG window is moved over object pointed at by lp.
2. The direct 16 bit address of the int at *lp is calculated.
3. The 16 bit address in MSEG of *lp is stored in sp.
4. The MSEG window is left where it is.

Step number 3 is the important one. Once the MSEG window is moved over the integer pointed to by the lp pointer, it has an address defined in the 32k direct addressing space. This means that the integer now has a 16 bit address which may be stored into a short pointer. This 16 bit address is stored into sp and, unless the MSEG window is moved again, sp may be used to reference the location in EMA pointed to by lp as in the following:

```
sp = (int*)lp;
*sp = 0;

/* Code which does not change MSEG */

x = *sp;
```

As long as MSEG is not moved, *sp may be used to reference the specified EMA address.

This technique becomes more sophisticated when used with structures. Using the structure "T" from the previous example, we can declare a short and a long pointer:

```
struct T *sp, *lp;
```

Next, assuming the long pointer has been set up to the appropriate EMA address, we can assign the short pointer the long pointer's value (using a cast):

```
sp = (struct T *) lp;
```

Once this is done, and if the MSEG window is not moved, the members of the structure may be referenced via the short pointer without any mapping overhead at all. To clear the member array "a" to all spaces we would write:

```
sp = (struct T *) lp;    /* move MSEG */

for(i=0; i<10; ++i)
    sp->a[i] = ' ';
```

This works because the structure lies inside the MSEG window and the short pointer has been set up to contain the real address within the MSEG of the start of the structure. Using casts, short and long pointers, the user has excellent control over EMA mapping and overhead. Such a fine control is not available in any other HP/1000 high level programming language.

Another interesting way in which MSEG window control can be used is with calls to subroutines which do not expect to access objects in EMA. For example, it is difficult in FORTRAN to read and write to buffers in EMA using standard FMP routines. This can be easily done, however, from C. Assume that lp points to an integer buffer located in EMA that is 200 words in length. To read a record into the buffer one need only write:

```
FmpRead(*dcb, error, *(int*)lp, 400);
```

The (int*) cast converts the long pointer lp to a short pointer and moves the MSEG window over the 200 word buffer. The additional "*" operation is required to call the non-C routine "FmpRead". Note that only one argument may be in EMA because there is only one MSEG window.

One must consider the size of the buffers mapped into MSEG. No buffer may be longer than 1024 words in length. This is because the mapping code used on a long pointer to short pointer cast will only map 2 pages of MSEG. If a buffer is longer than 1024 words in length, then the last few words (all words greater than 1024) will not be mapped in MSEG. A 1024 word limit is acceptable for many programs.

Declaring objects in EMA

Up to this point we have discussed how to reference data objects via pointers, but we have not said anything about how to locate objects in EMA. This is done with the EMA pragma.

If you wish to have a static object which is either local or external located in EMA, you need to specify this by using the ema "pragma".

The ema pragma is written:

```
#pragma ema <on-off>
```

The <on-off> field may be either "on" or "off". If the ema pragma with the on argument is encountered by the C compiler, all following static or external declarations are placed into EMA by the linker at load time. This continues until the ema definition is turned off by using another ema pragma with the off argument. There may be any number of ema pragmas in any given source file. Consider the following source example:

```
/* EMA storage */

#pragma ema on

int a[SIZE_A];
extern char b[];
static struct T x[10000];

/* Non-EMA storage */

int d[10];
struct T y[3];
```

In this example a, b and x are all objects which are to be placed into ema. The a object is an array of integers. Note that because a is in "block 0" and is not declared static, it is an entry point. Object b is external to this module, but it is in EMA. This is an EMA external. There must be another module linked at load time which has a declaration of b which defines b as being an entry point in EMA. The array of structures x, is a local EMA object. It is local because it has the static keyword associated with it. This allows the programmer to have EMA entry points, EMA externals and source module local EMA objects.

Turning the EMA off with the #pragma ema off indicates that the compiler should start placing objects in directly accessible memory. Remember, all EMA objects do not have to appear together in the source code. There may be many areas in a source module which make use of the EMA pragma.

Unfortunately, the current version of RTE-A does not allow initialization of EMA. For this reason, no objects placed in EMA may be initialized at compile time. Any attempt to initialize an object in EMA will cause an error indication and the relocatable will be produced as if the initialization were not present. For example the following is illegal:

```
#pragma ema on

int x[] = {1,2,3,4,5};

#pragma ema off
```

This is because x is to be placed in EMA and x is also to be initialized.

A second limitation imposed by LINK is that long pointers to characters may not be initialized with the addresses of objects which are in EMA. They may be initialized with the addresses of directly accessible objects, but not EMA objects.

The following is illegal:

```
#pragma ema on
char a[10000];
#pragma ema off

char ^p = a;
```

because of the attempt to initialize the pointer p to the address of the EMA character array a.

It is the programmer's responsibility to make sure that function arguments match the number and type of arguments expected by the called function. This seemingly poor design aspect of C is also a useful feature because C programmers often (with poor programming style) pass the "wrong" types of things to functions to achieve strange and sometimes

EMA objects and long pointers increase the caution which the C programmer must use when calling functions. Does the function expect a pointer or a long pointer? For example, consider the following function designed to clear arrays of integers to zero:

```

zero(p, n)
int *p;
int n;
{
    while(n--)
        *p++ = 0;
}

```

This function will not accept arguments which are located in EMA. Why? Notice that the pointer argument (points to the array to zero out) is declared with the "*" symbol. This means that the pointer points to directly accessible memory -- the pointer is a short pointer. Short pointers can't directly access EMA so zero() can't be called with an EMA array of integers. The following code would be illegal

```

#ema on
int a[2000];
#ema off

zero(a, 2000);

```

The reason is that the "a" array is located in EMA. Whenever a appears in an expression, it is converted to be a pointer to a, but because a is in EMA, a is actually converted to a long pointer to a. The zero function expects to see a short (16 bit) pointer. Calling it with a 32 bit pointer causes zero to function incorrectly.

With the same declaration for a, we can call zero correctly by the following:

```

zero( (int*)a, 2000);

```

This works because the cast of the long pointer (a) to a short pointer (int*) moves the MSEG window and returns the address within the MSEG of the start of a. Caution must be used to insure that MSEG is large enough to contain the entire a array. If a were 10,000 words long then a 2k MSEG would not work. Casting to a short sometimes works and sometimes does not. Also, if a function expects two addresses (such as a compare word function) the casting technique won't work at all because the first cast sets up the MSEG window and the second cast will disturb it destroying the first cast's work.

A second way to solve the problem is to redefine the declaration of zero's arguments:

```

zero(p, n)
int ^p;
int n;
{
    while(n--)
        *p++ = 0;
}

```

This function will work for any EMA array, but it will not work with a short pointer. This is easy to fix by simply casting the short pointer to be a long pointer. This always works because long pointers don't assume the MSEG window is set up. Long pointers can point anywhere including directly addressable memory.

The technique of casting pointers to be long pointers at function call time is so useful, there is a pragma to do this for you automatically. The pragma is "lpconvert". The lpconvert flag may be turned on or off as necessary. If on, all pointers used as function arguments are converted to be long pointers before the function is called. If turned off, pointer sizes are not changed at function call time. The default for pointer conversion is off. Note that turning lpconvert on will cause ALL pointers to be converted to long pointers -- this includes string constants. Functions like printf() only process short pointers. Using lpconvert and then calling printf would insure a memory protect at best.

Final Thoughts And Conclusions

Although EMA is quite unique to the HP/1000, the C programming language offers flexible control yet remains highly portable to the HP supplied Precision Architecture C compilers. Rumored enhancements to RTE-A to support the ADA programming language should also benefit C because they will address some of the limitations found currently in EMA. Foremost among these is initialization of EMA.

C's low level/high level approach to programming often provides surprising power when it comes to managing various processor quirks. C's ability to effectively deal with EMA is another example of this power.
[ok]



GIMMS Interactive Mapping/Image Processing

Virginia L. Kalb
Thomas E. Goff
Earth Resources Branch, Code 623
NASA / Goddard Space Flight Center
Greenbelt, Maryland

I. Introduction -- evolution of mapping/image processing package

This software package originated on HP computers by chance, not by design. Many years ago, an HP 2100 computer originally purchased for use on an airplane (the only computer available for flight use), was donated to our project when the airplane was transferred to another space flight center. We purchased a disc and other peripherals to allow RTE-II to be used instead of the original RTE-C operating system. Government surplus provided a Comtal wirewrap image system for use in analyzing flight instrument images. This original image system has been updated through the years to include a Ramtek 9351, then a Ramtek 9400 on MX-F machines (still in service), to the present A-900 computer with Ramtek 9465 image systems. The use of HP real-time computers has allowed us to design new I/O cards and/or modify existing I/O cards to interface the image systems to the 2100/MX-F and A series computers. We have continued to use the HP A900 machine as both a host for the image systems and the mapping engine for creating images from satellite data. The A900 computer is especially suited for the mapping process and has been compared with many other computers by the use of a small testing program. The results demonstrate that this machine has a far better cost/performance ratio than any other computer tested.

The choice of a Ramtek image system was based upon the need to use satellite data from the AVHRR instrument which has 10 bits of data available for each pixel instead of the more common 7 bits for MSS. Very few image systems support greater than 8 bit lookup tables and the spatial resolution of 1280 by 1024 pixels necessary for mapping large sections of the world in one computer run.

The purpose of this document is two-fold: to give a flavor for the software package itself, and to describe the unique interplay between the HP computer system and the image processing software.

II. What is GIMMS?

The acronym GIMMS stands for (among other things) Global Inventory Modeling and Monitoring System and is the name of our project that uses remotely sensed data from polar orbiting satellites to monitor earth resources and phenomena such as vegetation growth, deforestation, land and ocean temperatures, carbon dioxide and ozone production, etc. A recommended reference for information on a major product of the GIMMS group is: C. O.

Justice et al, Analysis of the Phenology of Global Vegetation using Meteorological Satellite Data, 1985, *International Journal of Remote Sensing*, 6, 1271. We deal with the entire earth using pixels larger than 1 by 1 km as opposed to other groups that use smaller pixels (30 to 100 meters) over smaller sections of the earth. Because we have fewer pixels to deal with, we compensate by using more pixels per unit of time to create temporal (time based) images of earth phenomena.

Our various users are interested in seeing data in registration with both earth coordinates and other disparate spatial data representations, such as elevation or climate data. We are interested in using data from various satellites as well as from other sources such as terrain elevation or ground based weather stations in our research. We therefore map all data to predetermined map projections rather than force a map to the data. This philosophy allows us to compute, in an interactive manner, the location of each pixel on the Ramtek screen given the latitude and longitude, and also the inverse calculation of finding the latitude and longitude given the Ramtek screen locations. This has developed into a friendly interactive image processing system in which a novice user can effectively obtain quantitative results from satellite images to compare, for example, with derived ground based data from another source, such as field observations. We frequently have off-site users who consequently are not familiar with the system, so it must be as fool-proof and self-explanatory as possible.

III. Hardware configuration

Our current computer systems consist of one HP A900 computer with a 7914 and 7937 disc which is used with two Ramtek 9465 image systems and contains all user files. In addition, we have two HP A900 computer systems with 7914 or 7937 discs that are used for generating maps (world mapping). These latter two computer systems are not used for image processing but for image generation and utilize predetermined answer files for unattended operation 24 hours a day. The older MX-F processor and Ramtek 9400 are used for overflow operations and smaller 640 by 512 images, and magnetic tape operations such as tape duplication. We currently have six 7978 HP tape drives, three 7970E HP tape drives, three 7970B tape drives, and three high speed three density STC tape drives which we constantly move amongst the four system computer systems as needed. Our primary A900 system supports four 12040 mux cards for all user terminals, a LaserJet, a 7550 plotter, and an automatic time clock. We use a modified HP 12006 A series parallel card and in-house written driver to interface the Ramtek image systems to the HP computer.

For the future, we have looked into the possibility of purchasing an HP 9000 series 840 processor for both the image processing and the mapping processes. However, the lack of a modifiable I/O card and I/O driver has eliminated the possibility for the image system and timing tests with our sample program show the 840 to be slower than the A900. Perhaps newer HP machines will tip the balance?

IV. What is an image?

For this system, an image is a 2-dimensional array of pixels with a 16-bit integer value assigned to each pixel (picture element). The lower 10 bits are a data value, and the upper 6 bits are available for graphics overlays. Image data differs from graphics data by being ordinal instead of cardinal. That is, the image data is a numeric value in which an order can be assigned; i.e. 300 is greater than 200 and may represent a quantity such as the number of rabbits per acre of field. Graphic data has no relational order and is usually represented by bit planes in which each bit is treated independently (e.g. one color for each bit). When two or more graphics bits are 'turned on', a decision must be made as to which graphics plane is visually 'on top of' the others and whether the graphics planes are 'on top of' the image data. This is an added software complication.

The images are stored as files using the FMP library, so each image has a unique user defined alphanumeric name <imagename>. We use file type extensions to logically connect the various files relating to a given image. The actual binary data is stored in a type 2 file <imagename>.IMG whose size is determined by the physical image size and is not limited to the Ramtek size of 1280 by 1024. A type 3 ASCII file, <imagename>.NOTE, is used to store GIS (Geographical Information System) 'hooks' (designed access points), including the original world mapping parameters so one can deduce the latitude and longitude for any pixel, as well as the <imagename>.IMG file attributes (e.g. record length, blocking factor, # records). (Note that HP does not keep a running End Of File pointer for type 2 files.) Notes about the image data are stored at the end of this file, allowing its length to be variable. Subsequent image manipulation appends information to this file which can then be printed or edited like any other ASCII file.

When an image is saved from the image system, an auxiliary file is created: <imagename>.AUX. This type 1 file saves the Ramtek image system display parameters so that the user needn't re-derive the best scaling and pseudo-color table each time the image is displayed. The size of this file is determined by the configuration of the Ramtek image system and cannot be changed.

No matter how large a disc a system has, users will run out of disc space. Our philosophy is to force users to manage this problem themselves, so the disc for image data is defined as global directory /PICTURE/ on a separate LU, and each user has a subdirectory <user name> on this directory. Every program that writes data files uses a subroutine NAMECHECK which automatically constructs a pathname on global directory PICTURE, and appends the user's subdirectory. The user can override the default subdirectory by explicitly appending a subdirectory to the filename. This way anyone, not just the system manager, can easily see who is responsible for keeping too many images online. Data integrity is another reason for having each user store data on his/her own subdirectory. A user would have to intentionally specify another's subdirectory to overwrite someone

else's image. Since each user has ownership of their subdirectory, even this can be avoided by protecting the files at the user level. The TF utility is used to archive image data.

V. Mapping Software to Create Images

The mapping software takes data from a variety of formats and origins and processes the data into image file format. The software consists of a shell program, NEWWORLD, which selects which relocatable modules are to be used for the current run. This can be done interactively or from a previously established answer file. A loader command file is created and LINK is scheduled. Next, program NEWWORLD schedules the mapping program just linked (MAPIT.RUN) which performs the mapping of the data.

NEWWORLD is menu driven to select the appropriate subroutines; in interactive mode, the user positions the cursor on the appropriate line and hits the terminal 'ENTER' key. The terminal then transmits the line from the terminal screen to the computer input buffer. The various data sources for imaging data, map projections, and vector overlays include:

1. data sources:

- AVHRR GAC
- AVHRR LAC
- AVHRR - Vegetation Index - GAC
- AVHRR - Vegetation Index - LAC
- Empty data set
- NOAA composite, Polar stereographic
- Holdridge life zones
- NAVY Terrain elevation data set
- Seasat data
- SMMR linear lat-long
- Carbon Balance Data from E. Box
- TOMS bi-directional reflectivity
- Oakridge data
- Skin-shelter temperature tape
- Re-map from INVERSEMAP tape
- LAC scan angle, lsbw = 1/10 degree

2. projections:

- Lambert conformal conic w/2 parallels
- Linear latitude & longitude
- Miller cylindrical # 2
- Cylindrical equal area
- Hammer-Aitoff polyconic equal area
- Equidistant Conical w/2 parallels
- Eckert IV pseudocylindrical equal area

3. mapping sources:

- Disc file
- CIA World Data Base - 1

An ASCII log file, MAP.LOG, is opened by NEWWORLD, in shared mode, and is used to record the questions and answers from NEWWORLD and MAPIT. The program is designed so that all questions and answers are completed before any magnetic tape operations or computation begins. Then the .LOG file is posted so that the user can verify the runtime parameters. Multiple runs can be chained together in one answer file, so listing the .LOG file during execution allows the progress of the job to be monitored. The log file has also proved useful to the computer operators by recording the AVHRR header records, and logging missing EOFs, header records, and bad record lengths from the data tapes.

To accommodate multiple mapping runs, NEWWORLD gets the session number from the system, and forms a 3 character suffix from it; session number preceded by asterisks. This suffix is appended to the run module name, the VMA backing store name, and the .LOG file name.

For unattended operation, it is possible to let the mapping execute in background, using the command XQ NEWWORLD,<answer file>. LU 1 is assigned to a scratch file if 'XQ' is specified in the runstring, thus eliminating screen output. Since LINK is scheduled, its output must also be suppressed; this is done with the system utility ATCRT.

VI. Image Processing Software

The default menu lists the basic functions of the image software, and also the more specialized menu names. The choices are displayed in touch-sensitive rectangles with a descriptive name and a 2 letter mnemonic which can be typed, if desired. The 'touch' code is handled through a collection of subroutines which will be described in more detail later.

Nearly every function is implemented as a separate program, so the size of the software package is unlimited. The main program, IMAGESYSTEM, uses FmpRunProgram to schedule the selected function. Variables are passed to and from the functions via a class buffer, with the class number passed in the runstring. (Actually, the negative of the class number is passed, so that the function can tell if it was scheduled from IMAGESYSTEM or from CI, since a class number of 1 would be ambiguous in the first RMPAR parameter.) Information from the class buffer is used to update a header which occupies the top four lines of the terminal screen. This header is memory-locked on the screen throughout the session. It lists the current image name, one line of notes about the image, current color table and scaling bounds, and overlay status (on/off and color, if on).

The Ramtek is a single user device, so IMAGESYSTEM locks the Ramtek LU at the beginning of the session, and keeps it locked until the program is terminated. The LU and key for the Ramtek are passed in the class buffer. For systems with more than one Ramtek, this makes it easy to make sure the various programs talk to the correct device. The class buffer also contains image system parameters which other programs must read and update during image processing, such as scaling.

Most of the scheduled programs also have touch-sensitive menus presenting the program options. The default choices are displayed in inverse video. This results in faster response time for the majority of users, without sacrificing the ease and interactive implementation of choosing options.

The main menu for the imaging software package has the following format with each menu position being presented in half bright inverse video and the user information and defaults in inverse video:

```
-----
User name  Image name      Color scale      Low, up bounds  Grid flags  Background
GIMMS      PROD/EA84121      BW               0.000  1.000  **RGBCYM**  BLACK
```

Notes: AVHRR GAC data, vegetation index, from 121 1984

```

help      dump image      color &      grids      read      read      save image
(HE)      from disc      scaling      on/off      pixel      area      to disc
          (DU)          (CO)        (GR)        (RE)        (RA)        (SA)

zoom      color          mask &      draw/grid   title     histogram   list
(ZO)      bar           merge      ops         (TI)     & plots    pictures
          (BA)          (MM)        (DR)        (HI)     (DL)
```

Additional menus:

```

          Enhance
Image     Image     Plotting  Geometric  Image     Location   Classify
Creation  Menu      Menu      Operations Analysis  /Mapping
(IC)      (IE)     (PL)      (GE)       (IA)     (IM)       (CL)
```

Next command?

Touch screen menu function definitions:

- HELP -- toggle function; when ON, the system will print a help file on the selected program instead of running the program.
- DUMP IMAGE -- dumps an image from the disc to the RAMTEK display.
- COLOR/SCALING -- provides contrast stretching & psuedo color.
- GRIDS ON/OFF -- turns the grid overlays and background to a user specified color or invisible.
- READ PIXEL -- read back a cursor-designated pixel location & value.
- READ AREA -- read back a cursor designated rectangle or area under a grid overlay.
- SAVE IMAGE -- save the image from RAMTEK to computer disc.
- ZOOM -- hardware blowup and pan; see Enhancement menu for software blowups.
- COLOR BAR -- put labelled color bar on the RAMTEK.
- MASK/MERGE -- logical operations on two images.

DRAW/GRID OPS -- grid manipulations, including draw and fill.
TITLE -- print titles and comments on the RAMTEK.
HISTPLOT -- histograms, line plots, and circular plots.
DL -- computer disc directory list of images.

The contents of the secondary menus are listed in Appendix A.

Some special features of this imaging software package are:

MANGLE provides a facility to write a Fortran subroutine on-line that allows the user to perform very flexible processing of images. *MANGLE* reads in the user's code, compiles the subroutine, and links it with a pre-existing skeleton program. The subroutine is passed a pixel vector and returns a value for that pixel. The user may then save the new subroutine for use again. The user subroutine is written to */PICTURE/<user_name>*.

Create your own *custom color* table. The video lookup table format consists of 2048 words, a pair for each gray level, 0 to 1023. 8 bits are available for each color. The user can set up a video lookup table by specifying the total number of bins, then successively entering a bin number followed by red, green, and blue values. The values may be in counts (0-255) or normalized (0.0 to 1.0). The lookup table is dumped immediately so you may choose the colors interactively.

Draw your own *enhancement* curve, e.g. non-linear or step functions. The standard enhancement curve is a plateau at 0 counts from 0 to the user specified lower bound, a straight line from (lower bound,0) to (upper bound,1023), followed by a plateau at 1023. Non-linear or piecewise linear curves can be drawn interactively on the Ramtek using the track ball.

Variable image size allows the user more flexibility in analyzing data from other sources, such as LANDSAT data. The program *BYTEI* reads in any byte-oriented image from magnetic tape and dumps it to the Ramtek. Then the cursor can be used to delineate a window on the Ramtek to be saved to a disc file (see below). Other needs for a variable image size arise from the desire to map data to a given pixel size to avoid gaps in the data; e.g. a 720x360 data area for 1x1 degree pixels. Image manipulation that occurs within the Ramtek memory is constrained to 1280 by 1024 in size, but, processing within the HP 1000 can be of any size (maximum size is a parameter in the software). This is a powerful argument for software image systems rather than hardware implementations.

Inverse map a previously derived image and remap the pixel data to a new size. Since the mapping information is kept in the *.NOTE* file, it is possible to locate every pixel by latitude and longitude, or to inverse map every pixel to find its latitude and longitude. This has many advantages from locating groundstations and reading back the data at that location, to inverse mapping an entire image and remapping it to another projection.

Processing is *window-oriented* to reduce processing time for most local

operations. The cursor can be used to delineate a subset of the screen to speed up processing when a local operation is possible and desired. More generally, a general region or regions can be specified by using the draw/grid options program to draw the region of interest on the Ramtek screen, then fill it with a graphics overlay, then save this image for input into whatever image processing program is desired. This is implemented by reading the Ramtek memory into the computer one line at a time along with the appropriate line (or lines) of data from the disc and processing the disc data depending upon bit settings in the Ramtek line. Care must be exercised when the Ramtek image is a subset of the disc image.

On-line help in documentation files and examples via touch screen. There is a help switch in each menu which when enabled, will print a help file on whatever menu selection is then made, instead of executing the function. Help is only available at the menu and program level at this point.

Creation of *data files* from image data and image processing programs which can then be used by other analysis packages. Many programs have an option to put derived data into a data file for further processing, either by other graphics packages such as GRAFIT or the statistics package STAT80, or by CORRELATE, an in-house scattergram/linear correlation program. All image system programs use our in-house parsing routine to allow differing formats in the ASCII disc file to be used. STAT80 allows a user specified format that can be set up to parse our data files correctly. If all else fails, EDIT/1000's regular expression facilities allow powerful column exchanging to be preformed.

3-D *scattergram* plotting on the Ramtek. CORRELATE can be invoked from the image system package to plot on the Ramtek, and then can be used to create 3-d scattergrams with gray level representing the 3rd dimension.

Whenever Ramtek *screen coordinates* are required, user entry can be made with the cursor or by typing in the screen coordinates. The Ramtek cursor is fine for ballpark coordinate selection, but for fine-tuning an analysis it is necessary to know the exact coordinates. A subroutine READCURSOR is used by all programs for inputting coordinates, and will parse them if entered; otherwise it executes a read cursor function on the Ramtek.

2-D *interpolation* to create a continuous image from discrete, ungridded data points. LOCALGRID creates gridded data via a least squares plane algorithm through discrete ungridded data points. This was developed to handle climate data available only at specific sites. Contouring can then be done on the interpolated image.

Interactive image *classification* algorithms. The classification programs have flexibility with regard to the selection of training sites by use of the draw/grid operations software to select the region of interest. The usual algorithms have been implemented: thresholding, 2-d histogram, maximum likelihood, and Karhunen-Louve; but have all been written for an interactive capability which keeps the user in the loop. Outlined, resultant,

and feature images may be saved at any point in a session for modification and analysis at a later time. For example, the eigen vectors from the Karhunen-Louve are saved and can be used by the EIGEN program to produce any eigen component images.

VII. Software Development Tools

There is a lot of redundancy in the image processing modules. To save programming time and disc space, several libraries and utilities have been developed:

* Library of subroutines to use/create/process/compare .NOTE file(s). All subroutines communicate the .NOTE information through a common block which is defined in the include file NOTESCOMMON.FTN.

ReadNotes - read .NOTE file into common block variables.
Create_Notes- create .NOTE file and fill in information from the common block.
Update_Notes- update existing .NOTE file with information from the common block.
CompareNotes- compare parameters which must match for registration.

InputImage - request an input image name, and read the .NOTE file.
OutputImage - request an output image name (purge it if the user authorizes).
SetWindow - set Ramtek format window according to image size and scan direction mode.

* Graphics library to talk to HP terminals, bed plotters, or the Ramtek screen. This in-house written library is modeled after the old CALCOMP plotter library used on main frame computers and the HP 7210 bed plotter library:

XSTRT - request a plotting device with an LU number.
XLEFT - for plotter: position pen at lower left corner.
XRITE - for plotter: position pen at upper right corner.
XPLOT - plot a point at given x and y coordinates.
XNUMB - print a numerical label at given x and y coordinates.
XSymb - print text at given x and y coordinates.
XSymbPLOT- like XPLOT, but the user can pass the plotting symbol too.
XSTOP - for plotter: put pen back in carousol.
XPEN - select pen, for plotter, or graphics overlay, for Ramtek.
XERS - for Ramtek: erase graphics plane.
XPIXPLOT - for Ramtek: plot pixel of selectable size and data value.

* 'Touch' library of subroutines to set up menu touch fields. This Fortran subroutine is used to define touch screen fields on an HP touch screen terminal or PC, or bypass the touch commands for a non-touch screen device.

Functions: CMD=INIT to initialize touch screen mode
CMD=SOFTKEYOFF to disable touch on function keys

CMD=SOFTKEYON to enable touch on function keys
CMD=DELETE to remove all user touch fields
CMD=FIELD, followed by specifics to define a field namely,
starting row (0-47) and column (0-79), # of rows, # of
columns, response string (up to 80), and label (use ';' to
put label on more than one line)
CMD=DELETEFIELD, followed by a point in the field to be deleted
(row,column)
CMD=TOUCHON to enable touch sensitivity (also unlocks keyboard)
CMD=TOUCHOFF to disable touch sensitivity (does NOT affect field
definition and also locks the keyboard)

* NAMECHECK, to create a pathname with the user's subdirectory given a filename, global directory, extend, and optionally file type, size, and record length.

* LPARSE, to parse multiple parameters from an input string. This subroutine handles characters, integers, decimal numbers, hex and octal numbers. It passes back several arguments that can be used to determine the type of item parsed and its location within the character string.

VIII. Problems ... and solutions

Whammy DCB(7) for type 2 files. For efficiency of file transfer to/from disc and to/from Ramtek, the images are stored in blocked mode, 4 lines per record. However, many programs need to process the image one line at a time so the solution is to overwrite the record length (after opening the file) with the unblocked line length, and everything is fine.

CDS and non-CDS libraries for modules that use FMP calls. You cannot mix CDS and non-CDS FMP calls, so duplicate libraries must be maintained for processing the .NOTE files, the only difference being a '\$CDS ON' compiler statement.

System EOF is useless for type 2 files unless the size is a block multiple. This makes the record length and number of records information in the .NOTE file indispensable; you cannot use the error indications from the FMP calls to find the end of file!

Must lock keyboard when putting up touch menu. If you touch the menu before it is complete (and before the program is expecting input), the touch fields are unstable, so now we lock the keyboard (escape c) while putting up the menus and unlock it (escape b) when the menu is completed.

Touch library must put out different code for 2393 and HP150 terminals. When defining the touch fields, the 150 requires an explicit carriage return character appended to the end of the line, but the 2393 does not, and generates a spurious CM> prompt if you do provide the carriage return character. Solution: the 'init' command determines the terminal type and saves it; the type is tested each time a field is defined, and the

appropriate code is generated. We will make any alterations needed to run the software on the Vectra, but we haven't had an opportunity to try it yet.

Only a finite number of touch fields can be defined at a time, including multiple definitions of the same field. This situation arises when the user makes multiple selections, and the menu is rewritten after each selection. Solution: every time a field is defined, the code must automatically delete it first.

VMA backing store size must be defined in the .LOD file, unless you want 8192 pages, the system default. Unfortunately, you can't specify the size with the VMA open command in the program. Also, you need to explicitly name a backing store file if there isn't room on the default directory. This file must be purged when the program completes, and if it aborts, the file remains to haunt you. Our convention is to put all backing store files on the user's subdirectory of /PICTURE/, and to name the file <program name>.VMAB; this allows multiple users to run the same program without file conflict.

IX. Future development

Interface optical discs. Image processing and derivation from satellite data is an ideal use of write once, read many times long term storage techniques as embodied in the WORM optical disc drive. Concerns such as seek and transfer times are relatively unimportant due to the slowness of the human user in the loop. Error rates are also of very little importance as the amount of missing data from the satellite greatly exceeds any current disc error rates. Space is a real driving force for implementing optical discs: the volume of data that we process on a daily basis is huge, and our magnetic tape storage volume is an order of magnitude larger than the combined computer and user space. Another important factor is ease of access; high density magnetic tape (CCT or newer VHS type) are sequential and hard to live with when accessing an image or data set. We will have an optical disc drive on our computer as soon as possible, but we may have to 'do it ourselves' unless commercially available systems for HP 1000's appear soon.

Convert all software to CDS. Several times, we have hit the memory limit when adding, modifying, or changing programs. CDS allows additions to be performed more easily with fewer programming tricks and time wasted and is a major requirement in our application.

Continuing software conversion and enhancement. Each new addition to this imaging software seems to spawn several additional ideas with the resulting programs proliferating like Lemmings. (And similar to lemmings, several run off the cliff into the bit bucket.) Each revision represents a major change in which older software has to be rewritten to incorporate new ideas and capabilities in either the operating system or the hardware (both Ramtek and HP). Our current emphasis is on computer peripherals as

described above with an expected migration to newer imaging hardware to follow in approximately one year. Software efforts will be expanded to improve the accuracy of the satellite images with respect to spatial positioning, instrument calibration, atmospheric corrections, and other areas, with the usual enhancement (and bug) fixes taking priority.

X. Summary

Although this software/hardware package is an on-going project, it has reached a plateau of capability in which the system is sufficiently mature for outside dissemination. A distinct advantage for our project is the fact that the entire software package including I/O card modifications and drivers has been written in-house. This means that explanations of what the system does or enhancement descriptions (otherwise known as bugs) can be resolved quickly in minutes instead of days. It also means that many items don't get documented in written form. However, outside users are going through the documentation, forcing its improvement.

Appendix A

Additional menus from auxillary programs:

IMAGE CREATION

- BYTEI -- retrieve byte images from magnetic tape.
- EXPOS -- test patterns for hardware debugging.
- CONVERTIMAGE-- convert images from Rev 4 format to Rev 5.
- LOCALGRID -- 2-D interpolation from discrete ungridded data.

IMAGE ENHANCEMENT

- LIFT -- 'lift' data value(s) to a grid overlay.
- PAINTAREA -- 'paint' a data value into all pixels under a grid.
- CUSTOMCOLOR -- make up your own color table.
- ILLUMINATE -- create light from point source effect.
- NO_HOLES -- fill or create holes
- RGB -- create true color from 3 red,green,blue components

PLOTTING MENU

- CORRELATE -- scattergram and linear regression of .DATV files
- HISTPLOT -- histograms, line plots, and circular plots

GEOMETRIC OPS

- SZOOM -- software pixel replicate
- SHIFT -- linear shift
- BLOWUP -- magnify or reduce image about a center point
- MANGLE -- create your own geometric operation

IMAGE ANALYSIS

- STAT -- 2-D correlation of 2 images
- DIFFER -- difference 2 images
- MANGLE -- create your own image analysis operation

LOCATION/MAPPING

NEWORLD -- menu-driven mapping software
INVERSEMAP -- create lat-long tape from previously mapped image
LOCATE -- interactive mapping/inverse mapping of discrete point
LOCAD -- mapping of lat-longs from disc file, then read back
neighborhood from image and print
LOCATEAREA -- inverse map region from image under grid plane

CLASSIFICATION

CLASS0 -- interactively define training sites
CLASS1 -- 1 parameter (thresholding)
CLASS2 -- 2 parameter histogram
CLASS3 -- maximum likelihood
CLASS4 -- Karhunen-Louve transformation
EIGENP -- create eigen value images





Remote Monitoring and Control of
Timing Equipment for Navigation Systems

Mihran Miranian, Supervisory Mathematician
Francine Vannicola, Mathematician

U.S. Naval Observatory
Time Service Department
Monitoring and Control Branch
Washington, D.C. 20392-5100

INTRODUCTION

The U.S. Naval Observatory (USNO) Time Service Department has remote Data Acquisition Systems (DAS) situated at worldwide locations for the purpose of monitoring and controlling timing equipment which regulate radio navigation systems (Figure 1). This timing equipment must be synchronized to the USNO Master Clock (USNO-MC) which is located in Washington, D.C. Until recently there was only one method of precise time synchronization which provided less than ten nanosecond time comparison. This was the extremely cumbersome, time consuming and expensive portable atomic clock operation carried out by USNO personnel. But now this same accuracy can be achieved by making time comparisons via the Global Positioning System (GPS).

An HP1000/A900 computer at the USNO in Washington, D.C. is in daily telephone contact with each remote DAS. Data from each site is sent to USNO at high speed (up to 9600 baud) for processing and time synchronization. Control of the remote atomic clocks is performed via the same link. This operation is fully automatic. The A900 calls the remote DAS computer, logs on, and commands it to carry-out the necessary data transfer and equipment control.

DAS SYSTEM (Figure 2)

The DAS is a microcomputer controlled system capable of monitoring locally available precise time navigation transmissions such as Loran-C, OMEGA and GPS. Also, the DAS can perform intercomparisons of local atomic time standards. The heart of the DAS is an HP9915A or HP9915B microcomputer. The computer controls two interfaces - an RS232 interface primarily used for data communications and an HP-IB interface for equipment control.

The RS232 interface connects the computer to a standard dial up telephone line via a 1200 or 2400 baud modem or as in the latest DAS a TELCOR Series 24 Accelerator modem. The accelerator provides additional capabilities such as a near 9600 baud data transfer rate, error correction and auto-baud detection. The HP-IB interface is connected to one or two HP59307 VHF switches, an HP5328 or HP5335 universal counter, an Austron 2100 Loran-C timing receiver, an HP9133 Winchester disc, a Timing System Technology (TST) precision digital multi-timescale clock, a TST microphase stepper, and a GPS timing receiver. The VHF switches are connected to the atomic time standards at the site. Once per hour the VHF switches are directed to connect up to fourteen different timing signals (one pulse per second) into the universal counter. The universal counter makes the time interval measurements and transfers them to the computer for storage on disc. Also each hour the Loran receiver is sequentially locked on to selected stations, and time of arrival measurements are made and

stored on disc. In addition to the hourly readings, the computer is continuously monitoring the GPS satellite timing receiver and storing the data for preprocessing and subsequent transfer to USNO.

USNO A900/DAS COMMUNICATION

The HP1000/A900 is scheduled to call each DAS once a day. The calls are usually made between 5:00 a.m. and 7:00 a.m. EST. During this time period telephone lines are relatively free and the call failure rate is low. The computer will try to call a site five times. If there is no response after the fifth call, then the call is abandoned until the next day and an alert is printed. Once a call has been established, the A900 logs onto the remote computer and sends the proper commands requesting data transmission. All received data is stored in a single file and sorted later into separate files for storage and processing. Some processing is done during the communication with the remote DAS and the results are sent back and printed on the DAS printer. There is constant checking being carried out by the A900 to ensure that the telephone link has not been lost and that data sent back has been received properly.

Each DAS computer interface is configured at 1200 or 2400 baud, even parity, and one stop bit. There is no handshaking enabled, but there is a ten millisecond inter-record gap to allow time for transmitted data to be logged into a file. This simple configuration permits a variety of computers to communicate with the DAS computer.

All I/O to the A900 is via the 12040 multiplexer. During the writing of this paper the 12040B was upgraded to a 12040D. Therefore, we will make reference to both versions of the MUX card. Unlike the 12005A ASYNC card, the MUX card does not provide status and control lines. Consequently, there is no way to directly control the telephone connection. This handicap is overcome with the use of a TELCOR Series 24 Accelerator modem. The TELCOR uses the Hayes command set and allows independent modem control. A normal call to a remote DAS is carried-out in three steps. First the modem is configured to operational mode, given the telephone number, and commanded to place the call. Once the connection has been established, a series of commands is sent to the remote computer to log-on, request transmission of data, and log-off. Finally, a command is sent to gain the attention of the modem so that the hang-up command will be understood and carried-out.

The remote DAS software is designed to operate in an interactive mode. Accordingly, the communication software in the A900 is geared to respond to certain requests made by the DAS computer. As there is no handshaking enabled on the DAS computer, all handshaking must be turned off on the MUX port. Flow control settings differ between the B and D versions of the MUX. On the B-MUX the port was configured with the ENQ/ACK handshaking disabled in function code 30B. Also, function code 34B was set to zero to ensure that XON/XOFF was disabled and function code 45B was set to zero to turn off the DC1 trigger character sent by the DD.00 device driver. On the D-MUX bit 7 in function code 30B is reserved and must be set to zero. Flow control is determined by setting function code 34B. For our purposes function code 34B is set to zero to select TTY protocol for no flow control handshaking and normal CRLF processing.

All read/writes are performed with EXEC calls. This allows the most flexibility in handling incoming data and total control of the MUX port. Reads are performed using Class I/O, thus permitting constant monitoring of incoming data and the ability to take corrective action if the telephone link is lost or if there is no response from the DAS computer. With the 12040B MUX bit 13 was set in function code 33B to enable type-ahead so that data received without a pending read is saved on the MUX interface until the next read request. Also, each read was made with the by-pass bit (bit 15) set in the control word to by-pass the device driver so that the MUX would not echo back the CRLF after each record. Even with type-ahead enabled there were times when data was lost at the 9600 baud transfer rate and a buffer overflow warning was displayed. This usually occurred when a long record was followed by a very short record (one character). Either all or part of the record following the short record might be lost.

This was probably caused by the short record filling the second buffer on the port allowing no space for the following record. Fortunately, this problem has been corrected on the D-MUX with the introduction of FIFO. FIFO is enabled by setting bit 15 in function code 33B. Another welcome feature, on the D-MUX, is the Transparent ASCII Read. With this feature you have the option to redefine the termination character or use the default CR. Transparent ASCII Read is enabled by setting bit 8 to 1 and setting bits 0-7 to the termination character in function code 17B. Additionally, bit 10 must be set in the read control word to trigger detection of the termination character.

At the end of each call, we gain control of the modem by sending three pluses (+++) with no CRLF. This is done on the B-MUX by setting bit 7 in the control word to disable the CRLF sequence normally transmitted. On the D-MUX this is done by setting bit 10 in the control word.

Our experience has shown that the 12040 MUX card is a strange animal which on occasion has a mind of its own. Especially the earlier version. But we have found that with proper care and feeding it can be trained to perform as it was intended. Actually, we have been quite satisfied with the MUX. Except for the buffer overflow problem and some early learning pains on our part, the MUX has served as a reliable interface between the A900 and our DAS sites.

CONTROL OF TIMING EQUIPMENT

Each DAS is equipped with one or more Cesium Beam Frequency Standards (Atomic Clocks). One of these clocks, usually the best performer, is designated the Master Clock (MC) for the station. The 5 Megahertz (MHz) frequency of the MC is fed into a phase microstepper. The phase microstepper can then be set to phase shift its own 5 MHz output which has been locked to the MC 5 MHz. The phase of the output 5 MHz is then advanced or delayed in one picosecond increments for any selected period of time. The phase shifting effectively advances or retards the rate of the MC. The 5 MHz from the microstepper is the input to a precision digital clock which converts the 5 MHz to one pulse per second (1PPS). The 1PPS from the digital clock is the official Station Clock (SC) (Figure 3).

Every atomic clock has a rate of change as compared to the USNO-MC. If this rate were to remain constant, then a single setting of the phase microstepper would compensate for this rate and the SC could be made to match the USNO-MC. But atomic clocks do not have a constant rate. They vary depending on environmental conditions and the natural aging of the cesium tube. It is this wandering of the rate which requires the resetting of the phase microstepper to keep the SC to within a predetermined range of the USNO-MC.

But, how do we compare a clock, which is thousands of miles away, to the USNO-MC? The solution is to compare both clocks to a third atomic clock at very nearly the same moment of time. GPS provides this capability. Each GPS satellite has onboard an atomic clock which can be monitored with a GPS timing receiver. In practice, the timing data from the GPS satellites are compared to the USNO-MC and the remote SC when the satellites are in common view of both sites. When the satellite clocks are mathematically removed the result is a value of the remote SC as compared to the USNO-MC. (Figure 4) Measurements made over a number of days yield the (USNO-MC) - SC rate. Once the direction and rate and the time difference of the SC as compared to the USNO-MC has been determined, then an evaluation can be made as to whether the microstepper requires resetting. When resetting the microstepper, one of two actions are performed. One is to reverse the direction and change the rate of the SC. This is done when the SC is moving away from the USNO-MC. The other is a braking action which is performed if the SC is moving towards the USNO-MC at too high a rate. The goal of both actions is to adjust the rate of the SC so that it is moving towards the USNO-MC at a rate of two (2) nanoseconds or less per day. The following conditions determine if microstepper resetting is required:

1. If the SC is within ten nanoseconds of the USNO-MC and the rate is less than or equal to two nanoseconds per day, then do not reset.
2. If the (USNO-MC) - SC time difference is increasing, then reset the microstepper so that the (USNO-MC) - SC time difference will begin decreasing at a rate of two nanoseconds per day.
3. If the (USNO-MC) - SC time difference is decreasing at a rate greater than two nanoseconds per day, then reset the microstepper to decrease the rate to two nanoseconds per day.

By following these simple rules, it is possible to control and maintain the Station Clocks at our remote DAS sites to within twenty nanoseconds of the USNO-MC (Figure 5).

CONCLUSION

The USNO HP1000/A900 computer is the control center for a network of remote DAS sites. The time synchronization link between these sites and the USNO is GPS and data transmission and equipment control is via telephone. This combination of satellite timing and computer control provides a precise method for stabilization and control of the SC at each DAS. The precise time navigation transmissions monitored at DAS sites are referenced to the DAS-SC. By maintaining the DAS-SCs to within 20 nanoseconds of the USNO-MC we are effectively comparing the navigation transmissions to the USNO-MC, thus providing a global time standard for synchronizing all radio navigation systems.

REFERENCES:

1. Wheeler, Paul J. (1983), "Automation of Precise Time Reference Stations", Proceedings 15th Precise Time and Time Interval Applications and Planning Meeting, p. 41-52.
2. Winkler, Gernot M. R. (1985), "Changes at USNO in Global Timekeeping", Proceedings of the IEEE, VOL. 74, No. 1, January 1986.
3. Winkler, Gernot M. R. (1985), "The U.S. Naval Observatory Master Clock", USNO Time Service Handout, July 26, 1985.

WORLDWIDE DAS STATIONS

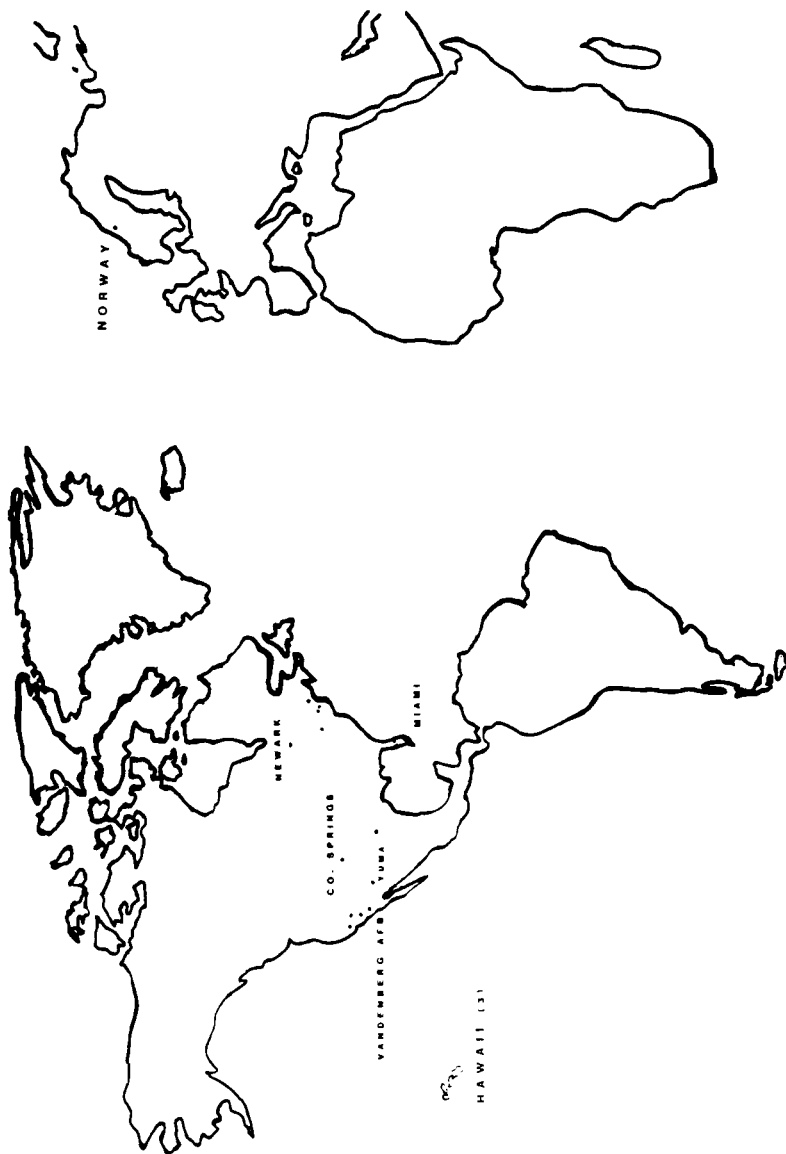


FIGURE 1

USNO AUTOMATED REMOTE DATA ACQUISITION SYSTEM (DAS)

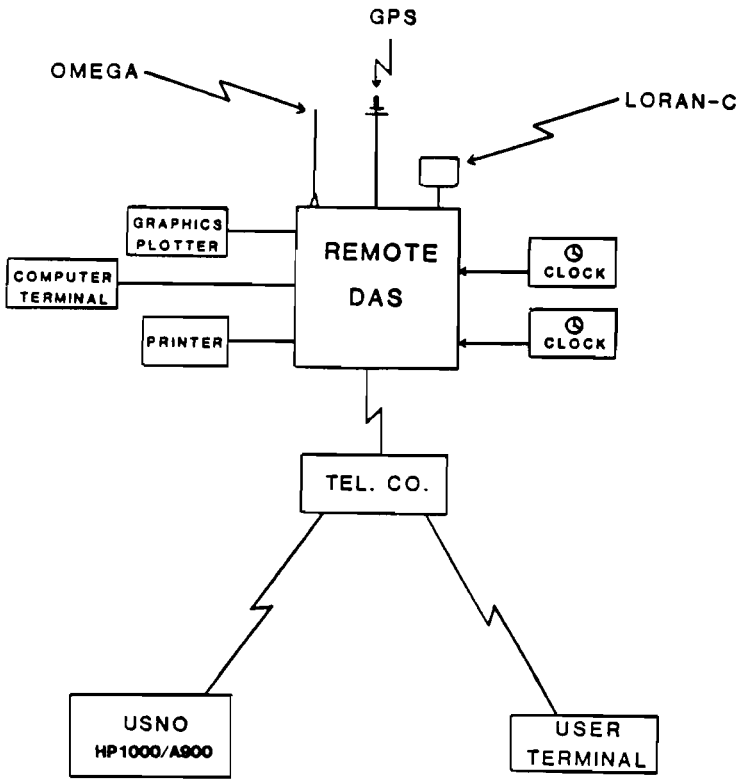


FIGURE 2

DAS STATION CLOCK CONFIGURATION

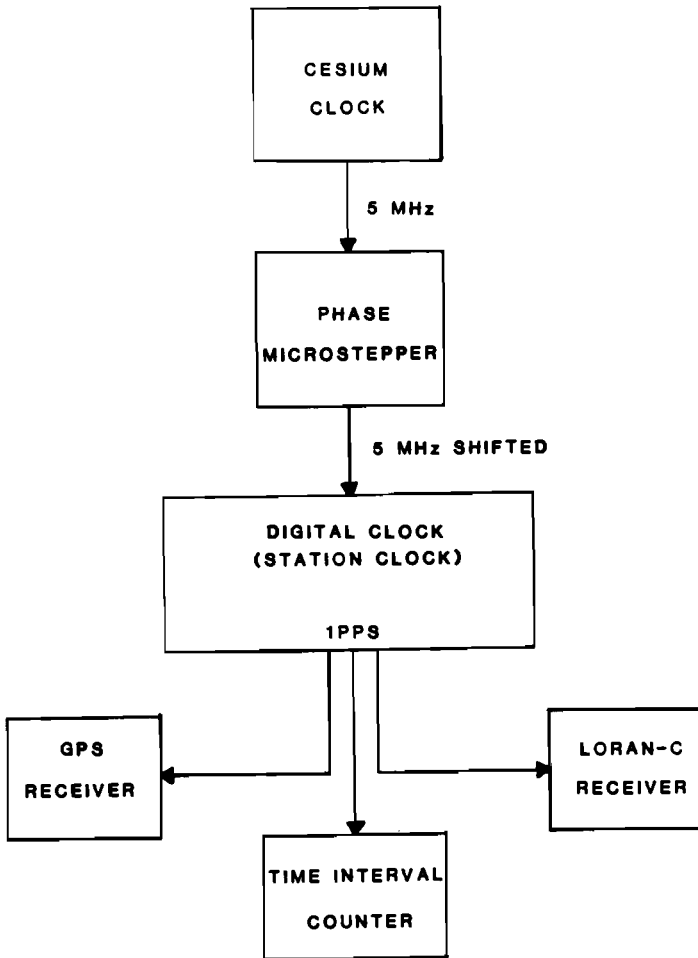
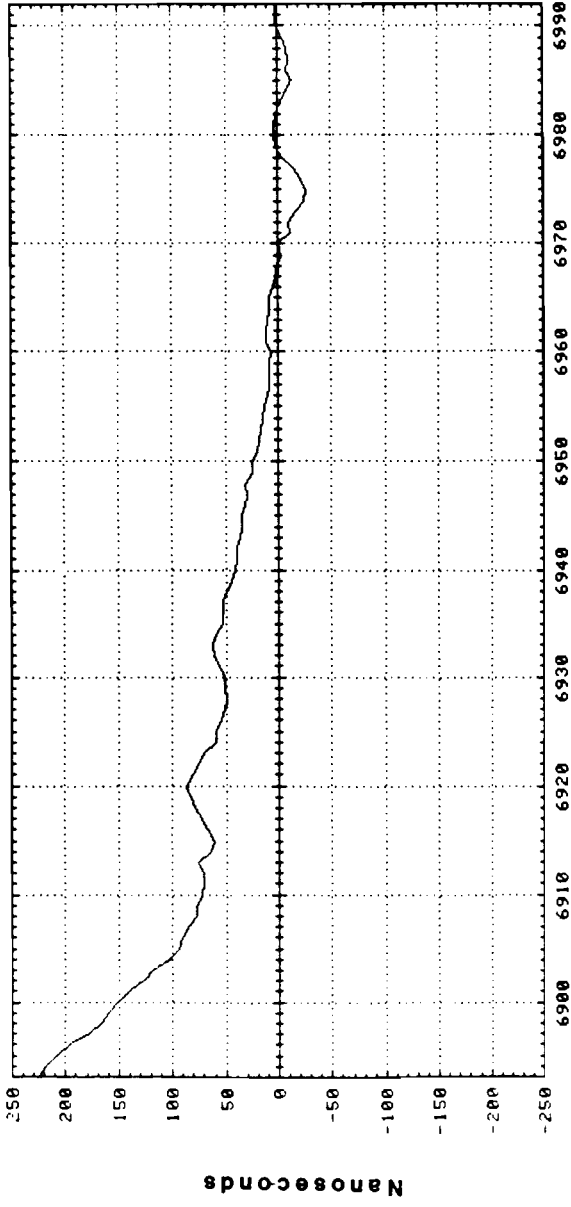


FIGURE 3

USNO (MC) - DAS (SC)		VIA		GPS	
DATE	MJD	USNO (MC) - GPS NS	DAS (SC) - GPS NS	USNO (MC) - DAS (SC) NS	
1987					
JUN	26	180	191	-11	
	27	175	191	-16	
	28	172	196	-24	
	29	165	191	-26	
	30	161	183	-22	
JUL	1	165	179	-14	
	2	162	167	-5	
	3	159	157	2	
	4	157	154	3	
	5	159	154	5	
	6	159	159	0	
	7	164	165	-1	
	8	161	167	-6	
	9	156	169	-13	
	10	152	161	-9	
	11	143	153	-10	
	12	143	151	-8	
	13	142	146	-4	
	14	142	141	1	
	15	142	140	2	
	16	139	137	2	

FIGURE 4

USNO(MC) - DAS(SC)



8 APR 87

16 JUL 87

DATE (MJD)

FIGURE 5



Data Acquisition at High Clock Rates

Hugh C. Hanks, Jr.

DOD

Ft. Meade, MD 20755

One of the most important phases in signal analysis system operation is Signal Acquisition. It must occur prior to signal processing and display. Signal acquisition systems currently available in the market place have constraints which make them unacceptable. The most severe constraint was the small number of samples which could be taken contiguously. Typically, the limitation on the number of samples lies between 1024 and 65,536; although the sampling rate might be as high as one GigaHertz per second. Normally the samples in these equipments are stored in solid state memory which places an economic limit on the number of samples which can be stored. In other cases the acquisition system design limits the number of samples to some discrete number. Existing systems are incompatible with the requirement for a signal acquisition system in which the data acquisition rate varies from one hundred KiloHertz to one GigaHertz and the limit on the number of contiguous samples is fifty Gigasamples. In this application the signal could be either analog or digital. The data acquisition system requirements are shown in Table 1. Note that at the maximum data acquisition rate of two Gigabits per second, the acquisition period is limited to twenty five seconds, while at the minimum sampling rate, the acquisition period can be approximately eight hours. A feasible system which was economical was required to meet this requirement.

The characteristics of typical commercial systems which have been in operation for several years are shown in Table 2. The most significant factor is that our present equipment can acquire only 778 Kilobytes per second in real time or 20 Megabytes per second in non-real time. The new system must acquire 250 Megabytes per second in real time. This is an increase of more than one order of magnitude relative to existing capabilities. A study was made to find a solution which was both practical and economical. The most difficult task was storing continuous bit streams at a one Gigabit clock frequency. Storage media available included magnetic tape, magnetic disks, optical laser disks, and solid state memory. These media are compared quantitatively in Table 3. These are typical values for 1986 and improvements in characteristics can be expected during 1987 and later years. Currently the magnetic disk appears as the best available storage media if it is practical to spread the data stream over a large number of disks. This means that continuous data streams of one Gigabit per second must be made into N continuous streams of 0.5 to 2 Megabytes per second (4.0 to 16 Megabits per second). The desirability of optical-laser disks is offset by the lower writing speed of the units currently available, which results in a higher system cost. There is some indication that the writing

speed may increase sufficiently by late 1987 or 1988 to make them competitive.

Current technology has provided hardware components for operational systems which demultiplex one Gigabit per second data streams into eight data streams at 125 Megabits per second. Typical of these chips is the LOGO41 made by GigaBit Logic which provides the demultiplexed streams and the necessary clocks for cascading chips and other operations required by a system. At least nine of this type chip would be required for each one Gigabit per second stream. This would provide 64 parallel streams, each at a clock rate of less than 2 Megabytes per second. Additional chips could be used to obtain more streams at a lesser number of Megabytes per second. At least one disk drive would be required for each data stream. The number of drives per data stream is a function of the controller and the FIFO either internal or external to the controller. The magnetic disk drives of many different manufacturers were studied. Other than capacity, the parameters of primary interest were burst transfer rate, time track to track, bytes track cylinder, bus/ interface, size, and MTBF. The cost of the drives becomes a major factor in the final design, and it is considered along with other pertinent factors.

All of the drives studied operated at 3600 rpm which makes the time of rotation approximately 16.67 milliseconds. The burst transfer rate must be greater than the input data stream rate if the disk is to keep up with the incoming data. At the highest bit rate the data stream will be two Megabytes per second. A typical burst transfer rate which exceeds 2 Megabytes per second is 2.4 Megabytes per second or 2.4 Kilobytes per millisecond. Normally the disk (using sequential formatting) can only record on every other revolution of the disk, since it is necessary to move the head to the next track during the second revolution. Neglecting overhead the disk could record 49.01 Kilobytes of data during each 16.67 millisecond period. During the period covering the recording and the movement of the head, 66.68 Kilobits of data will arrive for processing. This problem has two solutions 1) Use a recorder with a burst transfer rate of at least 4 Megabytes per second, or 2) Use two recorders with the same controller which results in one recorder recording while the head on the other is being moved. Another potential solution is to format the disks such that sector zero on successive tracks is positioned for recording to begin as soon as the head movement (track to track) is completed. This would reduce the non-record time to approximately 8 milliseconds. This would require the FIFO to handle only 49.34 Kbytes of data between starts of the record periods. This would reduce the FIFO and buffering required. It would probably be necessary to use double buffering. The use of a formatted disk requires downgrading the burst record rate to allow for overhead. This is normally about 20%. If a burst record rate of 2.4 MByte per second was used, then a single disk per bit stream could handle as a composite a total bit stream of 664 Mbits per second. This system configuration is shown in Fig. 1.

Commercial components exist which makes it possible to build this system. However, the components must be carefully selected to insure compatibility. Careful design is required in the reduction of the data streams from one Gigabit per second to sixteen Megabits per second (2 Megabytes per second). This design requires the application of microwave stripline techniques in the reduction of a Gigabit stream into sixty-four parallel streams. This technique is currently being used in commercial equipment and as such is a low risk design. The next most critical part of the design is the control of 64 to 512 disk drives in parallel.

Normally a controller located in the CPU, controls the disk directly or through an interface which might also contain another controller. In this case there must be a dedicated controller for the disk or disks for each bit stream. This set of controllers must then be controlled by a controller in the CPU. The system is kept simple by requiring that all the disks start at the same track and sector. It must be recognized that although the disks start and move on the same command, physically, they do not actually start recording at the same precise time. In the worse case the lagging disk might be approximately one track behind the leading recorder or over 16.5 milliseconds. In terms of data this would be about 33 kilobytes of data. The FIFO must be adequate to handle this spread in the data from disk to disk. With this configuration the data streams would not be going via the CPU but directly to the individual controllers or through the FIFO for that controller. This system is illustrated in Fig. 2. The problem is actually in the hardware rather than in software. The CPU located controller (master) acts in a normal control configuration; however, the data is not passing through the CPU. The dedicated disk controller responds to the master controller and controls both the disk and the FIFO. The available disk controllers do not contain enough FIFO buffers for this application; hence, the need for the external FIFO. HP equipment would not require significant changes to be made in existing application software and computer interface. The computer interface includes the hardware interface board, the firmware and the system software driver.

This system requires a multiple of 64 disk drives. In some configurations the number of disk drives might be as great as 1024. Taking the simple case of 64 drives, this means that the data is spread over these disks such that bits 1, 65, 129, . . . , etc. is on drive 1, with 2, 66, 130, . . . , etc. is on drive 2, and similarly on the other drives. Thus, before the data can be used, it must be assembled into a more normal manner. At the high clock rates this will require more time than was required to take the data.

This talk has been directed to the system configuration which will allow the acquisition of data at two Gigabit per second rates. Of great interest is the actual equipment used to achieve this goal. It is possible to implement this design with HP equipment or with the equipment of other companies. Similarly it can be implement with the combined equipment of several manufacturers. We had planned to

provide the final configuration in this paper; however, due to a delay in releasing the RFP the contract had not been let at the time this paper was prepared. The complete system which includes this high speed data acquisition subsystem is now scheduled to become operational in late 1988.

Type Signal	Clock/ Sampling Rate		Parallel Digital Streams	
	Min Clock/ Sample Rate	Max Clock/ Sample Rate	Min	Max
Analog	100 KHz		16	16
Analog		250 MHz	8	8
Digital	100 KHz		2	16
Digital		1 GHz	2	2

Time Period ... 25 Seconds at Max Clock Frequency
8 Hours at Min Clock Frequency

Table 1 – DATA ACQUISITION - 2 Gigabit/ Second (MAX)

Type Signal	Clock/ Sample Rate	Real Time			Media
		Digital Streams	Bytes Per Second	Number Samples Max	
Analog/Digital	1000 Hz	16	2 KBytes	(2**31-1)*12288 200 M	Tape Disk
Analog/Digital	389 KHz	16	778 KBytes	(2**31-1)*12288	Tape
Analog/Digital	288 KHz	16	576 KBytes	200 M	Disk
Digital	18 MHz	16	3.6 MBytes	1 M	Memory
Non-Real Time					
Analog	20 MHz	8	20 MBytes	3.6 G	
Digital	160 MHz	8	20 MBytes	3.6 G	
Digital	80 MHz	2	20 MBytes	3.6 G	

Table 2 DATA ACQUISITION - Typical Capabilities

Media	Time	Burst Record	Sustained Record	Cost Per Drive	Significant Problem	Available
Computer Tape	Real	0.75 MBytes/Sec	0.75 MBytes/Sec	\$40K	Space	Now
Instrument Tape	Non-Real	2 GBytes/Sec	2 GBytes/Sec	\$1M	Read/Write Heads	1990
DISKS						
Removable-Magnetic	Real	2.3 MByte	2 MBytes/Sec		Space	1986
Winchester-Magnetic	Real	0.5 - 2.4 MB/Sec	0.5 - 2.0 MB/Sec	\$0.5K - \$10K		1986
Optical Laser	Real	0.5 10 MB/Sec	0.4 - 0.8 MB/Sec	\$2K - \$10K		1987
Solid State	Real	10 MBytes/Sec	10MBytes/Sec	\$100K		1987

OEM Prices would be lower

Table 3 -- Comparison of Media

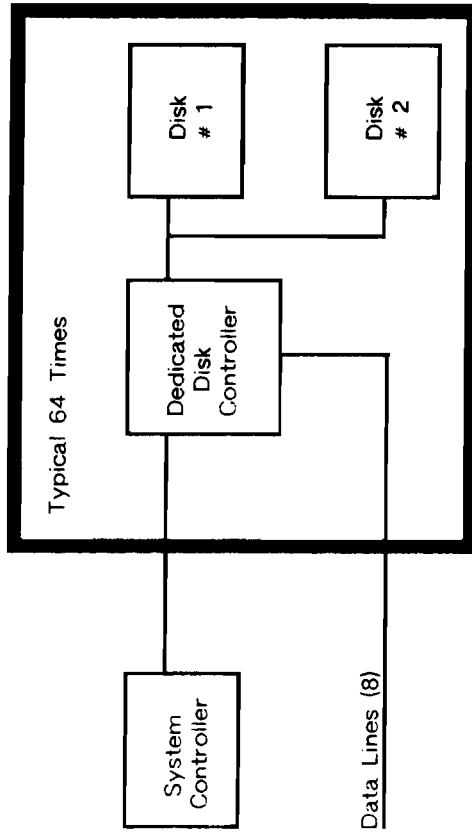


Fig 1 -- Method of Recording Data

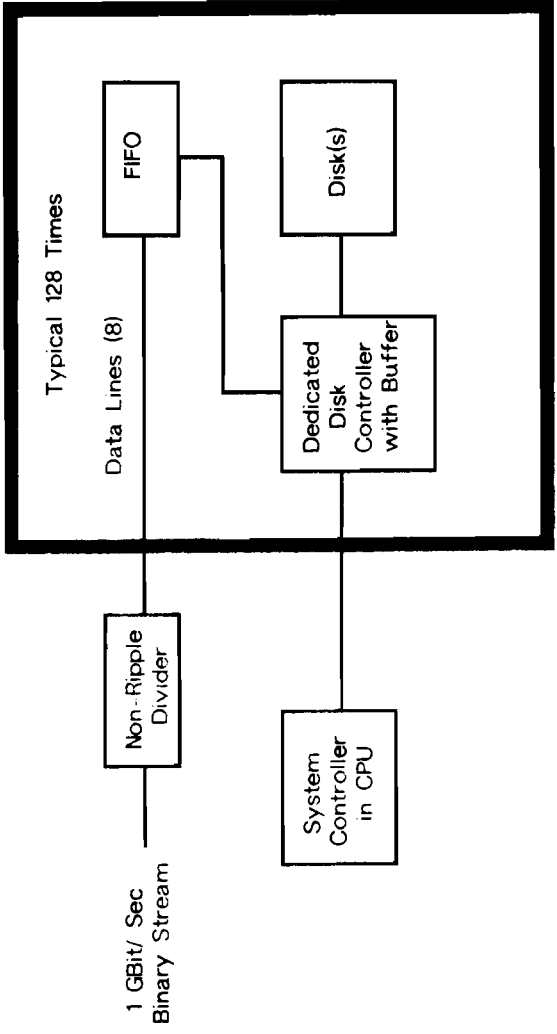


Fig 2 -- Simplified System Diagram of DAS Subsystem



An Introduction to Optical Disk Technology

Husni Sayed
IEM, Inc.
P.O. Box 8915
Fort Collins, CO 80525

INTRODUCTION

Optical disk drives, which use lasers to store and retrieve data from an optical disk, are one of the latest developments in mass storage technology. Optical disks have a number of advantages over more traditional storage methods. Their most obvious power comes from their tremendous data capacity, which allows large amounts of data to be stored compactly. Since the media can only be written to once, they are most ideal for storing data that does not change over time. Many military applications are ideally suited for optical disk drives.

Despite their advantages, there are a few problems that need to be dealt with when using optical disk drives. Optical disks are more susceptible to media defects than other storage media, so special care must be taken to see that all errors are detected and corrected. Also, since a "standard" computer-disk drive operating system interface expects that an area can be written to more than once for directory maintenance, special software drivers are necessary to utilize optical disks.

Once these considerations are dealt with, the advantages gained by using optical disks should far outweigh any disadvantages. And as the technology develops, more features, such as erasable disks, will add to the allure of optical technology.

HISTORY

Laser technology found its first popular commercial use in the music industry, storing digitized sound on CDs, or Compact Disks. CDs became popular very quickly, because the disks are so compact, and the sound quality far superior to tapes and records. Now, laser technology is being used in the computer industry as well, in the form of optical disk drives.

Optical disk technology was "fathered" by Phillips Corporation. CD-ROM and WORM (Write Once, Read Many) drives are now being developed by such companies as Sony, Toshiba and Ricoh, along with a host of smaller companies. Currently, CD-ROMs and WORM drives use optical disks that can be written to only once, as the writing process causes permanent alteration of the media. Erasable disks for optical disk drives are sure to appear in the future.

ADVANTAGES OF OPTICAL DISK DRIVES

Optical Disks are rapidly becoming the media of choice for data storage and archival, due to their numerous advantages over other data storage and archival methods. These advantages include the relatively long archival life of optical disks, the compactness of the optical disk cartridges, high capacity, and ease of data access. Optical drives are also less vulnerable to problems caused by media surface contamination, are not prone to head crashes, and data cannot be accidentally overwritten, or erased by magnetic fields.

Magnetic Tape

Magnetic tape, for instance, has traditionally been chosen for the archival task. But magnetic tape has a relatively short life span: three years or so. Stretching or breaking of the tape and print-through (which occurs when the magnetic field from one layer of the tape migrates to an adjoining layer) contribute to the degradation of the media, hence corrupting the stored data. Optical disks, on the other hand, have a projected archival life of 10-20 years. In addition, data access times are faster for optical disk drives which are random-access devices, than for magnetic tape which must be searched sequentially when data needs to be retrieved.

Microfilm

Microfilm, which has a 100 year life expectancy, is another media commonly used for archival purposes. Microfilm's greatest disadvantage is its inconvenience: retrieving stored data is a tedious task. This characteristic makes microfilm great for storing information that is rarely (if ever) accessed, but too inaccessible for storing data that needs to be used. Optical disks offer an alternative that allows fast and easy access to any stored piece of information.

Magnetic Hard Disk

Optical disk drives also have a number of advantages over magnetic hard disk storage. Firstly, a major concern when using magnetic disk drives is the possibility of a head crash, resulting in lost data. Magnetic heads must be very close to the disk to work properly: they are typically only 8-10 micro inches above the disk surface. When shock or vibration occurs, the possibility of a head crash is very real. Optical heads, however, may be as far as 2 millimeters away from the optical disk surface, making a head crash extremely unlikely. Secondly, magnetic disk drives are much more sensitive to contaminants (dust particles, smoke, etc.) in the head assembly and on the disk media. Particles on the disk or in the head can interfere with the reading and writing of data, causing the information to become corrupt. Optical heads do not focus on the disk surface, but on the media substrate. Because of this, contaminants have much less of an effect on the reading and writing of data to an optical disk. Thirdly, optical disks are much less vulnerable to stray magnetic fields than are magnetic media (tape or disk). Once data is stored on an optical disk, it cannot be erased by accidental exposure to magnetic fields.

Optical disks have additional advantages over all other storage techniques. A 5.25-inch optical disk can easily hold 200MB of data (or more) per side. The removeable cartridges are compact, capable of storing vast amounts of data, and easy to access when you need information. Optical disks are less expensive than other data storage methods when you take into account the overall data capacity, and with a WORM (Write Once, Read Many) drive, data cannot be accidentally overwritten. No other data storage technique has this unique combination of features.

DISADVANTAGES OF OPTICAL DISK DRIVES

Though optical disk drives have a number of advantages over other storage methods, they are not completely free from drawbacks. Speed is an important consideration in any application. Though optical disk drives are faster than sequentially accessed magnetic tape, their access times are at best comparable to magnetic hard disk. When speed is crucial, the optical disk drive may be outperformed by magnetic hard disk.

Presently, the most outstanding difference (which may or may not be regarded as a disadvantage) between optical disk drives and other storage techniques is that data written to an optical disk is permanent. Erasable optical disks may be available in the future, but for now, once data is written to an optical disk it cannot be overwritten, edited or erased. This is not the case for any other popular storage/archival techniques.

ERROR PREVENTION AND CORRECTION

This "write-once" feature of optical disk drives creates another problem: how to ensure data reliability in write and read operations. Optical disks, by their very nature, are more prone to surface defects than magnetic media. A typical optical disk has a bit error rate (BER) of $1.0E-4$ or $1.0E-5$: meaning that for every 10,000 ($1.0E+4$) or 100,000 ($1.0E+5$) bits written, one error (on the average) occurs. A typical BER for magnetic hard disk is much closer to $1.0E-12$.

On magnetic media, surface defects can be detected before the media is ever used by writing data to the media and reading it back. This procedure identifies the defective sectors and tracks, which are then skipped over when the drive is used to actually store data. This greatly reduces the chance that data will be written on a defective area. This method cannot be used with optical disks, since write operations destroy the media. So, the quality of the optical disk surface cannot be ascertained before it is used. Certainly, careful media production can help to reduce surface defects, but such defects cannot be eliminated.

Errors can also be caused during the writing operation if the laser is poorly focused, or if the head assembly is not properly centered on the track. Optical disks are grooved, much like a phonograph record: writing is performed on the raised portion rather than in the groove itself. If the head assembly is not properly centered during the write operation, the "pit" representing the data will not be centered on the raised portion, and may not be read properly.

To compensate for the fact that optical disks have a relatively high BER, very effective error-correction techniques must be employed. One method for detecting errors entails dividing data that is read by a known polynomial, and storing the result on the disk. When the data is read back, the division is performed again, and the result compared to the result stored on disk. If the results differ, an error has occurred. While this method will reveal that an error has occurred, it cannot pinpoint the exact location of the error, and it does nothing to correct the error.

Another alternative is to use an error check and correction (ECC) code. With this method, errors are checked and corrected as they occur. This method is more useful since it actually corrects errors as they occur, but this method also uses a significant amount of disk space.

The biggest problem with any error correction method is that bad sectors of the disk cannot be foreseen and compensated for. If a large area of the media is defective, a lot of time and space can be wasted detecting and correcting error after error as it occurs. Despite these problems, errors can be found and corrected, without significantly deteriorating the performance of the optical disk drive.

APPLICATIONS

Optical disk drives, because of their ability to store very large quantities of data in a limited space, are especially suitable to data archival and backup purposes. In general, applications involving information that does not change over time can take advantage of optical disk technology. This includes such things as testing results, seismographical data, topological data, medical data, and knowledge bases.

Meeting Military Specifications

Many military applications, such as terrain guidance systems, moving map systems, and data archival and backup, make use of unchanging data. With such a range of uses for optical disk technology, meeting military standards (not an easy task by any stretch of the imagination) is the next obstacle that must be overcome.

Different branches of the military have different specifications for the equipment they use. Standards must be met for every aspect of a drive's function, including radiation, noise, shock, vibration, temperature, humidity, and voltage. A drive that will be used on a submarine may have to fit through a hatch that is 25 inches in diameter, and be able to endure pitch and roll 30 degrees from horizontal with no loss of function. One shock test for the Navy involves dropping a hammer onto the drive during operation.

Optical disk media may well have the most trouble standing up to military guidelines. Two different substrates are considered for use: glass and plastic. Glass is usually considered a better choice as it can tolerate higher temperatures and more vibration. Most optical disks employ a sensitive material that is Tellerium based. Tellerium, however, does not tolerate humidity well, making it susceptible to corrosion. To compensate for this the media must be sealed within an air pocket, which can cause problems if the air pressure drops. Some companies have turned to a Platinum-based media which is not as easily corroded.

COMPATIBILITY AND INTERFACING

Most optical disk drives use an SCSI (Small Computer Systems Interface) for interfacing purposes. IEM's Optical Disk Drive has been designed especially for use with Hewlett-Packard (HP) computers: it uses the CS-80 data transfer protocol, and can be used with any HP computer that has a standard HP-IB (Hewlett Packard Interface Bus) interface. This facilitates interfacing, and the writing of customized software for read and write operations.

Read/Write Operations

Most Operating Systems, including those supported by HP computers, require a storage media that allows multiple writes to the same track or sector, as do standard magnetic

disk drives. Directory entries, which reside in clusters on the disk, must be updated each time a file is added to the disk. If a drive cannot be written to more than once, this directory cannot be updated using conventional methods. Therefore, standard operating system disk drive interfaces cannot be used to access an optical disk drive: special drivers, specifically tailored to optical drive read/write operations, must be used.

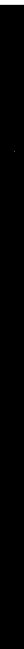
SOFTWARE SUPPORT FOR OPTICAL DRIVES

The software problems for supporting optical disk drives are centered around the fact that a given sector of an optical disk can only be written to once. Conventional operating system support for directories requires the ability to rewrite a given sector within the directory area. To circumvent this restriction, a new directory structure must be devised for use with optical disk drives.

Since optical disk drives will be used for archival purposes, one must be able to have more than one file with the same name. This requirement imposes the need to devise a method for specifying which instance of a given file the user is trying to access. It is also important to allow the user to retrieve files that have been previously archived on a given date.

Two classes of special software drivers are needed to properly interface to an optical disk drive. First, stand alone utilities must be available to allow the user to store and retrieve files from magnetic disks to optical disks. Secondly, the user must have access to (or be able to write) utility programs that allow running programs to access the optical disk for reading and writing of data.

The disk directory problem can be solved by using one sector before each file as the directory entry describing the file. The same sector can also be used as a file header. This header can contain information about the file, including the file name, the date the file was created, the file type, access information, the size of the file, the creator's name, and a link to the next file header/directory entry. Using such a method would allow the user easy access to any file on the optical disk.



STATISTICAL PROCESS CONTROL
A PRACTICAL APPROACH
TO INCREASED MANUFACTURING PRODUCTIVITY

Greg Lawson
ADVANCED MICROSOLUTIONS
2510 Middlefield Road
Redwood City, CA 94063

Statistical Process Control (SPC) is a technique that uses statistics to detect substandard materials and wasteful or counterproductive operations in manufacturing processes. By monitoring the information available from the various steps in a process and applying the appropriate statistical analysis, abnormal trends can be identified and problems resolved before they get out of hand.

Often called Statistical Quality Control (SQC) or Total Quality Control (TQC), this technique has been used to cut production costs by up to 30% and more!

While the Japanese did not invent SPC, they were first to see the potential benefits and adopt the techniques as a national policy. Today SPC techniques are used by more than one-third of Japanese companies while a mere 50 U.S. companies use SPC in their manufacturing. SPC has enabled Japan to become a world leader in quality and productivity, allowing them to successfully exploit worldwide markets.

This paper will address the implementation of SPC in today's manufacturing industries. First I will review some of the benefits that can be realized using SPC. Then I will show how SPC can be applied to your manufacturing process at a reasonable cost and without disrupting on-going production. An example showing how HP9000's can be used for a versatile, expandable SPC system is then described followed by seven steps you can take to implement a successful SPC system.

Use SPC To Reduce Production Costs

A manufacturer who implements and uses SPC techniques, on a consistent basis, can greatly reduce production costs and increase the quality of the product by **REDUCING COMPLEXITY AND SIMPLIFYING THE PROCESS**. Whether the quality of the product is determined by its functionality, durability, or appearance, it must be built into the product, not added as an afterthought.

When you look at a production process, at first it may

appear to be relatively simple. If you take a closer look at the operation, you find a number of tasks that complicate the smooth production flow. Things that generally were not intended to be part of the process. Things like rework, part shortages and "bone yards". All of these tend to add complexity to the production process.

This complexity is rarely understood in manufacturing. Rework areas, kit holding areas, bone yards and the personnel to man these functions are typically added after the need arises.

Complexity weakens any process because it provides opportunities for things to go wrong. It consumes time and resources unnecessarily. Not only does it cost time and money, it is invariably a cause of low quality.

Reduce Complexity

The goal of an SPC system is to identify and eliminate the causes of this complexity, to simplify the process. Once in place, the SPC system will enable you to continuously monitor your process for problems, and to rapidly isolate the sources of those problems. It will provide unbiased facts for making decisions, with a defined probability of making the correct decision.

The key is to make it right the first time and avoid the tremendous expense of both finding and then fixing defects. As stated in a Special Report on Quality, published in the BusinessWeek issue dated June 8, 1987, the "typical factory invests a staggering 20% to 25% of its operating budget in finding and fixing mistakes." Additional, largely unmeasurable costs, are incurred when defective products pass inspection and are passed on to customers and then ultimately returned for repair or refund.

Treat Product Failures As Process Failures

SPC teaches you to treat product failure as an indication of a process failure. In order to improve product quality, you must improve the quality of the process.

Simplification of a production process with the aid of SPC, will yield a reduction of both process and product failures. This can be done in many ways, both by monitoring the process to keep it in control, and by designing products for manufacturability. When manufacture is a key ingredient of product design, keeping the manufacturing process under control assures a product that consistently meets specification.

Focus On The Exception

Monitoring of product and process, when in control, is a simple procedure with SPC. Your SPC monitoring system notifies you when your process is about to go "out of control", that is, when it may start producing products which are not of the quality you desire. Efforts can be focused on the problems, or exceptions, not the normal acceptable production which is hopefully over 95% of the time.

You can then choose to adjust your process to put it back into control or halt production to avoid building faulty product which will need rework. Ideally, your SPC system can adjust your process automatically, by controlling switches, valves, temperatures, etc.

Rather than spending valuable resources to improve the method of rework or repair, SPC focuses on eliminating the causes of these problems.

There Is No "Acceptable Level Of Scrap"

Today, manufacturers are being forced to eliminate the "acceptable level of scrap" from their operating philosophy, in order to remain competitive. They must begin to look for ways to reach what many have called "zero defects", an idea that was previously considered too expensive and unrealistic outside of the aerospace and defense industries.

If you can stop the manufacture of one defective unit in a lot, or half of the lot, you will save significant dollars in the form of rework, scrap, labor and floor space. No matter what your lot size, the principles of SPC can be applied.

Companies that are consistently using SPC today have found they can:

- Reduce defects by 1 to 2 orders of magnitude;
- Shorten manufacturing times by a factor of 10;
- Cut inventories in half;
- Significantly reduce floor space;
- Cut labor in some areas by a factor of 2.

SPC - A Capital Expenditure Or Source Of Funds

The executives of many companies today view SPC as a capital expenditure, something that should be justified like a piece of machinery or additional manpower. Purchase of a large automated system can be a large expenditure, but it is typically not necessary to begin on such a large scale

today. It is actually implementation of the philosophy of SPC that makes the difference. Starting small, with a few SPC stations, or even with manual SPC, can reap great benefits.

If implemented properly, SPC techniques should not represent an expenditure, but in fact should make a significant contribution to increased profits.

When a firm adopts the SPC philosophy and diligently applies the techniques, they typically find that they not only reduce manufacturing costs, their sales often increase as well.

By improving the quality of the products delivered and more accurately forecasting delivery dates, customers will come to view the manufacturer who uses SPC as a desirable vendor, one that can meet commitments and helps to reduce lead times, costs and rework.

Why Isn't SPC More Widely Used Today?

If what I have said is true, you might wonder why more U.S. companies aren't using SPC today. I have asked myself that same question many times. It appears the main reasons for the limited application of SPC today are:

1. a lack of understanding,
2. a shortage of trained personnel, and
3. the fear of failure.

If the benefits are so great why don't we read about fantastic successes in the trade press? What has gone wrong with past SPC programs? Why have the attempts of others not resulted in resounding success?

There are a number of reasons why past programs have met with limited success. Some of these reasons are:

- o The tools were harder to develop, or more expensive than planned.
- o Too much emphasis was placed on statistical training and not enough on the application of the statistics.
- o There was a general lack of understanding of the goals of SPC and a lack of planning. SPC was viewed as a "miracle cure", the end rather than the means.

Implementing SPC - The Necessary Commitments

SPC is a technique that requires a deterministic attitude. Management commitment is not only necessary, it is crucial to its success. Full implementation requires long-term thinking, and an attitude that looks to prevention of problems rather than crisis management.

This can mean a much less visible role for the successful manager. They must look for their reward as a smooth, error free environment. This can be threatening to individuals that have gained reputations as good "fire-fighters".

SPC MUST BE KEPT UP! That is the cold hard fact! This is the only way SPC will lead to the continuous improvement in productivity and lowering of defect rates that is promised. The true savings will appear in your on-going production.

Firms that adopt SPC expecting to have the benefits accountable in a short period of time can become frustrated. You should not expect to show a date on a time line that says "ALL SAVINGS REALIZED - END OF PROBLEMS".

The SPC techniques, as I have mentioned, can be applied using manual or automated systems. For the purposes of this discussion, I will focus on the automated system. In your situation it may be appropriate to start with a manual effort, the techniques are the same.

The Automated SPC System

In the past, SPC systems involved a significant effort to put in place. The only automated systems capable of delivering the necessary power also required a staff to first implement the system and then for maintenance and upgrades. Furthermore, the "experts" that were assigned to the project typically didn't have the experience to define and implement the entire system.

To prepare for the use of SPC, many firms believed that statistical training would enable them to determine what was needed and how to do it. Engineers were learning all sorts of statistical rules and calculations, but nobody was learning how to make the techniques work for their operation. There was no "cookbook" formula for achieving the potential savings promised by SPC.

There is no requirement that SPC be implemented on a large and expensive computer system. The fact of the matter is that many successful SPC systems can consist entirely of manual SPC stations. The need for an automated system arises once the number of points to be monitored or the desired calculations render the manual system too slow or expensive.

SPC techniques are best implemented by starting small and showing success at each stage of the implementation. Getting assistance from someone who has implemented SPC before greatly increases your chances of success and the development of a system which can grow to meet your needs in the future.

A Building Block Approach to SPC

Now we can look at the implementation of an SPC system in a manufacturing environment. When you look at a process to be monitored, there are essentially three tasks to be performed; data collection, data reporting and data analysis.

Each of these tasks are equally important, whether your system is comprised of people filling out forms on paper and using a calculator to do the desired calculations, or you have a sophisticated computer system that performs all of the functions automatically.

In fact you may already be using some SPC techniques in a manual form today! As stated before, the best approach to implementing SPC is the systems approach: Start small with big ideas. Begin with a well understood section of your process, show success, and then grow and expand your system, building upon the knowledge that you have acquired.

Hardware - HP9000's

The wide availability of powerful computers such as the HP9000's provide the platform to accommodate the gradual introduction of an SPC system. The Series 300 computers provide an entry level machine that delivers the horsepower to handle a number of these functions at a cost well below traditional minicomputer or mainframe based systems.

The modular design of these systems allow you to upgrade individual pieces as the need arises. You will never have to be concerned about replacing your entire system because you have outgrown it. An individual station can be added without affecting the balance of the system. This makes a system based on the HP 9000 hardware a very low level risk for the budget conscious.

The Rocky Mountain BASIC Language System, available on the HP9000, is a capable and user-friendly environment developed by HP. Over the years it has been optimized specifically for the Measurement Automation market. Ease of use and speed for computation and data acquisition make an excellent choice as a platform for your SPC system.

Application Software

Like any other computer application, the key to developing a successful automated system is the software. The software which performs the data acquisition, statistical analysis and reporting functions must be modular, much like the HP9000 hardware, in order to allow for the growth and evolution of your system. For many reasons including, simplicity and reduction of human errors, the same software should be used at all SPC stations regardless of it's location within the process.

Acquiring the applications software can be the most difficult and frustrating part of implementing an SPC system. Until the last few years, the most practical option available for acquiring SPC application software was to write your own. Today there are standard software packages available to perform various SPC functions, eliminating the need for the SPC implementer to also be an expert programmer.

The SPC implementer must define what data is important and how it can be acquired but should not have to design data structures and write code to store and retrieve the data. You should plan to specify the types of data reports which are required from the system, but you should not have to write a program for each type of report that is desired. Statistical analysis functions should be provided by the standard software, as well as human interaction features, without requiring programming on the part of the SPC implementer.

A Relational Data Base Management System (RDBMS) provides versatile data management to enable the collection, analysis and reporting functions to be performed in a very efficient manner, without the need to program. The implementer must simply define what is desired and let the software do the rest.

Each of the software building blocks used for data collection, data analysis and data reporting should allow for smooth integration of additional functions without the need to modify what already exists. Sharing data through the Relational Database allows you to select any of the available data, the software modules will handle the manipulations to generate the desired charts and reports.

A common language for all SPC computers and functions will simplify the installation and maintenance of the SPC system. Users and developers will find life much easier if they can learn one system and expect it to grow with their needs.

An SPC Example - PCB Manufacturing

To give you a feel for how this SPC system can be put to use today, for very little cost, and grown to meet future needs, I will go through an example. In the interest of time and effort, I will focus on an example that shows how a printed circuit board assembly line could begin to utilize SPC. From there I touch upon the global picture of where the system could go as it evolves.

When you look at a printed circuit board (PCB) manufacturing operation, you can see essentially two activities - assembly and inspection. Each of these has a number of supporting functions that are certainly important but for now we will look at these as simple examples. From this you will be able to see how the supporting functions can be addressed in the same manner.

Assembly is comprised of activities such as component sequencing, automatic insertion, manual insertion, wave soldering, touch-up, etc.

The inspection portion covers activities such as receiving inspection, bare board testing, component testing, assembly inspection, in-circuit test, functional test, etc.

Data As A Status Indicator

Each one of these activities has vital information that is pertinent to the overall health of the process. Just like in a continuous process, each of these activities will reflect what is occurring in the manufacturing process.

There are a number of these points that could be monitored along the manufacturing line. These points all have data that may be necessary to your SPC system.

When you view each of these points as a status indicator with a defined set of conditions that are acceptable, they can be used as indicators of the condition of the process.

The data can be acquired in a number of different ways depending on the area.

In receiving inspection, some data such as part numbers, quantities and purchase order numbers can be entered by bar code readers or through keyboard entries. Bare board test and component test information can be acquired, in many cases, directly from automated test systems.

Sequencing and auto-insertion information can be entered by bar code, keyboard, etc. Wave solder can provide

temperature, speed and other information that can be acquired automatically or through manual input.

Functional and in-circuit testers can typically supply test result data through automated means. Inspection results can be input through the bar code readers or manually through keyboards. The results and findings in the rework or repair activities can be entered in the same fashion.

When you look at these different kinds of data and the varying sources, there may appear to be an overwhelming number of ways that this uncommon data would need to be gathered and organized. In the actual task of data acquisition there are really a limited number of ways to accomplish this. They boil down to two methods, manual or automated.

Storing the collected data in a RDBMS enables it to be analyzed either on-line or off-line. On-line you can analyze data from various sources to produce alarms or other control aids.

Off-line you can review the data to produce summary charts and reports, or to send status information to other Management Information Systems (MIS). Any of these can be performed independently or at one of the SPC stations.

In fact all three of the SPC functions, data acquisition, data analysis and data reporting, can be performed either manually or automatically and at a single station. It will depend on the stations and the responses that are desired.

Manual Or Automated The Critical Factors

The critical factors that should be considered when deciding between the two methods is - time and money. Time to collect, sort, analyze and report. Time to train the people that will be implementing the system and the people that will be using it.

Money to provide manpower, to purchase the needed equipment, to spend on scrap and rework. The choice is yours. You must look carefully at the alternatives.

Investigate the pro's and con's of each alternative. And be sure to look at what future needs you will have. A system should allow you to add functionality as it is needed.

With a system developed using the HP9000, you can add the functions and processing power as needed. The SRM (Shared Resource Manager) is a proven method of connecting the SPC stations.

Start With An Existing CPU

You can start with a HP9000 that has been used as an instrument controller, perhaps in the final test area. Many test systems today use the HP9000 as a controller to measure the acceptability of a product. If it passes the test, move it on and start the next test.

This data that is being used to judge the product can also be used to monitor the condition of the process. Much of the data that is thrown away after a test, could be utilized for SPC.

At the hub of the SPC system is the task of data management. It can be as simple as keeping sheets of paper in a filing cabinet or as sophisticated as needed to collect, sort and store information from every area of the manufacturing process.

In an automated environment, this data management can be a difficult procedure. The term "Islands of Automation" came about through the experience of many firms where computerized test systems have been established without concern for sharing of information. Many of these systems have hardware that cannot be easily connected. Usually the file formats, at least, are incompatible.

A key ingredient to avoiding or solving this data compatibility problem is the database management system that is used. Relational data base management systems are the latest technology that enables you to share data and add functionality without concern for the physical data management task.

An RDBMS enables you to keep your data completely independent of the applications that are collecting and analyzing it.

No matter who writes the individual applications programs, it is easy to access that data and use it for statistical analysis or process tracking. You can eliminate those "Islands of Automation", by simply having each "island" send it's data to the appropriate table(s) in the RDBMS.

Seven Steps to SPC Implementation

Lets now look at the process of implementing SPC in the real world. This is not intended to be a shopping list approach or step by step directions for every case. You must carefully consider your application.

This example will give you an idea of how to address the implementation, but remember, the philosophy is key to any successful SPC system.

There seven steps that should be taken when you begin your SPC development.

- 1) Pick a section of the production process.
- 2) Develop an accurate process flow chart.
- 3) Select the "SPC stations".
- 4) Identify what is needed at these stations.
- 5) Select the tools needed at each station.
- 6) Establish a training program.
- 7) Pick the day to get started implementing.

1) Pick a section of the production process.

When you look for the section to start with it is important that you choose an area with willing partners. The first few stations will be critical to the overall success of the program.

Look for production bottlenecks, large scrap or rework areas. Try to find an area that will remain stable long enough to finish. Finally, keep the number of stations realistically limited.

Whether you will start with a manual system or go directly to a computerized system, it is important to be able to do it in modules, small pieces that will keep the size of effort reasonable. Show success with the first stations and the gradual introduction into new areas will be much easier.

2) Develop an accurate process flow chart.

The process flow chart must be an accurate representation of the production process. It is important to study the situation carefully to identify all of the steps involved in your operation, even the ones that no one wants to own, the rework and holding areas that everyone tries to ignore.

This is a simplified flow chart for the PCB manufacturing example we addressed earlier.

RECEIVING INSPECTION

COMPONENT SEQUENCING

AUTO INSERTION

MANUAL INSERTION

INSPECTION
WAVE SOLDER
TOUCH-UP
INSPECTION
IN-CIRCUIT TEST
FUNCTIONAL TEST
FINAL INSPECTION

3) Select the "SPC stations".

Selection of the individual SPC stations should take into account a number of factors that have been pointed out. From the flow chart, you can select the most reasonable areas to start.

For our example we will look at the post manual insertion INSPECTION and FUNCTIONAL TEST areas. These will give a good representation of differing requirements for SPC stations.

RECEIVING INSPECTION
COMPONENT SEQUENCING
AUTO INSERTION
MANUAL INSERTION
INSPECTION SPC STATION
WAVE SOLDER
TOUCH-UP
INSPECTION
IN-CIRCUIT TEST
FUNCTIONAL TEST SPC STATION
FINAL INSPECTION

4) Identify what is needed at these stations.

The next step is to identify the needs for the SPC STATION. Factors that should be considered are:

- A) Manual vs Automated
- B) Available Budget
- C) Ease of Use

A) Why should you choose a manual system? A manual system is generally a reasonable way to get a feel for SPC, what you will need to do and what you can expect from an SPC system. If the necessary data analysis can be performed manually in 1 hour or less, a manual system may be reasonable. A manual system might be a good choice to do preliminary investigations or to test the techniques you will be using.

Reasons for choosing an automated system that should be noted are the speed of data collection and analysis, and the simplicity of data management and reporting, fewer errors and the wide variety of analysis techniques that are available when desired.

B) Often the available budget is the most critical factor in the system you will implement. If you have unlimited resources available you can plan out an organized effort that will lead to a very sophisticated system. More typically you will start with a system that has a mixture of both manual and automated stations.

C) The ease of use question is one that can be difficult. If you look at our example, the Inspection Station can be accommodated with manual data collection. The analysis of this data may or may not be easily performed manually. This will depend on the volume of boards and the analysis that will be performed.

The Functional Test Station would be more difficult to perform manual data collection and reduction because of the large amounts of data. Often an automated tester can be utilized as the data collector and sometimes you can even add the analysis and reporting functions.

5) Select the tools needed at each station.

Identifying the tools to be used at each station involves a study of the types of analysis to be performed and is again related to the manual vs automated station. The most common tool for the SPC system is the Shewart Control Chart, sometimes called "Trend Charts". Shewart Control Charts are

basically extensions to the standard trend chart, which enable you to see, at a glance, just how far from normal each sample of measurements resides.

To use this chart at the post insertion inspection SPC station you could track the number of boards rejected per shift or the number of missing components per thousand or whatever is a common problem. Over a period of time you will begin to see a pattern to the results. You can watch the pattern and clearly see the natural fluctuations.

From this, the control limits can be set. When the points go out of the control limits, that is a sign that something has gone wrong in the process.

The product failure is an indication of a failure in the process, a machine that is in need of repair, a new worker that requires additional training, part shortages, or parts out of tolerance.

You can track the differences between machines, operators, shifts, assemblies, there is no limit to the number of areas that can be tracked.

A point out of range is easy to see. It gives an immediate indication of present trouble spots and an early warning of developing problems.

Once these variations have been manifested on the charts, it is possible to establish what the significant factors are that influence the smooth running operation. A useful technique at this point would be bar chart showing the number of defects of each type. If the bar chart is ranked by the number of defects, it is called a Pareto chart.

The clear presentation of the most common types of defects, combined with an analysis of circumstances when they most often occur, takes the manager one more step in the direction of making process corrections that are on target.

Other commonly used charts are:

- X-BAR and RANGE CHARTS
- X-BAR and SIGMA CHARTS
- BAR CHARTS
- HISTOGRAMS
- PARETO CHARTS
- P and nP CHARTS
- SCATTERGRAMS

Each of these has particular strengths. Just remember that they will each take training and experience to produce and

use effectively.

An automated system that provides the capability to perform a wide variety of calculations allows you to experiment. Teaching and performing these in a manual system can be rather expensive and time consuming, and often the results are too late to be pertinent.

6) Establish a training program.

Establishing a training program is another very important point in the implementation of an SPC system. In a manual system the individuals must be trained on how to perform each of the calculations, how to maintain the database and how to produce the desired charts. In many cases it is difficult to find people that want to learn the statistics that are required, and would be willing to do this on a regular basis.

With an automated system, statistical training is often not necessary. Operators must be trained on how to enter the data and detect warning signals. Engineers and 1st level supervisors need to know trend analysis and basic chart generation, but the computer does all the calculations and plotting.

When deciding between manual and automated systems keep in mind that results come from the implementation of SPC rather than the math and statistics training. In many cases you can spend more on the training required for a manual system than to purchase a good statistical analysis package.

7) Pick the day to get started implementing.

Picking a day to get started requires obtaining the agreement of all involved. It takes a commitment on the part of everyone. It takes a change in philosophy starting at the management level and moving right down to the production worker.

All of these people must be involved to make the program successful. All should be taught about the program and informed how it works and how it will be used. If just dropped on the work force, you can be virtually assured of resistance.

SPC works best when each worker keeps his own "score" and is directly involved in his own performance. Seeing the progress of the lines on the chart becomes a great motivator to do better the next time.

Ford Motors and HP have both found that production workers

have offered a fund of ideas for improving the production process.

Even though SPC sometimes focuses attention on an individuals work, it seldom points the finger at any one individual. People at every level are often afraid that an SPC program will expose their incompetencies. So it's important that everyone understand the purpose of SPC before the charts are put up. In this way, each person can look at the design of the system with an eye to improving processes in his or her area.

This involvement leads to the ownership of an area of work and is strong motivation for improvement. The idea of looking at product failures as process failures establishes a mandate to look at the process critically and to come up with creative solutions. It can lead to a high degree of involvement from workers. It also calls for managers that are willing to regard leadership as the ability to listen and make changes according to what the workers observe.

So, if all of this sounds good, but you are saying to yourself, "I'm already too busy to do the work I have now, much less learn new techniques and take on more". Remember, the easiest and often most effective way to begin your SPC implementation would be to work directly with someone that is familiar with SPC and has been there before.

This approach can help you to avoid many of the common pitfalls. Your SPC expert should have knowledge not only of the SPC philosophy but of how to actually implement SPC in your plant. This requires a thorough knowledge of the available standard "off-the-shelf" pieces that can be put together to solve your individual puzzle.

The pieces should provide an efficient solution, based on your needs and resources today, with an eye toward the future. A future that will include additional needs and new functions to spread SPC throughout your entire operation.

**DISTRIBUTED DATABASE MANAGEMENT
FOR MANUFACTURING AUTOMATION**

**Lori Mooney
ADVANCED MICROSOLUTIONS
2510 Middlefield Road
Redwood City, CA 94063
(415) 365-9880**

Automatic test systems are the tools of choice for identifying defective products. With the emphasis today on quality, productivity and reduced cost, manufacturers are concerned with identifying defective process steps as well as defects in products. Identification of process defects requires analysis of data from a multitude of sources.

Information from every portion of the process must be integrated into a useable information system. From R&D to final product test, each step has valuable information that can be shared to improve the production process through better control. This data must be accessible to everyone that makes decisions about the product or process.

Integrate Your Manufacturing Data with a Distributed DBMS

A distributed Database Management System (DBMS) is a practical approach to successful integration of data and the associated computers throughout your manufacturing process. **It provides a method to link design, process or assembly monitoring and test data for use throughout your facility.**

MRP (Manufacturing Resource Planning) systems have been linking financial data (inventory, purchasing, order processing, etc.) for years. Factories being built today are striving to attain a JIT (Just In Time) manufacturing philosophy which promises a more efficient and cost effective operation. This requires accessibility to product and production data as well as financial information.

Control Your Manufacturing Process for Increased Profit

All manufacturing companies are striving to achieve increased profits through:

- * Higher yields
- * Improved quality
- * Reduced inventory
- * Higher efficiency
- * Reduced cycle times
- * Increased flexibility
- * Reduced cost

These are some of the benefits that can be realized with a manufacturing process that is in control. When a process is in control, it produces a product that is consistent and, if designed properly, a product that is of acceptable quality.

This control requires analysis of extensive quantities of detailed data related to your process. In most cases the volume of information requires an automated management system to organize the data and make the information available efficiently. To ensure that the information can be shared and to avoid the "island of automation" syndrome, a distributed database management system will satisfy today's needs and provide a growth path for future requirements.

In a computer integrated manufacturing (CIM) environment, not only is it necessary to have the traditional functions of MRP or JIT and MRP II in place, it is also essential to have coordinated and consistent database management at the lower work cell and cell controller levels.

Critical Data Is Available On-line

Significant amounts of critical data is available on the factory floor. To avoid costly downtime, rework and scrap, this data must be available in "real time". It must also be accessible for production personnel, process engineers, quality engineers, design and development engineers and management.

On-line availability is crucial for these individuals to act with certainty when instituting manufacturing and design changes to meet the goal of increased profitability.

Avoiding The "Islands of Automation" Syndrome

How can this important information be managed with the existing "islands of automation" in today's factories? The various and often dissimilar controllers and computers in use today can present a significant roadblock to this integration of data.

Even if you are fortunate enough to have standardized on a single controller and computer, generally at least the file formats for storing the data are incompatible. How can Incoming Inspection data be related to Test data and Inspection data to Quality Analysis data? And then, how can all of this information be linked to Engineering Department's design data?

A distributed DBMS is the key!

Using HP 9000's For Your Distributed DBMS

A distributed DBMS allows you to place the computing power where it is needed. You can add functionality or horsepower without affecting other stations. The HP 9000 family of computers provide a wide range of capabilities and the connectivity required to effectively address distributed database management.

Since many HP 9000 Series 200 and 300 computers are already in use performing a variety of testing, design and data acquisition functions, why not start by making the information currently being collected available to more people.

Even if you do not currently have HP 9000's in place, this is a viable system for implementing a distributed database management system from the ground up. Why the HP 9000 instead of a DEC VAX or an HP 1000 or a network of PC's?

There are a number of reasons for giving the HP 9000 serious consideration.

The POWER is there:

With the modular structure of the 9000 family, computers with a variety of cost/performance ratios can be chosen based on the needs of the particular area to be automated. Built around the Motorola 68000, 68010, and 68020 processors, the 9000's have the horsepower to handle even the most demanding applications.

From the inexpensive Model 216 up to the latest offering, the powerful Model 350, you have the option to choose a "fast PC" on up to a computer with 3 times the power of a VAX 11/780. As your needs grow and change, you can easily upgrade processors without reinvesting in duplicate hardware and new peripherals.

Software, an often overlooked component, can be the most expensive and difficult piece to put in place. Software compatibility is a major strength of the 9000 family. When developed properly, software is transportable across the entire family.

The CONNECTIVITY is there:

HP's SRM (Shared Resource Manager), although not a true local area network (LAN), enables your HP 9000's to share peripherals and data. In particular, the SRM enables you to share large disc drives containing your distributed database

and provides access to many users in different areas.

The SRM is even available to Vectra/PC's, using the Viper Co-processor, for less time critical data collection and analysis tasks. Data can also be shared among multiple SRM's and interfaced to other computers such as the HP 1000, HP 3000, DEC VAX's and mainframe computers along with various Local Area Networks.

You can also run UNIX workstations on the SRM. A workstation running HP-UX (HP's version of UNIX) can provide a multitasking environment and access to a multitude of applications, networks and software packages.

The FLEXIBILITY is there:

Besides the modular nature of the HP 9000 hardware, there are several operating systems available and a significant amount of "off-the-shelf" application software products that enable you to gradually add features and functions to your distributed database system.

Some of this software, available as modular building blocks, enable you to expand your system as desired. Often you can even add stations and modify test procedures and data reports "on-line" without affecting the on-going operation.

The USABILITY is there:

The HP 9000's and SRM are proven - they work when you plug them in. You won't need to spend months or years making your network work. The hardware is all from the same vendor and best of all - it works. Hardware maintenance, although minimal, can be handled easily. HP hardware reliability and service is established and well respected.

What is even better, if a standard and well designed software system is used, the faulty computer or peripheral can be replaced in minutes by simply plugging in another unit. Any configuration changes should be accomplished in minutes. System down time is held to a minimum.

The PRICE is right:

Both the installation and on-going costs of a system based on the HP 9000 are controllable and predictable, and best of all - they are reasonable!

As mentioned previously, the modular HP 9000 family provides a wide range of price/performance choices and a clear upgrade path to the higher performance machines. The workstation environment of Rocky Mountain BASIC is a

friendly and capable language for use by individuals of varying backgrounds.

There is no need for extensive system preparation or special operating environments. No system generations are required at installation or when peripherals are added or changed.

When standard software building blocks are used, there is no need for a dedicated programming staff - most engineers can do the necessary computer work.

Implementing a Distributed Database Management System

A gradual introduction strategy is possible with a modular system. Development can be a continuous process. You will see results quickly. There is no need for weeks of training and months or years of development. You can gradually introduce workstations and then add stations as you learn more about your manufacturing process.

Those involved in the development and use of the system can learn as they go, making enhancements and modifications as they are needed. There is no fear of missing an important item in the specification of the system or hoping to cover all possible future needs when the initial specification is issued. New processes and changes can be made by the people that know best what is needed.

The data and analysis options available in a distributed database management system for manufacturing enables you to accomplish tasks like:

- * Identifying production process problems
- * Accessing production statistics like yields, production volumes, test times, etc.
- * Monitoring vendor quality
- * Maintaining and enhancing test plans and procedures

Three Categories of Manufacturing Data

Three categories of information are required to optimize manufacturing productivity - Product, Process, and Production data must all be collected and managed. The DBMS should provide the capability for any individual needing information to easily access it, on an ad-hoc basis, and have the results presented in a meaningful format.

PRODUCT DATA

Product data consists of both product design and specification information. Information such as parts lists, measurement and test descriptions and procedures, product

specifications and measurement and test limits for both parametric and attribute results can be stored. Some information may be acquired from CAD systems, or from R&D and Engineering groups. Manufacturing uses this information to analyze defects by part number or location as well as to monitor and enhance test plans and evaluate vendor quality.

PROCESS DATA

Process data contains a list of the production steps and the measurements required at each step. This information is used for tracking work in process (WIP) and for optimizing the production process by analyzing bottlenecks and downtimes. The type of equipment used and it's associated calibration information can be managed as well in order to produce calibration schedules and maintain equipment.

PRODUCTION DATA

Production data is collected as each part is built and tested, measurement results and often test results and summary statistics are stored. This data is typically stored by lot number and it is the fundamental data used to determine if the production process is statistically in control. This data is also used to calculate yields, production volumes, and track a part's pedigree.

THE KEY TO SUCCESS - Your Database Design

Whether you are producing wafers, boards, or entire assemblies, you can use a distributed DBMS to improve manufacturing productivity by collecting and analyzing the appropriate data.

The key to the successful implementation of a distributed DBMS is the design of the database. Often called the database "schema", in a relational DBMS the "schema" consists of a list of the columns of data to be stored in each data table. State-of-the-art DBMS's are based on relational technology enabling you to view your entire database as a set of flat, two dimensional tables. Each table can be viewed, and manipulated, in a way much like you would use a spreadsheet on a PC.

Factors For Successful Database Design

Many factors must be taken into consideration as you layout the data tables for your distributed DBMS. Remember that any user at any station on the SRM can access data from any table, assuming the user has the appropriate password. Different "views" of the database may be defined for particular users as well, either for security or for ease of

use. Laying out the tables so that your application programs can be as independent of the table layouts as possible is the goal for ease of program development and maintenance.

System performance and resource utilization must also be taken into consideration. Storing all data for each category in one large table reduces the amount of database design required, but it also results in redundant data storage, slower response, more complicated queries and less efficient use of resources.

Determinations must be made as to the type of data to be stored. Will all measurement data be stored or only test results or summary statistics? Will all parts be tested or will a sampling plan be used to test a limited number of parts? Speed of acquisition and disc storage time must be considered, but with the HP 9000, these are not typically the limiting factors.

An Example - A Distributed Database on the HP 9000

Following is an example of a distributed database for a hypothetical manufacturing environment. This system is easily implemented in the Rocky Mountain BASIC environment of the HP 9000 using the SRM network to provide the distributed capability.

You can start off as simply as desired and easily add features by changing your database later. Ideally, your DBMS should allow you to alter your database with little or no changes required to the application software.

Using SQL - The Standard Relational Database Language

The examples below show how you can use Structured Query Language, or SQL, within your Rocky Mountain BASIC application programs to implement your distributed DBMS.

SQL is the ANSI standard relational database language. Developed by IBM, and currently in use by the majority of relational DBMS suppliers, this non-procedural language enables you to develop application programs in a fraction of the time required using traditional, third generation languages. You specify WHAT you want to do, not a procedure describing HOW you want it done. The SQL language then determines the best "procedure" for what you want to do. That is, you specify SELECT YIELDS FROM PRODUCTION WHERE MONTH = JULY rather than writing a subroutine to perform the procedure.

Add SQL to Rocky Mountain BASIC

Distributed Database Management -7-



In addition to reducing programming time, an SQL command is executed like a line of Rocky Mountain BASIC, at a speed fast enough to handle a distributed manufacturing DBMS. No system programming is required. You just design your tables and then develop your application programs.

As an alternative to "embedding" SQL commands in your application program, most SQL implementations provide an interactive, forms-driven system for end users. These systems require no programming! The only requirement is that a database be designed, or available to the user on-line. These systems are most useful for off-line table creation and ad-hoc queries and reporting.

SQL - A Single Keyword For ALL Queries, SELECT

The keyword for the query operation in SQL is SELECT. All queries are performed by issuing one or more SELECT commands. The SQL SELECT command gives you the capability to access ANY data stored in ANY table. You just indicate what columns are to be "joined" and retrieved and which records (or table rows) are to be retrieved.

Examples of Using SQL in Rocky Mountain BASIC Programs

The following examples show how SQL can be used in your Rocky Mountain BASIC programs to implement your distributed DBMS.

To design the database, first list the items to be stored in each of the three categories:

<u>PRODUCT</u>	<u>PROCESS</u>	<u>PRODUCTION</u>
Part_#	Step_#	Lot_#
Part_description	Description	Wafer_#
Department	Instructions	Time_tested
Test_#	Temperature	Site
Test_name		Result
Min_limit		
Max_limit		

Once you have listed all of the information to be stored, break each category up into additional tables, as needed based upon the amount of data to be stored for each item, the frequency with which the table is used, the relationships between the data items, etc.

If you have trouble with this step it often helps to draw the tables out on paper with actual or dummy data and change the tables around until they meet your needs. Several iterations are usually required for an optimized design.

Remember to pay careful attention to any relationships between the data values, that is, between one-to-one, one-to-many, and many-to-many relationships. Note that no linkages or connect points are actually stored in a relational database, these "joins" between tables are temporarily defined only when you have interactively or programmatically executed a query operation.

Shown below is one way that the data tables can be laid out to store the information described above in a distributed database:

PRODUCTS

Part_#	Part_description	Department
5247A	Oscillator	47 - R & D
5999B	Amplifier	48 - Engineering
.	.	.

TESTS

Test_#	Test_name
27	Length
28	Width
29	Thickness
.	.

TEST SPECS

Part_#	Test_#	Min_limit	Max_limit
5999B	27	.0057	.0077
5999B	28	.0125	.0250
5999B	29	12	13.2
.	.	.	.

PROCESS

Step_#	Test_#	Temperature
1	29	0
2	29	-40
3	29	40
4	27	25
5	28	25
.	.	.

STEPS

Step_#	Description	Instructions
1	Oxide A - MID	1) Set to 0 deg 2) wait 10 min
2	Oxide A - LO	1) Cycle to -40 2) wait 20 min
3	Oxide A - HI	1) Cycle to +40 2) wait 20 min
.	.	.

PRODUCTION

Lot_#	Part_#	Wafer_#	Step_#	Test_#	Time_tested
1560	5999B	378	1	29	870710.0815
1560	5999B	214	1	29	870710.0832
1560	5999B	378	2	29	870712.0930
1560	5999B	378	3	29	870714.1432
.

LOT 1560

Time_tested	Site	Result
870710.0815	A	12.3
870710.0815	B	12.6
870710.0815	C	12.9
870710.0815	D	13.1
870710.0815	E	12.8
870710.0832	A	12.7
870710.0832	B	12.6
870710.0832	C	12.7
870710.0832	D	12.9
870710.0832	E	12.5
870712.0930	A	12.1
870712.0930	B	12.3
870712.0930	C	12.2
870712.0930	D	12.1
870712.0930	E	12.1
870714.1432	A	12.7
870714.1432	B	12.9
870714.1432	C	13.2
870714.1432	D	13.1
870714.1432	E	12.9
.	.	.

You can create your database interactively or within your program using the SQL CREATE TABLE command. This command is especially useful for production data. For example, if your lot numbers automatically change on a weekly basis, you can

use SQL to create a new LOT_xxxx table every monday morning by issuing the following command:

```
Sql("CREATE TABLE LOT_1560 (Time_tested TIMEDATE, Site
  STRING[1], Result REAL) RECORDS = 500 BUFFERS = 50")
```

During production, your data collection program uses the INSERT command to add the measurements to the appropriate lot table:

```
Sql("INSERT INTO LOT_1560 ( 870710.0815, A, 12.3 )")
```

The program running at a Rework station can use the update command to change the results for a given test to reflect the new measurement result:

```
Sql("UPDATE LOT_1560 SET Result = 12.8 WHERE Time_tested =
  870710.0815 AND Site = D")
```

Queries are easily performed interactively or within your program by issuing the SQL SELECT command. For example, to generate a current WIP REPORT showing the process steps and test results for Wafer No. 378 of Lot No. 1560, issue the following command:

```
Sql("SELECT Part_#, Step_#, Test_#, PRODUCTION.Time_tested,
  Site, Result FROM PRODUCTION, LOT_1560 WHERE
  PRODUCTION.Time_tested = LOT_1560.Time_tested AND
  Wafer_# = 378 AND Test_# = 29")
```

When executed, the PRODUCTION and LOT_1560 tables will be joined to produce the following report:

Part_#	Step_#	Test_#	Time_tested	Site	Result
5999B	1	29	870710.0815	A	12.3
5999B	1	29	870710.0815	B	12.6
5999B	1	29	870710.0815	C	12.9
5999B	1	29	870710.0815	D	12.8
5999B	1	29	870710.0815	E	12.8
5999B	2	29	870712.0930	A	12.1
5999B	2	29	870712.0930	B	12.3
5999B	2	29	870712.0930	C	12.2
5999B	2	29	870712.0930	D	12.1
5999B	2	29	870712.0930	E	12.1
5999B	3	29	870714.1432	A	12.7
5999B	3	29	870714.1432	B	12.9
5999B	3	29	870714.1432	C	13.2
5999B	3	29	870714.1432	D	13.1
5999B	3	29	870714.1432	E	12.9

Tabular reports can be generated automatically with the SQL SELECT command. Reports can be formatted as you desire in your application program by having the results of the SELECT command returned to your program in arrays for presentation in any manner you choose.

A distributed database management system for manufacturing that is based upon the HP 9000 and the SQL language will provide a good, solid system for years to come as well as being a system which can be implemented in a short period of time and grown to meet your growing needs.

Distributed database management enables you to overcome a major hurdle in the development of your CIM system.

HP9000 - IBM 4381 Communications
Argemiro Rodriguez Araujo
Occidental de Columbia
Apartado Aero 092171
Bogota
Columbia

CHAPTER I

INTRODUCTION

The scope of this presentation is to show the limitations we have found in the use of the RJE emulator as a Hewlett Packard product, and the way we have overcome the problems.

To achieve this goal, we will present a summary of the fundamental concepts as background, without pretending to cover all of telecommunications matters in detail.

To be more precise, we will frame the discussion in the environment of the HP9000 Series 500-TO-IBM4381 telecommunications, and the experiences gained in our installation.

Occidental de Colombia is a subsidiary of Occidental Exploration and Production Co. with headquarters in Bakersfield, California, devoted to the oil and gas industries in Colombia and producing an oil field of 1054 MMBBLS of proved reserves to 1/1/2008, and 196 MOBPD of production rate. To simulate the field, we are using ECLIPSE -a black oil simulator by ECL- which runs in an HP9000 installation.

There is also an IBM3090 in Tulsa holding the MVS/XA version of the same software package to serve as backup; and an IBM4381 installation in Bogotá being used to store and print files in faster peripherals, and to hold a petrophysical database which is updated via HP1000-HP9000-IBM transmissions among the multiple applications. This triggers the need to count on a mechanism to transfer files back and forth between HP and IBM machines.

CHAPTER II

HP-TO-IBM NETWORKS

This chapter presents a rough overview of HP-TO-IBM communication methods. Basically two kinds of IBM networks exist:

- o Binary Synchronous (BSC) Networks.
- o Systems Network Architecture (SNA) Networks.

These two networks are related to two kinds of system communications: batch and interactive communications.

Batch communications allows submitting jobs from a terminal, sending them to the host computer for processing and receiving their output back at the terminal. Batch communication is associated with the transmission of large amounts of data without immediate response. Interactive communication implies exchanging information with a host system and receiving immediate responses.

Let us discuss each of these concepts in following sections.

2.1 BSC NETWORKS

The Binary Synchronous protocol handles the flow of information over a line with some control characters to synchronize the two stations involved in the transmission, to identify the start and the end of data, and provide checking for transmission errors.

It was only in the mid -1970s that the first versions of binary synchronous protocol, also known as BSC protocol, appeared in the IBM sky, after series of trials to provide IBM networks with an overall structure. Now, BSC networks are wide-spread used in supporting batch and interactive communications.

2.2 SNA NETWORKS

The SNA network concept was developed by IBM in 1974 to give an answer to the lack of an overall network structure. After a gradual development, at the beginning of the 80s, SNA was more feasible to implement and started to gain popularity in the communications world. The introduction of Advanced Communication Function (ACF) allowed the real concept of multiple nodes.

SNA is basically made up of nodes connected by data links. A node can be defined as a set of hardware and software at the end of a data link, and can be broken down into the following components:

- o NAU Network Addressable Unit (as: System Services Control point - SSCP, Logical Unit - LU, Physical Unit - PU).

- o PC Part of Path Control Network (functions include: receiving information from a NAU, routing this information through the network, ensuring that the information

crosses the data links without errors, delivering the data to its destination NAU).

Concepts like domains, subarea nodes and peripheral nodes will be left to the user's study as they are out of the scope of this paper.

2.3 BATCH COMMUNICATIONS

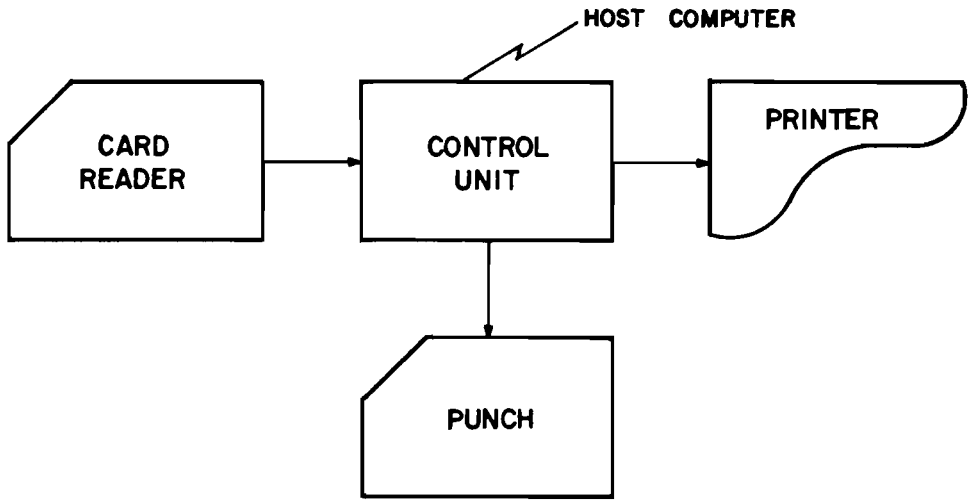
Emulation in the Hewlett Packard communications world refers to the communications product developed by HP that provides a terminal or HP computer with some of the functions of an IBM piece of equipment.

To perform batch operations, some Hewlett Packard machines emulate IBM RJE terminals.

Some of IBM's first devices devoted to communications were batch terminals, using a simplified version of BSC protocol. IBM presented its 2780 data transmission terminal in 1967, and the 3780 data transmission terminal in 1972.

The two of them basically have the following configurations:

IBM 2780



These machines allow the transmission of one job stream at a time in only one way from the terminal to the host computer or viceversa.

The 2780/3780 uses a contention binary synchronous protocol. In this type of protocol a host and a terminal bid for the use of the line on a first-come-first-served basis.

Information is transmitted either from the host or the remote, in a whole block and the receiving station responds to the transmission with acknowledgement characters to indicate the existence of error in the transmission.

This bidding process is repeated every time one station needs to transmit.

Hewlett Packard developed the DSN/RJE for the HP9000 Series 500 computer family. The host computer sees the remote terminal as having a reader, a printer and a punch, although these are actually logical devices in the remote side, communicated with the host via data streams as a path.

2.4 INTERACTIVE COMMUNICATIONS

With the interactive products you have interactive access to a host computer, for data entry and retrieval and time-sharing sessions.

The 3270 Information Display System is one of the most remarkable IBM products for interactive communications.

3270 devices can be central units, display stations or printers. Control units can operate either in BSC networks or SNA networks. The 3270 family holds many types and models of Central units like 3271, 3274, 3276. Some 3270 central units are cluster controllers. Also we can find several types of display station work: 3277, 3278 and 3279. Printers in this family are: 3284, 3286, 3287, 3288 and 3289.

CHAPTER III

BATCH COMMUNICATIONS: RJE

We will focus our attention on batch communication, as the HP9000 (Series 500) product to communicate with IBM is dubbed DSN/RJE and emulates IBM 2780/3780 with BSC protocol.

3.1 HP9000-IBM NETWORK

Let us switch from theory to the implementation of an HP9000-TO-IBM network as it was carried out in our company. In the following sections we will examine the configuration of the network in the portion pertinent to HP9000-IBM communication.

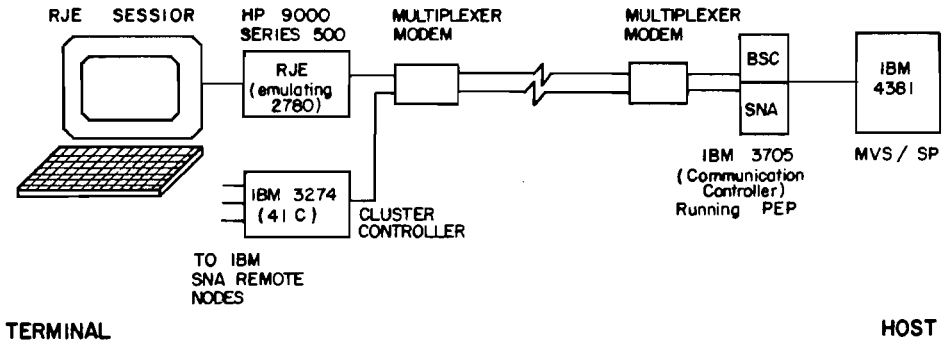
IBM SIDE:

An IBM4381 (from the System 370 mainframe family) is currently installed with 24 Mbytes of memory and the Multiple Virtual Storage/System Product Operating System.

A part of the operating system is the job entry subsystem which controls the input, execution and output of jobs by means of JCL. JES2 (Job Entry Subsystem 2) is installed within MVS/SP.

To handle the most important function in data communication there are communication controllers with specialized software, namely communications access

HP9000 - IBM NETWORK



methods. We use the IBM 3705 communication programmable controller which runs the Partitioned Emulation Program (PEP). This program enables the controller to handle BSC and SNA lines in the same time frame. NCP and JES2 identify the HP9000 as a 3780 terminal.

HP9000 SIDE:

The HP9000 Model 550, with HP-UX v5.11 holds the HP50968A Hewlett Packard product for RJE to emulate IBM2780/3780.

The HP9000 is equipped with the HP27122A upgraded card.

3.2 WHAT IS RJE IN HP9000 ENVIRONMENT

The RJE emulator provides you with the capability of transmitting files back and forth between the HP9000 and a mainframe in a batch mode for processing.

Again, the HP9000 and the RJE emulator configure a card reader, a printer and a punch from the IBM point of view to send and receive files. RJE translates files from ASCII to EBCDIC when transmitting files from the HP9000 to the IBM.

3.3 MODES OF OPERATION

To transfer information between the HP9000 and the mainframe through a data link, two modes can be used: r2780 and background. These modes are exclusive when

using the line, which means that, only one mode can be used at a time if there is only one RJE interface card.

CHAPTER IV

r2780 MODE

4.1 DESCRIPTION

In r2780 mode an interactive session is carried out to transmit files between the HP9000 and the host. This is the first approach made by Hewlett Packard in providing communications between these two machines.

The RJE is invoked by the r2780 command, which configures the communication link. Options can be set in this command, such as suppressing translation, tracing, input/output record size and others.

4.2 UTILITIES

Basically r2780 offers two utility programs:

- . rje-printfmt to clean of control characters a printout file received from an IBM.
- . rje-tracefmt to format information received from the tracer facility.

4.3 PROBLEMS

A procedure has to be followed when using r2780 mode in RJE emulation. The steps are described as follows:

1. Define the r2780 command with the options you need to use. It is recommended to save the command in a file to avoid re-typing in recurrent transmissions.
2. Send a signon card to the IBM. In our case this statement is /*SIGNON REMOTE13.
3. Send the command <@ to terminate the transmission.
4. Start sending or receiving information. The command starts with "<" for sending files, and ">" for receiving files.
5. Send a signoff to terminate transmission.
6. CTRL D to terminate transmission.

Problems detected when using this mode deal with:

1. The RJE operator has to be present all the time during the transmission. Wrong keystrokes are fatal and can abnormally terminate a long session. The transmission of large files is cumbersome.
2. Files sent to the host computer require JCL leading the data and JCL trailing it. Options set in the r2780 command to apply only to data apply also to the JCL; therefore, -b, -o, -i, generate problems when data and JCL do not have the same format.

3. The `-b` option to suppress character translation (from ASCII to EBCDIC), typed as a single option during the session, transmits a binary file to the host computer. The file has to be sent with `#:` command to allow transparent mode. This action works OK. Sending a binary file from the host to the HP9000 produces spurious information.
4. When sending a printout from the HP9000 to the IBM some alternate ways have to be worked out. If you set `-on` option in the `r2780` command, all of the statements including the JCL will be taken as having `n` characters. Option `-on` is not allowed in an interactive session as a single option, and sending the file with default options would cause truncation of the file.
5. Receiving a printout from the host computer to the HP9000 will require a further cleaning of control characters. The utility `rje-printfmt` does not clean the file properly. This problem is more serious when we are transmitting data for processing.

4.4 HINTS TO SOLVE THEM

1. Although the fully interactive way is required to set some options such as `b`, `c`, `i`, `k`, `t`, and `z`; unattended way for long files is recommended.

A script was written to allow sending files in unattended way as shown in Figures 1 to 5.

2. Figures 6 and 7 show how to embed JCL in a transmission of a printout.
3. The fourth problem mentioned above is solved by splitting the 132-character printout record into two records that the RJE can handle. Program "split" shows the approach. Then, a "merge" program is required for the IBM to set the record as was originally on the HP9000. See Figures 8 and 9. What is important here is the method implemented, not the compiler chosen. FORTRAN was used because of its simplicity. We encourage other users to improve it.
4. Problem 5 as mentioned before was solved with a filter program called "cnvrt". It was also developed in FORTRAN, and what it does is to get rid of scape sequences and control characters added during the transmission from IBM to HP9000. See Figure 10.

Figure 11 shows how the file appears on screen when we use "vi", right after the file is received from the IBM. Control characters are shown with "vi" but do not appear when we "cat" the file (See Figure 12).

The utility rje-printfmt does not eliminate all of the control characters. Figure 13 shows how the file shows up with "vi" after filtering it with "rje-printfmt".

Therefore a filter program to fully remove undesirable characters is required. The program

shown in figure 10 is really a simple way to get rid of these characters but it is not the only approach. Again we encourage users to develop alternatives.

Figures 14, 15 and 16 show script variations to catalog a file sent from the HP9000 on IBM mass storage.

CHAPTER V

BACKGROUND MODE

5.1 DESCRIPTION

This a more elaborate and automatic mode of transmitting files back and forth between HP9000 (550) and the mainframe.

Files are queued for transmission even without an RJE link previously established. Once the link is initiated, files are sent in background, and its process can be traced. The RJE administrator initiates the background process, and then the process examines the existence of files to be sent to the host, and sends the files it finds.

The background process also receives files from the host and routes them if a destination is indicated.

The background process performs the same basic functions as the r2780 mode. More files and utilities are involved in this mode.

5.2 UTILITIES

Commands allowed are:

- rje-send to queue files for transmission.
- rje-init to initiate the RJE background.
- rje-halt to stop the RJE background process.
- rje-stat to check the status of the link and your files.
- rje-printfmt to format files received from the host.

Details of these commands and modus operandi are found in the RJE User's Manual - Chapter Five.

Files involved are:

- rje-acctlog (transaction file),
- rje-config (holds the link configuration).
- rje-errlog (log of errors).
- rje-joblog (entries of files to transfer to the host).
- rje-jobqueue (files to be transferred to the host).

5.3 PROBLEMS DETECTED

1. Files with record longer than 80 bytes are truncated.

The option to set the record length, goes in rje-config command, and it is not allowed in command lines, therefore it would apply to every line including JCL being transmitted to the host.

2. Files received from the host carry BSC control characters. The rje-printfmt command does not remove these characters.

5.4 HINTS TO SOLVE THEM

For a link configuration as the one shown in figure 17, the following solutions were worked out for the problems mentioned.

1. A script was developed to embed JCL in the rje-send command, as Figures 18 to 21 show. Files in Figures 20 and 21 are used to catalog data on IBM discs. To solve problem No. 1, the JCL in Figures 6 and 7 can be used, as well as programs in Figures 8 and 9 to split and then merge records.
2. Problem 2 is also present in r2780 and it is not solved in background. The solution given to problem No. 5 in r2780 is valid for background. Use program "cnvrt" or another one designed to eliminate control characters.

CHAPTER VI

CONCLUSIONS

- Transmissions between the HP9000 series 550 and the host IBM 4381 can be created with the RJE emulator program.
- The r2780 mode, as the first version of RJE for the HP9000, works fine for those cases when you want to keep an eye on the process. The background mode offers an improved method for transmission, but both of them still have bugs when dealing with longer-than-80-byte records.
- The solutions we provided to the problems detected are simple ones for simple problems, but not unique methods.
- We hope to contribute to ease the use of RJE to those users working with HP9000-IBM 4381 communications.

REFERENCES

1. COMMUNICATING WITH IBM, Hewlett Packard, 1984.
2. RJE USER'S MANUAL FOR HP9000 COMPUTERS. HP Part No. 50966-9000.

```
//TAXR001 JOB (UNBGAXR,USERTS01),'HP 9000 TO IBM',CLASS=A
//*PASSWORD XANADU
//*JOBPARM Q=F
//STEP1 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSIN DD DUMMY
//SYSUT2 DD DSN=&&WORK1,DISP=(,PASS,DELETE),
// DCB=(LRECL=80,BLKSIZE=6320,RECFM=FB),
// UNIT=SYSDA,SPACE=(TRK,(50,30),RLSE)
//SYSUT1 DD *
```

figure 1

File /comm/rjejcl containing the script to run rje
for transmitting/receiving printouts in r2780 mode

```
r2780 -3 -n0 -w0 -s4800 /dev/rje
```

figure 2

File /comm/trans containing r2780 command

```

echo "Type t if you are going to transmit"
echo "type r if you are going to receive"
read a
echo "Type the name of the file to be transmitted or received"
read b
if test "$a" = "t"
then
  if test ! -s $b
  then echo "The file $b does not exist"
  exit 1
  fi
fi
if test "$a" = "r"
then
  cp $b /comm/tempi.prt
  ( /sim/dave/split < /comm/commsplit > /comm/dummy )
  rm $b
  cp /comm/tempo.prt $b
  cp /comm/ibmjcl1 /comm/output.prt
  cat $b >> /comm/output.prt
  cat /comm/ibmjcl2 >> /comm/output.prt
  cp /comm/seed /comm/commands
  echo "</comm/output.prt" >> /comm/commands
  echo "<@" >> /comm/commands
  cp /comm/tempi.prt $b
else
  cp /comm/seed /comm/commands
  echo ">$b" >> /comm/commands
  echo ">@" >> /comm/commands
  cd $HOME
fi

```

figure 3

File /comm/inputrje that contains the script to prompt input data for the transmission.

```

</comm/signon
<@

```

figure 4

File /comm/seed containing the name of the file which contains the signon and the terminator command

```
/*SIGNON      REMOTE13
```

figure 5

File /comm/signon containing signon

```
//TAXR001 JOB (UNBGAXR,USERTS01),'HP 9000 TO IBM',CLASS=A
/*PASSWORD XANADU
/*JOBPARM Q=F
//STEP1      EXEC PGM=IEBGENER
//SYSPRINT  DD SYSOUT=A
//SYSIN     DD DUMMY
//SYSUT2    DD DSN=&&WORK1,DISP=(,PASS,DELETE),
//           DCB=(LRECL=80,BLKSIZE=6320,RECFM=FB),
//           UNIT=SYSDA,SPACE=(TRK,(50,30),RLSE)
//SYSUT1    DD *
```

figure 6

File /comm/ibmjcl1 containing leading JCL

```
/*
//MERGEHP    EXEC PGM=MERGEHP
//STEPLIB   DD DSN=UNBGAXR.DAVE.LOAD,DISP=SHR
//FT05F001  DD DSN=&&WORK1,DISP=(OLD,DELETE)
//FT06F001  DD SYSOUT=A
//FT10F001  DD DSN=UNBGAXR.BE75.PRT,DISP=(NEW,PASS),
//           DCB=(RECFM=FBA,LRECL=133,BLKSIZE=3059),
//           UNIT=SYSTS,SPACE=(TRK,(50,30),RLSE)
/*
//STEP3     EXEC PGM=IKJEFT01,DYNAMNBR=25
//*****
//SYSTSPRT  DD SYSOUT=A
//DSPROSTQ  DD DSN=DSPRINT.REQUEST.QUEUE,DISP=SHR
//SYSTSIN   DD DDNAME=SYSIN
//DDNAME1   DD DSN=UNBGAXR.BE75.PRT,DISP=(OLD,DELETE)
//SYSIN     DD *
           DSPRINTB 'UNBGAXR.BE75.PRT' BG43U216 DDNAME(DDNAME1)-
           NONUM CCHAR
/*
//
```

figure 7

File /comm/ibmjcl2 containing trailing JCL


```

program SPLIT
character*80 record1
character*52 record2
character*24 infile,outfile
*   open (6,file='/dev/console')
*****
*
*
*
*
*   THIS PROGRAMS SPLITS THE 132 BYTE PRINT RECORDS
*   FROM ECLIPSE OUTPUT INTO 2 RECORDS OF 80 BYTES EACH
*   TO FACILITATE THE RJE TRANSMISSION.
*
*
*   PROGRAMMER :   DAVE KATRAGADDA
*   DATE       :   JAN 17TH 1986
*
*****
  write(*,*) 'enter the name of print file on hp9000
  .that needs to be transferred to an 80 byte file'
  read (*,'(A24)') infile
  write (*,*) 'enter name of 80 byte output file'
  read (*,'(A24)') outfile
  open (10,file=infile,err=50,iostat=ios)
  open (11,file=outfile,err=60)
  write(*,*) 'output file is opened please wait till
  .conversion is done'
10  read(10,'(a80,a52)',end=100) record1,record2
  write(11,'(a80)') record1
  write(11,'(a52)') record2
  goto 10
50  write(*,*) 'error in opening input file iostat no. = ',ios
  goto 100
60  write (*,*) 'error in opening output file'
100 close(11)
  close(10)
  stop
  end

```

figure 8

File /sim/dave/split containing the FORTRAN program to split the printout records into 80-character records.

```

program MERGE
character*80 record1
character*53 record2
character*24 infile,outfile
*****
*
*
*
*
*
*
*          PROGRAMMER :  ARGEMIRO RODRIGUEZ
*          DATE       :  JAN 17TH 1986
*
*****
      open (5,err=50,iostat=ios)
      open (11,err=60)
10  read(5,'(a80)',end=100) record1
      read(5,'(a53)',end=100) record2
      write(10,'(a80,a53)') record1,record2
      goto 10
50  write(*,*) 'error in opening input file iostat no. = ',ios
      goto 100
60  write (*,*) 'error in opening output file'
100 close(5)
      close(10)
      stop
      end

```

figure 9

FORTRAN program MERGEHP that merges data previously split and produces a 132-character printout for an IBM printer.

```

program cnvrt
character*1 inarray
character*2 blank
character*11 blank11
character*24 infile,outfile
dimension inarray(72)
open (6,file='/dev/console')
*****
*   this programs edits or filters the escape sequences arising
*   from the file transfer from ibm mainframe to HP 9000.
*   the escape sequences are part of printer control characters.
*   The program will prompt for the ibm file and also for the
*   filtered file.
*
*   PROGRAMMER :   ARGEMIRO RODRIGUEZ
*   DATE       :   OCT 1ST 1985
*
*****
write (6,*) 'enter the name of ibm input file to be filtered'
read (*,'(A24)') infile
write (6,*) 'enter name of filtered output file'
read (*,'(A24)') outfile
open (10,file=infile,err=50,iostat=ios)
read (10,'(A2,72A1)') blank,inarray
open (11,file=outfile,err=60)
write(6,*) 'output file is opened'
10 read(10,'(A11,72A1)',end=100) blank11,inarray
write(6,'(72A1)') inarray
write(11,'(72A1)') inarray
goto 10
50 write(6,*) 'error in opening input file iostat no. = ',ios
goto 100
60 write (6,*) 'error in opening output file'
100 close(11)
close(10)
stop
end

```

figure 10

FORTTRAN program "cnvrt" that filters a file received from the IBM.

```

$ vi /comm/d7pw.fortran
^Q
^I/**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-A
PR-87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
^I/**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-A
PR-87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
^I/**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-A
PR-87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
^I/**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-A
PR-87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
^I/**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-A
PR-87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
^I/**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-A
PR-87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
^I/00000000 C          DATA SET D7PW00001 AT LEVEL 001 AS OF 06/16/81
^I/00000010 C          DATA SET D7PW003L01 AT LEVEL 001 AS OF 01/09/79
^I/00000020 C          MAIN PROGRAM FOR CALCULATING PERFORMANCE AND
^I/00000030 C          PSEUDO RELATIVE PERMEABILITIES OF STRATIFIED RESERVOIR
^I/00000040 C
^I/00000050 C
^I/00000060 C
^I/00000070 C          NOTATION
^I/00000080 C

```

Figure 11

vi of the file as received from the mainframe

```
# cat /com/d7pw.fortran
```

```
**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
**** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
87 B381 R13.PR1 5.1.1      TSU 3354  START  A****
00000000 C          DATA SET D7PW00001 AT LEVEL 001 AS OF 06/16/81
00000010 C          DATA SET D7PW003L01 AT LEVEL 001 AS OF 01/09/79
00000020 C          MAIN PROGRAM FOR CALCULATING PERFORMANCE AND
00000030 C          PSEUDO RELATIVE PERMEABILITIES OF STRATIFIED RESERVOIR
00000040 C
00000050 C
00000060 C
00000070 C          NOTATION
00000080 C
```

Figure 12

cat of the file as received from mainframe

```

  vi d7pw
  0
  **** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
  87 B381 R13.PR1 5.1.1      TSU 3354 START  A****
  **** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
  87 B381 R13.PR1 5.1.1      TSU 3354 START  A****
  **** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
  87 B381 R13.PR1 5.1.1      TSU 3354 START  A****
  **** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
  87 B381 R13.PR1 5.1.1      TSU 3354 START  A****
  **** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
  87 B381 R13.PR1 5.1.1      TSU 3354 START  A****
  **** UNBGAXR          RODRIGUEZ ARGEMIRO          8.41.56 15-APR-
  87 B381 R13.PR1 5.1.1      TSU 3354 START  A****^M^L^M00000000 C      DA
  TA SET D7PW00001 AT LEVEL 001 AS OF 06/16/81
  00000010 C          DATA SET D7PW003L01 AT LEVEL 001 AS OF 01/09/79
  00000020 C          MAIN PROGRAM FOR CALCULATING PERFORMANCE AND
  00000030 C          PSEUDO RELATIVE PERMEABILITIES OF STRATIFIED RESERVOIR
  00000040 C
  00000050 C
  00000060 C
  00000070 C          NOTATION
  00000080 C

```

Figure 13

vi of the file after rje-printfmt

```
# SHELL TO SEND FILES TO THE MAINFRAME IBM4381
# TO BE CATALOGUED IN "UNBGAXR" USER-ID
( /comm/inputrjel )
( /comm/trans < /comm/commands )
if test -s /comm/commands
then rm /comm/commands
fi
if test -s /comm/tempi.prt
then rm /comm/tempi.prt
fi
if test -s /comm/tempo.prt
then rm /comm/tempo.prt
fi
if test -s /comm/dummy
then rm /comm/dummy
fi
if test -s /comm/output.prt
then rm /comm/output.prt
fi
```

figure 14

File /comm/rjejcl2 containing the script to run rje
for sending a file to be catalogued in the IBM4381

```

echo "Type the name of the file to be transmitted "
read b
if test ! -s $b
then echo "The file $b does not exist"
  exit 1
fi
echo "This procedure transmits a file to IBM4381 "
echo "and catalogs it in UNBGAXR"
echo "Make sure this is what you want, before proceeding."
echo "Type 'y' if you are sure ."
read a
if test "$a" = "y"
then
  cp /comm/ibmjcl6 /comm/output.prt
  cat $b >> /comm/output.prt
  echo "/*" >> /comm/output.prt
  echo "/*" >> /comm/output.prt
  cp /comm/seed /comm/commands
  echo "</comm/output.prt" >> /comm/commands
  echo "<@" >> /comm/commands
fi

```

figure 15

file /comm/inputrjcl that prompts the filename and prepares the block to be sent to the IBM.

```

//TAXR001 JOB (UNBGAXR,USERTS01),'HP 9000 TO IBM',CLASS=A
/*PASSWORD XANADU
/*JOBPARM Q=F
//STEP1 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSIN DD DUMMY
//SYSUT2 DD DSN=UNBGAXR.SATNUM.ECL,DISP=(,CATLG),
// DCB=(LRECL=80,BLKSIZE=6320,RECFM=FB),
// UNIT=SYSTS,SPACE=(TRK,(50,30),RLSE)
//SYSUT1 DD *

```

figure 16

File /comm/ibmjcl6 containing JCL leading data to be catalogued on IBM discs

IBM "*" "-n0 -w0 -s4800" /dev/rje /users/IBM TRANS 3 -1

figure 17

file /usr/rje/rje_config containing the link configuration to the host , for background mode.

```
# SCRIPT TO TRANSMIT A FILE TO THE MAINFRAME IN BACKGROUND
( /usr/rje/bin/inputrje )
rje_send /users/IBM/inputfile
if test -s /users/IBM/inputfile
then rm /users/IBM/inputfile
fi
```

figure 18

Main script to send a file in background

```

echo "          RJE PROGRAM - BACKGROUND      "
echo "Files sent are queued previously  "
echo "and released according to priorities"
echo " "
echo "Type the name of the file to be transmitted "
read b
echo " "
if test ! -s $b
  then echo "The file $b does not exist"
  exit 1
fi
echo "~! cat /usr/rje/bin/ibmjcl1" > /users/IBM/inputfile
echo "~! cat $b" >> /users/IBM/inputfile
echo "~! cat /usr/rje/bin/ibmjcl2" >> /users/IBM/inputfile

```

figure 19

File /usr/rje/bin/inputrje containing the script to send a file to the mainframe in background mode (it prompts the filename and prepares the block for transmission).

```
//TAXR001 JOB (UNBGAXR,USERTS01),'HP 9000 TO IBM',CLASS=A
/*PASSWORD XANADU
/*JOBPARM Q=F
//STEP1 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSIN DD DUMMY
//SYSUT2 DD DSN=UNBGAXR.FOLLOW,DISP=(,CATLG),
// DCB=(LRECL=80,BLKSIZE=6320,RECFM=FB),
// UNIT=SYSTS,SPACE=(TRK,(50,30),RLSE)
//SYSUT1 DD *
```

figure 20

File /usr/rje/bin/ibmjcl1 containing the JCL to send a file to be catalogued on mainframe mass storage.

/*

figure 21

File /usr/rje/bin/ibmjcl2 containing JCL to close input file being sent to the mainframe.



Portable Tape Handling for HP-UX

Stephen C. Fullerton
Statware, Inc.
P.O. Box 510881
Salt Lake City, UT 84151

July 10, 1987

1 Introduction

When you install an HP-UX system you receive a wealth of programs and utilities. For example, there are several programs for the handling of magnetic tapes: *cpio(1)*, *dd(1)*, *tar(1)*, and *tcio(1)*. However, with the exception of *dd(1)*, there are no programs for reading and writing tapes that are interchangeable with other computer systems¹. Using *dd(1)* you can read an EBCDIC tape written on an IBM machine, and you can also write an EBCDIC tape, but the task is very painful. Reading and writing blocked ASCII tapes is another matter.

This paper will discuss two programs, *tputil* and *ansitape*. Both of these programs are in the public domain; *tputil* was developed by the author and *ansitape* resides on the mod.sources public domain library and was ported to HP-UX by the author.

tputil is designed for reading and writing unlabeled tapes in EBCDIC and ASCII and *ansitape* is for reading and writing ANSI and IBM labeled tapes. Both of these programs are in the public domain and have been contributed to the CSL tape.

2 TPUTIL

TPUTIL² is an interactive program for reading and writing unlabeled fixed-blocked tapes that are easily read on other computer systems. Its features include:

- Automatic generation of headers for IBM, SPERRY, CDC, and custom headers for the VAX,
- Appending a list of disk files to a single tape file,
- Interactive tape positioning, both absolute and relative,
- Screening of files based on dates,
- Translation of special characters; e.g., international characters, zoned decimal, etc.
- Interactive examination of unknown tapes using ASCII or EBCDIC translation in ASCII, hexadecimal, octal, and decimal.

¹On some systems, the *ansitar(1)* system is made available as part of the contributed library, (*/usr/contrib*).

²Pronounced *tape util*. The name was derived from "tape utility".

- Automatic generation of a control file to be used in subsequent runs,
- Generation of a LOG file containing complete documentation of how the tape was written and what files were written.

TPUTIL was originally written in 1982 for a Sperry 1100. The majority of the source code is in Sfttran3, the structured FORTRAN preprocessor from JPL. The remainder of the code is in "C" for HP-UX, and MASM for a Sperry 1100. TPUTIL is based on a DECsystem-20 program by the same name that was written by Dr. Nelson H.F. Beebe of the College of Science at the University of Utah. The DEC-20 version of TPUTIL is written entirely in assembly code.

TPUTIL uses the tape handling capabilities of HP-UX defined in *mt(4)* in the *HP-UX Reference*. In order for TPUTIL to perform correctly, the driver for the tape drive must have a device configured with Berkeley compatibility and the nowind bit set. This is described in *mt(4)* and must be used in conjunction with *mknod(1M)*.

2.1 Sample Session

The following is a sample session using TPUTIL to write an EBCDIC tape complete with IBM headers (IEBUPDTE headers).

```
TPUTIL 2R1 UU/CC -- Statware, Inc. -- Mon Jul 13 09:26:09 1987
```

```
TPUTIL>tape /dev/mtnb
Tape /dev/mtnb is now assigned to TPUTIL and is positioned
at Load Point.
TPUTIL>lrecl 80
TPUTIL>blksiz 4000
TPUTIL>ebcdic
TPUTIL>remark Sample of an IBM tape write
TPUTIL>log sample.ibm
Now logging output on: sample.ibm
TPUTIL>header ibm
TPUTIL>info
```

IBM Job Control Language specification is:

```
DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3),
LABEL=(1,NL),
VOL=SER=TAPE
```

```
TAPE unit selected is /dev/mtnb
Record count = 0. Blocks in = 0. Blocks out = 0.
Character count = 0. Record count this file = 0.
Tape positioned at file 1.
An IBM filename heading will precede each disk file on tape.
Lowercase is NOT converted to uppercase.
Last block written may be short.
ASCII control characters ARE included in output records.
Maximum number of records to read is infinite.
Message limit = 10. Current message = 0.
```

TAPE files are recorded in EBCDIC at 1600 Bpi.
 Lines longer than 80 characters will be truncated.
 File transfers are currently being logged on sample.ibm
 Remark: Sample of an IBM tape write

TPUTIL>remark All Sftran/3 source files
 TPUTIL>append *.sf3

apccmd.sf3 [OK]	Records=434/434	Blocks=8	File=1
asccmd.sf3 [OK]	Records=57/491	Blocks=1	File=1
befcmd.sf3 [OK]	Records=358/849	Blocks=7	File=1
blkcmd.sf3 [OK]	Records=76/925	Blocks=2	File=1
blkdat.sf3 [OK]	Records=93/1018	Blocks=2	File=1
capstr.sf3 [OK]	Records=31/1049	Blocks=1	File=1
chkint.sf3 [OK]	Records=25/1074	Blocks=1	File=1
chknam.sf3 [OK]	Records=24/1098	Blocks=1	File=1
chkspc.sf3 [OK]	Records=38/1136	Blocks=1	File=1
comcmd.sf3 [OK]	Records=22/1158	Blocks=1	File=1
ctlcmd.sf3 [OK]	Records=130/1288	Blocks=2	File=1

. . . .

wrthdr.sf3 [OK]	Records=285/9105	Blocks=6	File=1
wrtlgf.sf3 [OK]	Records=53/9158	Blocks=1	File=1
wrtlgi.sf3 [OK]	Records=92/9250	Blocks=2	File=1
wrtrec.sf3 [OK]	Records=165/9415	Blocks=3	File=1
wrtsum.sf3 [OK]	Records=113/9528	Blocks=2	File=1
wrttrm.sf3 [OK]	Records=51/9579	Blocks=1	File=1
zaprec.sf3 [OK]	Records=40/9619	Blocks=1	File=1

*** EOF ***

File = 1 Records (this file) = 9619 Records (all files) = 9619

TPUTIL>remark Sftran/3 include file
 TPUTIL>write sftran.inc

sftran.inc [OK]	Records=250/250	Blocks=6	File=2
-----------------	-----------------	----------	--------

*** EOF ***

File = 2 Records (this file) = 250 Records (all files) = 9869

TPUTIL>remark All C source files
 TPUTIL>append *.c

afile.c [OK]	Records=20/20	Blocks=1	File=3
cvttim.c [OK]	Records=14/34	Blocks=1	File=3
dirget.c [OK]	Records=20/54	Blocks=1	File=3
expfil.c [OK]	Records=25/79	Blocks=1	File=3
filchk.c [OK]	Records=27/106	Blocks=1	File=3

. . . .

opndir.c [OK]	Records=8/217	Blocks=1	File=3
sysget.c [OK]	Records=10/227	Blocks=1	File=3
tapectl.c [OK]	Records=52/279	Blocks=1	File=3
trans.c [OK]	Records=16/295	Blocks=1	File=3

*** EOF ***

File = 3 Records (this file) = 295 Records (all files) = 10164

TPUTIL>remark Recent changes

TPUTIL>since saturday

TPUTIL>info

IBM Job Control Language specification is:

DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3),
LABEL=(4,NL),
VOL=SER=TAPE

TAPE unit selected is /dev/mtnb

Record count = 295. Blocks in = 1398. Blocks out = 203.

Character count = 813120. Record count this file = 16.

Tape positioned at file 4.

An IBM filename heading will precede each disk file on tape.

Lowercase is NOT converted to uppercase.

Last block written may be short.

ASCII control characters ARE included in output records.

Maximum number of records to read is infinite.

Message limit = 10. Current message = 0.

TAPE files are recorded in EBCDIC at 1600 Bpi.

Lines longer than 80 characters will be truncated.

Only disk files written SINCE Saturday, July 11, 1987 will be included in APPEND/WRITE/DIR commands.

File transfers are currently being logged on sample.ibm

Remark: Recent changes

TPUTIL>append *.sf3

appcmd.sf3 [OK]	Records=434/434	Blocks=9	File=4
dircmd.sf3 [OK]	Records=351/785	Blocks=7	File=4
getcmd.sf3 [OK]	Records=59/844	Blocks=1	File=4
getlin.sf3 [OK]	Records=175/1019	Blocks=4	File=4
main.sf3 [OK]	Records=238/1257	Blocks=5	File=4
putnew.sf3 [OK]	Records=51/1308	Blocks=1	File=4
wdcmd.sf3 [OK]	Records=83/1391	Blocks=1	File=4

*** EOF ***

File = 4 Records (this file) = 1391 Records (all files) = 11555

The SINCE date selection resulted in NOT writing 87 files to tape.

TPUTIL>no log

Now closing log file: sample.ibm

Portable Tape Handling for HP-UX

TPUTIL>write sample.ibm

sample.ibm [OK] Records=200/200 Blocks=5 File=5

*** EOF ***

File = 5 Records (this file) = 200 Records (all files) = 11755

TPUTIL>rewind

TPUTIL>dump,a 5

Block read had 4000 bytes.

Record #1 of block: ASCII

./ ADD NAME=APPCMD.SF3 13-Jul-87 08:09:11

Record #2 of block: ASCII

 SUBROUTINE APPCMD(IOPT)

Record #3 of block: ASCII

C\$-----

Record #4 of block: ASCII

C\$

Record #5 of block: ASCII

C\$ TPUTIL*TPUTIL --> UNLABELED TAPE HANDLER

*** Warning *** The tape is positioned within (or at the end of) file 1.

TPUTIL>file 3

Tape is now positioned at start of file 3.

** Warning -- Partially dumped block flushed **

TPUTIL>dcb

IBM Job Control Language specification is:

DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3),

LABEL=(3,NL),

VOL=SER=TAPE

TPUTIL>dump,a 5

Block read had 4000 bytes.

Record #1 of block: ASCII

./ ADD NAME=AFILL.C 17-Oct-86 11:37:41

Portable Tape Handling for HP-UX

Record #2 of block: ASCII

/* afill -- fill buf with n characters of c starting at position pos */

Record #3 of block: ASCII

Record #4 of block: ASCII

void afill(buf,pos,c,n)

Record #5 of block: ASCII

char *buf;

*** Warning *** The tape is positioned within (or at the end of) file 3.

TPUTIL>exit

Exit TPUTIL, your tape remains positioned at File 3.

The LOG file produced by this TPUTIL run is as follows:

TPUTIL Version 2R1 File Transaction Log

Tape: /dev/mtnb (Density 1600 bpi)

Log file: sample.ibm

Created by: scf

Date: Mon Jul 13 09:26:54 1987

Remark: Sample of an IBM tape write

=====

F I L E 1 Mon Jul 13 09:27:26 1987

DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3) (EBCDIC)

All Sfrtran/3 source files

./	ADD	NAME=APPCMD.SF3	13-Jul-87	08:09:11	434	434
./	ADD	NAME=ASCCMD.SF3	13-Oct-86	13:53:43	57	491
./	ADD	NAME=BEFCMD.SF3	10-Oct-86	14:14:12	358	849
./	ADD	NAME=BLKCMD.SF3	10-Oct-86	09:35:36	76	925
./	ADD	NAME=BLKDAT.SF3	16-Oct-86	10:06:01	93	1018
./	ADD	NAME=CAPSTR.SF3	10-Oct-86	14:16:22	31	1049
./	ADD	NAME=CHKINT.SF3	09-Oct-86	16:42:19	25	1074
./	ADD	NAME=CHKNAM.SF3	10-Oct-86	13:57:38	24	1098
./	ADD	NAME=CHKSPC.SF3	09-Oct-86	16:42:37	38	1136
./	ADD	NAME=COMCMD.SF3	10-Oct-86	13:59:02	22	1158
./	ADD	NAME=CTLCMD.SF3	14-Oct-86	17:18:58	130	1288
.
.
./	ADD	NAME=WRTHDR.SF3	05-Nov-86	15:42:47	285	9105
./	ADD	NAME=WRTLGF.SF3	10-Oct-86	14:09:42	53	9158
./	ADD	NAME=WRTLGI.SF3	15-Oct-86	15:41:49	92	9250
./	ADD	NAME=WRTREC.SF3	15-Oct-86	08:51:47	165	9415
./	ADD	NAME=WRTSUM.SF3	09-Oct-86	16:42:33	113	9528
./	ADD	NAME=WRTTRM.SF3	10-Oct-86	14:12:03	51	9579
./	ADD	NAME=ZAPREC.SF3	15-Oct-86	15:46:50	40	9619

*** EOF ***

File = 1 Records (this file) = 9619 Records (all files) = 9619

=====

F I L E 2 Mon Jul 13 09:28:47 1987

DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3) (EBCDIC)

Sftran/3 include file

./	ADD	NAME=SFTRAN.INC	12-Jul-87	16:37:36	250	250
----	-----	-----------------	-----------	----------	-----	-----

*** EOF ***

File = 2 Records (this file) = 250 Records (all files) = 9869

=====

F I L E 3 Mon Jul 13 09:29:03 1987

DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3) (EBCDIC)

All C source files

Portable Tape Handling for HP-UX

./	ADD	NAME=AFILL.C	17-Oct-86	11:37:41	20	20
./	ADD	NAME=CVTTIM.C	14-Oct-86	11:57:10	14	34
./	ADD	NAME=DIRGET.C	16-Oct-86	13:37:42	20	54
./	ADD	NAME=EXPFIL.C	14-Oct-86	14:14:29	25	79
./	ADD	NAME=FILCHK.C	14-Oct-86	12:57:00	27	106
	
	
./	ADD	NAME=OPNDIR.C	16-Oct-86	13:12:56	8	217
./	ADD	NAME=SYSGET.C	13-Oct-86	14:15:02	10	227
./	ADD	NAME=TAPECTL.C	13-Oct-86	16:05:36	52	279
./	ADD	NAME=TRANS.C	20-Oct-86	08:38:29	16	295

*** EOF ***

File = 3 Records (this file) = 295 Records (all files) = 10164

=====

F I L E 4 Mon Jul 13 09:29:37 1987

DCB=(LRECL=80,BLKSIZE=4000,RECFM=FB,DEN=3) (EBCDIC)

Recent changes

./	ADD	NAME=APPCMD.SF3	13-Jul-87	08:09:11	434	434
./	ADD	NAME=DIRCMD.SF3	13-Jul-87	09:03:45	351	785
./	ADD	NAME=GETCMD.SF3	12-Jul-87	16:19:53	59	844
./	ADD	NAME=GETLIN.SF3	12-Jul-87	16:56:59	175	1019
./	ADD	NAME=MAIN.SF3	12-Jul-87	16:36:31	238	1257
./	ADD	NAME=PUTNEW.SF3	12-Jul-87	16:56:06	51	1308
./	ADD	NAME=WDCMD.SF3	12-Jul-87	16:53:24	83	1391

*** EOF ***

File = 4 Records (this file) = 1391 Records (all files) = 11555

Log terminated on file: sample.ibm

2.2 TPUTIL Commands

The following is a complete description of the commands available in TPUTIL.

APPEND <one or more file specs>

Write a file, or group of files to the tape using the current format settings. As each file is written, the user is notified of the progress of the transfer, as well as the number of records and blocks

transferred, and the current file position. Each file will be appended to the end of the preceding file while remaining on the same tape file. The first file specified will be written at the start of the file at which the tape is currently positioned.

Multiple file specifications should be separated by spaces, and may contain wildcard characters.

Examples:

```
TPUTIL>APPEND foo.bar
TPUTIL>APPEND tputil/src/*.sf3
TPUTIL>APPEND src/*.r inc/*.h
```

ASCII

Set TPUTIL to read and write ASCII (default)

Example:

```
TPUTIL>ASCII
```

BEFORE <date and time>

In APPEND/WRITE commands, select only files which were written before the specified date and time. The BEFORE and SINCE commands can be conveniently used to automatically construct software update tapes. To do so, simply use a BEFORE and/or SINCE command at the start of the run which produces the software distribution tape. Any APPEND or WRITE commands which do not output any files at all because of the BEFORE and/or SINCE selection are simply ignored; an empty file is NOT written on the tape, and neither is a LOG file entry recorded.

Examples:

```
TPUTIL>BEFORE 01-OCT-81
TPUTIL>BEFORE TUESDAY 13:45:03
TPUTIL>BEFORE TODAY
TPUTIL>BEFORE 12-DEC-79 17:00:10
```

BLKSIZ <decimal integer in the range [18 TO 32768]>

Set the tape blocksize to be used for subsequent reads and writes. This should be an integral multiple of the desired logical record length (LRECL). If it is not, the APPEND and WRITE commands will display an error message and the tape will not be written. The minimum blocksize permitted on magnetic tapes by ANSI standards is 18 characters, and the maximum supported by IBM is 32767 characters. Records shorter than 18 characters are regarded as noise records, and are discarded by hardware or software. The default BLKSIZ is 5760; this number is a multiple of the wordsizes of all known machines, and is chosen for reasons of portability. Some machines cannot read tape blocks which do not fill an integral number of words.

Examples:

```
TPUTIL>BLKSIZ 4000
TPUTIL>BLKSIZ 80
TPUTIL>BLKSIZ 8000
```

CTL <file name>

Begin writing executed commands to the specified batch control file. Only commands that have passed all validity checks and successfully executed will be written to the CTL file. Also, a header is generated showing the userid and date and time the CTL file was written. The writing is terminated and the CTL file is closed when a NO CTL or EXIT command is issued.

Example:

```
TPUTIL>CTL temp.ctl
```

DCB

Display the DCB from the IBM JCL which corresponds to the current settings of the various parameters. This is the DCB which should be used to read a tape written here, and it should be the same as the DCB which was used to write a tape which is being read here.

Example:

```
TPUTIL>DCB
```

DEFAULTS

Reset all parameters to defaults. This can be displayed with the DCB or INFO commands. This command reinitializes TPUTIL as if it has just been entered for the first time. All open files are closed, and the tape file number is reset to 1. DEFAULTS should normally only be issued when a different tape than the initial one is to be read or written.

The default options are as follows:

ASCII	BLKSIZ 5760	DENSITY 1600
LRECL 80	MAXRECORDS 0 (infinite)	MESSAGELIMIT 10
NO BEFORE	NO FILTER	NO HEADER
NO FORMFEEDS	NO LOG	NO RAISE
NO REMARK	NO SINCE	NO STATISTICS
NO TRANSLATE	RECFM FB	SHORT
TRUNCATE		

DENSITY <800, 1600, or 6250>

Set the density to be used for subsequent reads and writes. You may use 800, 1600 or 6250 as density. TPUTIL will not allow the density to be changed after an APPEND/WRITE command. Tapes with mixed densities are unreadable on most systems (default DENSITY 1600).

Examples:

```
TPUTIL>DENSITY 800
TPUTIL>DENSITY 1600
TPUTIL>DENSITY 6250
```

DIR<,opt> <one or more file specs>

The DIR command will produce a directory listing of the specified files and/or directories. Use of the L option will cause the long form of the directory listing to be produced.

Examples:

```
TPUTIL>DIR my/src
TPUTIL>DIR,1 src/*.r inc/*.h
```

DISMOUNT

The DISMOUNT command is used to release a previously assigned tape; assigned either by the TAPE or MOUNT command. The tape is rewound and set offline.

Example:

```
TPUTIL>DISMOUNT
```

DUMP<,opt> <number of records>

DUMP records to the terminal. The DUMP command allows the user to dump records read from the tape onto the terminal in a variety of formats; i.e., ASCII, HEXIDECIMAL, OCTAL, and DECIMAL.

By default, DUMP will “dump” 5 records onto the terminal in ASCII; however, the user may specify the number of records to be “dumped”. For example,

```
TPUTIL>DUMP 15
```

requests that 15 records be dumped in ASCII.

The following options are available for the DUMP command:

- A** ASCII. Dump the records onto the terminal in ASCII.
- D** DECIMAL. Dump the records onto the terminal in decimal (the decimal ASCII ordinates).
- H** HEXIDECIMAL. Dump the records onto the terminal in HEXIDECIMAL.
- O** OCTAL. Dump the records onto the terminal in OCTAL.

These options may be used together if desired. For example,

```
TPUTIL>DUMP,AFHO
```

requests that 5 records (default) be dumped in ASCII, HEXIDECIMAL, and OCTAL.

**** Warning** – The records dumped have already been translated to or from the specified character set; e.g, ASCII or EBCDIC. To retrieve a record just as it was written onto the tape with no translation, specify ASCII as the character set by using the ASCII command.

EBCDIC

Set TPUTIL to translate to EBCDIC on output to tape and from EBCDIC on input from tape. Files which contain control characters, other than tabs, carriage returns, and line feeds may get changed in the process since not all EBCDIC characters have an equivalent ASCII character.

Example:

```
TPUTIL>EBCDIC
```

EOT

Advance to the end of tape. This command is useful if you wish to add files to the end of an existing tape. End-of-tape is signalled by two successive tapemarks, and the tape is left positioned between them, so that a subsequent APPEND or WRITE command will overwrite the second of them.

Example:

TPUTIL>EOT

EXIT

Exit from TPUTIL. Your tape, if any, will remain mounted at its current position. If a LOG file is open, it will be closed and kept.

Example:

TPUTIL>EXIT

FILE <decimal integer>

Position to the start of the specified file. Tape files are numbered 1, 2, 3, ... from load point.

Examples:

TPUTIL>FILE 4

TPUTIL>FILE 2

FILTER <single quoted character>

Replace ASCII control characters (0-31, 127+) by the specified character in records which are output to tape. Any printable character may be specified ("?" is suggested). By default, TPUTIL is in "NO FILTER".

Examples:

TPUTIL>FILTER "?"

TPUTIL>FILTER "@"

FORMFEEDS

Include ASCII formfeed characters when copying from disk to tape. These are normally deleted (if RECFM=FB) or converted to ASA carriage control characters in the first column of the following record (if RECFM=FBA).

Example:

TPUTIL>FORMFEEDS

HEADER <CDC> or <DEFAULT> or <IBM> or <UNIVAC> or <USER>

Include a header on the first line of a file or element written to tape. There are four build-in headers; however, the user may specify his/her own header by specifying USER. Then TPUTIL will prompt for the header (1-80 characters in length).

When building user headers, the header is copied exactly as the first record of each file or element written to tape. However, the user may include the following keywords in his/her header:

- @DIR Directory name
- @FILE File name
- @FILN File name without extension
- @EXT Extension
- @DATE Date the file was last written
- @TIME Time the file was last written

The four built-in headers have the form:

#HEADER NAME=file	(HEADER DEFAULT)
*DECK file	(HEADER CDC)
./ ADD NAME=file	(HEADER IBM)
@ELT,IQ file	(HEADER UNIVAC)

These headers will also include the date and time the file was last written.

Examples:

```
TPUTIL>HEADER IBM
TPUTIL>HEADER USER
HEADER>./ ADD MEMBER NAME=[@ELT,MYLIB] (@DATE,@TIME)
```

HELP <topic or [Confirm with carriage return]>

Request help on the specified topic. The HELP command scans this documentation file to find the section describing the topic and displays it on the terminal.

Examples:

```
TPUTIL>HELP APPEND
TPUTIL>HELP
```

INFO

Displays the tape specification given by the DCB command, as well as everything else TPUTIL knows about your tape, including the settings of all current options, the name of the current LOG file, and the contents of the current REMARK buffer.

Example:

```
TPUTIL>INFO
```

LOG <file name>

Begin logging disk-to-tape transfers on the specified file. Logging terminates and the file is closed when a NO LOG, DEFAULTS, or EXIT command is issued. This option is very useful for creating a compact description of what is written to a tape. For best results, it should be issued after the TAPE and DENSITY commands have been given (since their settings are recorded in the log), and before any APPEND or WRITE commands. Any REMARK text which has been supplied will be written in the LOG file header. When the last file is written, a "NO LOG" command can be followed by a "WRITE logfile" command to copy the LOG file to the last tape file, thus providing a record of what was written on the tape. The LOG file can be printed as well so that a human-readable copy can accompany the tape.

Examples:

```
TPUTIL>LOG foo.log
TPUTIL>LOG tputil.log
```

LRECL <decimal integer in the range [1,32768]>

Set the logical record length to be used in subsequent reads and writes. This may be any value between 1 and 32768 (default LRECL=80). On output to tape, records than LRECL characters are padded with blanksto exactly LRECL characters, and records longer than LRECL characters are truncated (by default), or wrapped around (if NO TRUNCATE is specified).

Examples:

```
TPUTIL>LRECL 120
TPUTIL>LRECL 40
```

MAXRECORDS <decimal integer>

For convenience in scanning through a tape, the maximum number of records to read may be specified. If zero (0) is specified, the entire tape file will be read. After a READ, the tape will be positioned at the start of the next file.

Examples:

```
TPUTIL>MAXRECORDS 10
TPUTIL>MAXRECORDS 50
```

MESSAGELIMIT <decimal integer>

Set an upper limit for the number of error messages to be printed at the terminal. The default is 10. An error count is maintained, and may be sampled at any time by using the INFO command. The error count is reset to zero (0) at the beginning of processing on each input disk file.

Examples:

```
TPUTIL>MESSAGELIMIT 5
TPUTIL>MESSAGELIMIT 0
```

MOUNT <reel name>

The MOUNT command is used to enter a tape mount request to the operator's console.

By default, the MOUNT command will assign the tape as "write-enable"d unless the "R" option is used.

Examples:

```
TPUTIL>MOUNT DCL392
TPUTIL>MOUNT,R 1234
```

NO BEFORE

Turn off any file selection set by a previous BEFORE command (default).

Example:

```
TPUTIL>NO BEFORE
```

NO CTL

Terminate the writing of executed commands to the batch control (CTL) file. This file may be edited and submitted as a batch job to write tapes exactly as done in the current session.

Example:

```
TPUTIL>NO CTL
```

NO FILTER

Turn off the FILTER option, thus preserving ASCII control characters in records output to tape.

Example:

TPUTIL>NO FILTER

NO FORMFEEDS

Do NOT include formfeed characters when writing to tape. These may be changed to ASA carriage control characters (RECFM=FBA) or deleted (RECFM=FB) (default).

Example:

TPUTIL>NO FORMFEEDS

NO HEADER

Do NOT include a file name heading on the tape (default).

Example:

TPUTIL>NO HEADER

NO LOG

Turn off any file transfer logging set by a previous LOG command, and close the log file. The log file may then be written to the tape in the same invocation of TPUTIL, providing a record of the tape's contents.

Example:

TPUTIL>NO LOG

NO RAISE

Set TPUTIL so that lower case is not converted to upper case on output. Lower case is NEVER converted to upper case on input.

Example:

TPUTIL>NO RAISE

NO REMARK

Clear any text set by previous REMARK commands.

Example:

TPUTIL>NO REMARK

NO SHORT

Do NOT write short blocks at the end of the tape file, but instead output padding records containing NULs (binary zeros). This option is NOT recommended for IBM use, and should only be necessary for those few primitive operating systems which raise an error condition if a short block is read at the end of a file. TPUTIL will NEVER write a short block in the middle of a file.

Example:

TPUTIL>NO SHORT

NO SINCE

Turn off any file selection set by a previous SINCE command.

Example:

```
TPUTIL>NO SINCE
```

NO STATISTICS

Suppress the statistics message normally printed after each file is read or written (default).

Example:

```
TPUTIL>NO STATISTICS
```

NO TRANSLATE

Return the translate tables to their default values, regardless of how many TRANSLATE directives were processed.

Example:

```
TPUTIL>NO TRANSLATE
```

NO TRUNCATE Do not truncate records which exceed LRECL characters in length. Instead, continue them on succeeding lines. A message will be displayed informing the user of how many records were wrapped.

Example:

```
TPUTIL>NO TRUNCATE
```

QUIT

Synonym for EXIT.

RAISE

Set TPUTIL to convert lower case to upper case on output to tape. Lower case is NEVER converted to upper case when copying from tape to disk. Since case conversion is an irreversible operation which can be exceedingly difficult to undo correctly with a text editor, this option should be used with extreme caution. It should only be necessary if the machine on which the tape is to be read does not support lower-case letters (e.g. CDC).

Example:

```
TPUTIL>RAISE
```

READ<,opts> <one or more output disk files>

Read the current file from the tape, using the current format settings, into the designated file or element. The tape is left positioned after the end-of-file mark so that a subsequent READ will access the next tape file. If MAXRECORDS is greater than zero (0), then only the first MAXREC records of the tape file will be read. If you subsequently decide to copy a particular file to disk, remember to issue a MAXRECORDS 0 command BEFORE the READ command.

The READ command always positions the tape at the beginning of a tape file before and after reading unless the "N" option is used.

The READ command also has a U option available for writing without any header information. This option should be used with care and is probably only useful to systems programmers.

Examples:

```
TPUTIL>READ myfile
TPUTIL>READ file1 file2 file3
```

RECFM <FB, FBA, or U>

Specify the tape record format to be one of:

FB fixed-length blocked records.

FBA fixed-length blocked records containing an ASA carriage control character in the first column of each record.

U Undefined block length, each tape block will be output as a single disk record (for tape input ONLY).

For an APPEND or WRITE command, if the file already contains ASA carriage control characters in the first column of each record, you should specify RECFM FB (not FBA) for TPUTIL, but when the tape is read at an IBM installation, you should then specify RECFM=FBA. Specification of RECFM FBA for output to tape should be done when the disk file is a print file; the print control images for overprinting, single spacing, double spacing, and page ejects will be converted to standard carriage control characters in the first column of each output record.

Examples:

```
TPUTIL>RECFM FB
TPUTIL>RECFM FBA
```

REMARK <1-80 character text string>

Provide a text line which is to be recorded in a log file opened by the LOG command. Each REMARK command provides one text line, and repeating REMARK commands overwrites the previous one. A non-empty REMARK text string is written to the LOG file when the LOG file is opened, and when each new tape file is opened for output. The NO REMARK command clears the text buffer. Thus, a REMARK command which precedes a LOG command can be used to provide global information which is written in the LOG file header. Then as each disk file is written to tape, the APPEND/WRITE command can be preceded by a REMARK command to enter file-specific information into the LOG file. The INFO command will display the current contents of the REMARK string.

Examples:

```
TPUTIL>REMARK This a test of the REMARK command.
TPUTIL>REMARK SFTRAN3 files SINCE 01-Oct-82
```

REWIND

Rewind the tape to load point (beginning of the first file).

SHORT

The last block written may be short if there are not enough records to fill it. This is the normal way IBM writes tapes (default).

Example:

```
TPUTIL>SHORT
```

SINCE <date and time>

In APPEND/WRITE commands, select only files which were written since the specified date and time. The BEFORE and SINCE commands can be conveniently used to automatically construct software update tapes. To do so, simply use a BEFORE and/or SINCE command at the start of the run which produces the software distribution tape. Any APPEND or WRITE commands which do not output any files at all because of the BEFORE and/or SINCE selection are simply ignored; an empty file is NOT written on the tape, and neither is a LOG file entry recorded.

Examples:

```
TPUTIL>SINCE 01-OCT-81
TPUTIL>SINCE TUESDAY 13:45:03
TPUTIL>SINCE TODAY
TPUTIL>SINCE 12-DEC-79 17:00:10
```

SKIP<.opt> <decimal integer>

Skip the specified number of files forward. A negative number skips backwards. SKIP 0 is a no-op. If the "B" option is used, then TPUTIL will skip blocks rather than files. If a file mark is encountered when skipping blocks, TPUTIL will not continue skipping blocks and will display a message stating the current tape position.

Examples:

```
TPUTIL>SKIP 4
TPUTIL>SKIP -3
TPUTIL>SKIP,B 3
TPUTIL>SKIP,B -10
```

STATISTICS

After each file is read or written, write a message giving the number of characters, records, and blocks transferred.

Example:

```
TPUTIL>STATISTICS
```

STOP

Synonym for EXIT.

TAKE <file spec>

The TAKE command directs TPUTIL to read the input commands from the specified file, rather than from the terminal. The TAKE command may be nested up to 5 levels.

Example:

```
TPUTIL>TAKE my.cmd
```

TAPE <device name>

Select a tape unit. The tape must have already been mounted before executing TPUTIL.

Example:

```
TPUTIL>tape /dev/mtnb      ! tape on berkeley format servo
```

**** Warning – TPUTIL will ONLY read and write 9-track tapes.**

TRANSLATE <from char set> <from ordinate> <to char set> <to ordinate>

Dynamically alter TPUTIL's internal character translate tables. When translating from ASCII-to-ASCII, ASCII-to-EBCDIC, etc, TPUTIL uses internal tables to perform the character conversions. However, not all characters from one character-set have a representation in another. For example, EBCDIC is a 8-bit character set and ASCII is 7-bit character set; therefore, EBCDIC has twice as many characters possible as ASCII. If a character doesn't have a representation in the translated character set, TPUTIL uses a blank instead.

Even though ASCII is 7-bit and has characters with decimal ordinates from 0-127, and EBCDIC is 8-bit and has decimal ordinates from 0-255, TPUTIL allocates a table of length 256 (0-255). Hence most characters translate into blanks.

The TRANSLATE command enables the user to change one or more of TPUTIL's translate tables. For example,

```
TPUTIL>TRANSLATE EBCDIC 189 ASCII 80
```

tells TPUTIL to alter the EBCDIC-to-ASCII translate table so that the character corresponding to EBCDIC decimal ordinate 189 ("BD" in hexadecimal) to the character corresponding to ASCII decimal ordinate 80 (uppercase "P").

TPUTIL allows the following character set translations:

```
ASCII-to-ASCII (to tape)
ASCII-to-EBCDIC (to tape)
EBCDIC-to-ASCII (from tape)
```

Example:

```
TPUTIL>TRANSLATE ASCII 63 ASCII 32
```

TRUNCATE

Truncate records longer than LRECL characters in length. For every record truncated, a warning message will be issued giving its position in the file (default). NO TRUNCATE may be specified to cause long lines to wrap around into multiple records.

Example:

```
TPUTIL>TRUNCATE
```

WD [<directory name>]

Set or display the current working directory. TPUTIL will set the specified directory name as the current working directory. If no directory name is specified, then TPUTIL will display the name of the current working directory.

Example:

```
TPUTIL>WD /dave/dat
TPUTIL>WD
```

WRITE <one or more file specs>

Write a file, or group of files, to the tape using the current format settings. As each file is written, the user is notified of the progress of the transfer as well as the number of records and blocks transferred and the current file position. Each input file will be written on a new tape file.

Multiple file specifications should be separated by spaces, and may contain wildcard characters.

For transferring software to other machines, it will usually be more convenient to use the HEADER option together with the APPEND command, rather than the WRITE command. Processing multi-file tapes on some machines (particularly IBM) can be EXCEEDINGLY tedious for the user.

Examples:

```
TPUTIL>WRITE foo.bar
TPUTIL>WRITE tputil/sperry
```

3 ANSITAPE

ANSITAPE reads, writes, and creates magnetic tapes conforming to the ANSI standard for magnetic tape labeling. Both text and binary files can be read and written; TPUTIL cannot be used for binary files.

A primary use of ANSITAPE is for reading tapes written with VAX/VMS, which makes labeled tapes by default.

ANSITAPE can also read and write IBM standard labeled tapes.

3.1 Sample Sessions

The following example demonstrates how to read an ANSI tape written by a VAX/VMS system. First, a list of files on the tape are displayed.

```
$ ansitape tv mt=/dev/mtnb
volume s80vax
distrib.com          1 blocks, rec fmt var max 86 bytes, cc implied
stat80.bck          515 blocks, rec fmt fixed 8192 bytes, cc implied
maks80.bck          225 blocks, rec fmt fixed 8192 bytes, cc implied
```

To extract the file, distrib.com, enter:

```
$ ansitape xv mt=/dev/mtnb distrib.com
volume s80vax
x distrib.com          1 blocks
```

This example writes a list of files to tape with volume name, "sample".

```
$ ansitape cv mt=/dev/mtnb vo=sample rf=v *.sf3
volume sample initialized
r appcmd.sf3          8 blocks
```



```

r asccmd.sf3          1 blocks
r befcmd.sf3          7 blocks
r blkcmd.sf3          2 blocks
r blkdat.sf3          3 blocks
r capstr.sf3          1 blocks
r chkint.sf3          1 blocks
r chknam.sf3          1 blocks
r chkspc.sf3          1 blocks
r comcmd.sf3          1 blocks
r ctlcmd.sf3          3 blocks
.
.
.
r wrthdr.sf3          5 blocks
r wrtlgf.sf3          1 blocks
r wrtlgi.sf3          2 blocks
r wrtrec.sf3          4 blocks
r wrtsum.sf3          2 blocks
r wrttrm.sf3          1 blocks
r zaprec.sf3          1 blocks

```

Finally, display a directory of the tape just written:

```

$ ansitape tv mt=/dev/mtnb
volume sample mounted
appcmd.sf3            8 blocks, rec fmt var max 76 bytes, cc implied
asccmd.sf3            1 blocks, rec fmt var max 76 bytes, cc implied
befcmd.sf3            7 blocks, rec fmt var max 76 bytes, cc implied
blkcmd.sf3            2 blocks, rec fmt var max 76 bytes, cc implied
blkdat.sf3            3 blocks, rec fmt var max 77 bytes, cc implied
capstr.sf3            1 blocks, rec fmt var max 78 bytes, cc implied
chkint.sf3            1 blocks, rec fmt var max 78 bytes, cc implied
chknam.sf3            1 blocks, rec fmt var max 76 bytes, cc implied
chkspc.sf3            1 blocks, rec fmt var max 76 bytes, cc implied
comcmd.sf3            1 blocks, rec fmt var max 76 bytes, cc implied
ctlcmd.sf3            3 blocks, rec fmt var max 76 bytes, cc implied
.
.
.
.
.
.
wrthdr.sf3            5 blocks, rec fmt var max 76 bytes, cc implied
wrtlgf.sf3            1 blocks, rec fmt var max 78 bytes, cc implied
wrtlgi.sf3            2 blocks, rec fmt var max 78 bytes, cc implied
wrtrec.sf3            4 blocks, rec fmt var max 76 bytes, cc implied
wrtsum.sf3            2 blocks, rec fmt var max 78 bytes, cc implied
wrttrm.sf3            1 blocks, rec fmt var max 78 bytes, cc implied
zaprec.sf3            1 blocks, rec fmt var max 76 bytes, cc implied

```

3.2 ANSITAPE Program

The command form of the ANSITAPE program is:

```
ansitape txrc[vqfaei3] [mt=device] [vo=volume-name] [rs=[ r | recordsize ]] [bs=blocksize]
[rf=[ v | f ]] [cc=[ i | f | e ]] filename1 filename2 ...
```

ANSITAPE reads, writes, and creates magtapes conforming to the ANSI standard for magtape labelling. Primarily, this is useful to exchange tapes with VAX/VMS, which makes this kind of tape by default.

ANSITAPE is controlled by a function key letter “(t, x, c, or r)”. Various options modify the format of the output tape.

3.2.1 Writing ANSI Tapes

The list of files on the command line is written to the tape. A full Unix pathname may be specified, however, only the last pathname component (everything after the last /) is used as the filename on the tape.

Normally, regular text files are to be exchanged. ANSITAPE reads the files one line at a time and transfers them to the tape. The newline character at the end of each line is removed, and the file is written in a variable-length record format. Variable-format files have the length of the longest record specified in a file header. Therefore, ANSITAPE will read each input file from disk before it goes on to tape, to determine the maximum record size. The read is skipped if the file is more than 100,000 bytes long. The default carriage control (implied) instructs the other host to restore the newline character before printing the record.

If ANSITAPE thinks that the input file is a Unix text file (Fortran or implied carriage control), it will automatically strip the the Unix newline from the end of each record. No strip is done with embedded carriage control files, or with any file using a fixed-length record format.

For binary files, fixed-length records should be used. VAX/VMS normally uses a record length of 512 bytes for things like directories and executable files, but data files may have any record length. Binary files should be flagged for embedded (rf=e) carriage control.

3.2.2 Reading ANSI Tapes

When reading, the input file list is presumed to be the names of files to be extracted from the tape. The shell wildcard characters asterisk (*) and question-mark (?) may be used. Of course, they must be quoted to prevent the shell from interpreting them before ANSITAPE sees them.

None of the options for record format or carriage control need be specified when reading files. ANSITAPE will automatically pick up this information from the header records on the tape, and do the right thing. If you can't get just what you want from ANSITAPE, the resulting files may be run through *dd(1)*.

3.2.3 Function Letters

These function letters describe the overall operation desired. One of them must be specified in the first argument to ANSITAPE. For lexically rigorous Unix fans, a minus sign (-) is allowed, but optional, to introduce the first keyword option set.

r Write the named files on the end of the tape. This requires that the tape have been previously initialized with an ANSI volume header.

- c Create a new magtape. The tape is initialized with a new ANSI volume header. All files previously on the tape are destroyed. This option implies **r**.
- x Extract all files from the tape. Files are placed in the current directory. Protection is **r/w** to everyone, modified by the current *umask*(2).
- t List all of the names on the tape.

3.2.4 Modifier Key Letters

These key letters are part of the first argument to ANSITAPE.

- v Normally ANSITAPE does its work silently; the **v** (verbose) option displays the name of each file ANSITAPE treats, preceded by the function letter. It also displays the volume name of each tape as it is mounted. When used with the **t** option, ANSITAPE displays the number of tape blocks used by each file, the record format, and the carriage control option.
- q Query before writing anything. On write (**c** or **r** options), this causes ANSITAPE to ask before writing to the tape. On extract operations, ANSITAPE displays the Unix pathname, and asks if it should extract the file. Any response starting with a "y" or "Y" means yes, any other response (including an empty line) means no.
- f File I/O is done to standard **i/o** instead. For example, when writing a tape file that is to contain a lint listing, we could specify

```
lint xyz.c | ansitape rf xyz.lint
```

instead of

```
lint xyz.c > /tmp/xyz.lint
ansitape r /tmp/xyz.lint
rm /tmp/xyz.lint
```



When reading, this option causes the extracted files to be sent to **stdout** instead of a disk file.

- a The tape should be read or written with the ASCII character set. This is the default.
- e The tape should be written with the EBCDIC character set. The mapping is the same one used by the *dd*(1) program with *conv=ebcdic*. This option is automatically enabled if IBM-format labels are selected.
- i Use IBM-format tape labels. The IBM format is very similar, but not identical, to the ANSI standard. The major difference is that the tape will contain no HDR3 or HDR4 records, thus restricting the name of the files on the tape to 17 characters. This option automatically selects the EBCDIC character set for output. To make an IBM-format label on a tape using the ASCII character set (why?), use the option sequence **ia**.
- 3 Do not write HDR3 or HDR4 labels. The HDR3 label is reserved for the use of the operating system that created the file. HDR4 is for overflow of filenames that are longer than the 17 characters allocated in the HDR1 label. Not all systems process these labels correctly, or even ignore them correctly. This switch suppresses the HDR3 and HDR4 labels when the tape is to be transferred to a system that would choke on them.

3.2.5 Function Modifiers

Each of these options should be given as a separate argument to ANSITAPE. Multiple options may be specified. They must appear as after the key-letter options above, and before any filename arguments.

mt=device

Select an alternate drive on which the tape is mounted. The default is */dev/rmt8*.

vo=volume-name

Specify the name of the output volume. Normally, this defaults to the first six characters of your login name. The string "UNIX" is used as the default if ANSITAPE cannot determine your login name.

rs=recordsize

Specify the output recordsize in bytes. This is the maximum size in the case of variable-format files. This option also turns on the fixed-record-format option. Thus, if you want to have variable record sizes with a smaller maximum, you must specify, **rs=recordsize rf=v**.

When the recordsize is manually given, ANSITAPE does not read disk files to determine the maximum record length.

rs=r

This is a variant of the **rs=** option. This causes ANSITAPE to read all disk files for recordsize, regardless of their size. Normally, files larger than 100K bytes are not scanned for recordsize. Using this option also implies variable-length records.

bs=blocksize

Specify the output blocksize, in bytes. As many records as will fit are crammed into each physical tape block. ANSI standards limit this to 2048 bytes (the default), but you may specify more or less. Be advised that specifying more may prevent some systems from reading the tape.

rf=v

Record format is variable-length. In other words, they are text files. This is the default, and should be left alone unless you really know what you're doing.

rf=f

Record format is fixed-length. This is usually a bad choice, and should be reserved for binary files. This also turns off the newline strip usually done for Unix text files.

cc=i

Carriage control implied (default). Unlike Unix text files, where records are delimited by a newline character, ANSI files do not normally include the newline as part of the record. Instead, a newline is automatically added to the record whenever it is sent to a printing device.

cc=f

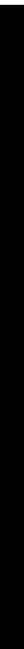
Carriage control Fortran. Each line is expected to start with a Fortran carriage-control character. ANSITAPE does not insert these characters automatically, it merely marks the file as having them. This is of limited usefulness. (Good opportunity for another ambitious hacker.)

cc=e

Carriage control is embedded. Carriage control characters (if any) are a part of the data records. This is usually used in the case of binary data files.

4 Summary

Even though the vanilla HP-UX doesn't provide the necessary tools for reading and writing labeled and unlabeled tapes in a portable format, there are public domain programs that do it quite well. There is a wealth of public domain software available for UNIX machines, you just have to look for it.



**COMPUTER-AIDED TEST SOFTWARE:
DO YOU WANT TO DO A BETTER JOB?**

Terie Robinson and Dave McDorman
Hewlett-Packard Co.
3003 Scott Blvd.
Santa Clara, CA 95054

ABSTRACT: Automation isn't easy: You have to learn all about computers, write lots of code, and the program specs keep changing, along with the test equipment and what's to be tested. There are ways to develop computer-aided test (CAT) software that significantly reduce these frustrations. These ways involve a new approach to software development. This paper makes an introduction and shows how you can bring the process under control and develop more satisfactory CAT software in less time and with fewer headaches.

INTRODUCTION

Computer-aided test (CAT) is becoming quite popular for two reasons: It doesn't cost nearly so much as it once did; foreign competition is automating with a vengeance. Dropping hardware costs have had the biggest impact on overall costs for CAT systems. The availability of easier to use software environments also has encouraged people to try their hand at becoming programmers. Many find rather quickly that it's rather easy to generate a lot of code. Unfortunately, it's much more difficult to generate reliable and supportable code.

A group in Hewlett-Packard decided to tackle the problem of creating a methodology for the development of high-quality CAT software and a related standardized architecture to speed

Computer-Aided Test Software:
Do You Want To Do A Better Job?

implementation time. Since the commercial computing marketplace had been experiencing these problems for quite a while, the group investigated there first.

The group found that many of the problem-solving techniques and their associated documentation tools were useful. The major differences between commercial applications and the CAT software HP was concerned with were project size and treatment of device I/O: Commercial systems tend to very large scale projects emphasizing data bases manipulation and some limited human interaction; CAT software projects, by comparison, seem rather small and do a tremendous amount of interrupt-driven device I/O with highly interactive human interfaces.

By leveraging the problem-solving approach and documentation tools, the result was a much-simplified software development methodology based heavily on something known as structured techniques, plus the development of a software architecture for CAT applications running on the HP 9000 Series 200/300 BASIC Language Workstation.

The group passed on the methodology and architecture tools to others in Hewlett-Packard through internal training. This training later became the following two courses:

- * HP 05096A Structured Analysis and Structured Design for Test and Measurement Systems;
- * HP 51473A Software Architecture for Test and Measurement Systems.

This paper is devoted to an overview of the methodology and architecture and why it is useful. It begins with a look at how software may be thought of as having a "life cycle." From there, the different phases of the life cycle will be examined, followed by a discussion of the tools used in each phase. Finally, the standardized architecture will be reviewed.

THE LIFE CYCLE APPROACH

Many researchers in the area of structured techniques have discovered that software development tends to follow some well-defined stages. These stages include the gathering of

Computer-Aided Test Software:
Do You Want To Do A Better Job?

technical requirements from the potential users, figuring out how the software ought to be put together, writing the code, and testing it. (See Figure 1 at the end of the paper for a diagram of the idealized software life cycle.)

Most of us tend to do all of these tasks simultaneously. The result is often a piece of software never quite done because it never quite works.

By making the tasks distinct, there is more of a chance that the work assigned to each phase will be completed before going on to the next phase. This limits the problem's complexity by forcing a step-by-step approach to software development. The benefits include:

- * Saving time (and money) over the long run;
- * Properly setting expectations (yours and the user's);
- * Sharing information between the project members;
- * Implementing projects consistently;
- * Facilitating document standardization;
- * Creating software which may be easily updated and/or reused.

Examination of the individual life cycle phases should illustrate how these benefits may be realized.

THE MAJOR PHASES

THE SURVEY PHASE

The survey phase deals with project feasibility. The questions to answer at this time are:

- * In general, what is the software to do?
- * Is it realistic to attempt the project?

Computer-Aided Test Software:
Do You Want To Do A Better Job?

- * What kind of resources would be required to do the project?

This phase allows the software developer to assess risks. These risks are determined by gathering general technical requirements from the potential user, then answering the questions above.

Structured analysis tools are used to create a general on-paper model of the potential software system. The model serves as documentation of the high-level analysis that is done at this stage and as a communication tool with the user.

If it seems like a good project, a proposal is created for the user to approve.

THE ANALYSIS PHASE

The analysis phase deals with nailing down the nitty-gritty technical requirements of the project. That means performing the following tasks:

- * Gathering *detailed* technical requirements from the user;
- * Developing the functional specification;
- * Developing the acceptance test specification;
- * Getting both documents approved.

Notice that each of the steps is aimed at defining just what the software is to do, that is specifying its functionality.

The resulting document is the **functional specification**. It will be the source of technical information in the design phase and will act as a golden standard against which the finished product will be judged.

Structured analysis tools are used to create the functional specification. As with the proposal, the tools help to model the software functionality on paper.

The **acceptance test specification** is a document based on the functional specification. It defines the process the developer will use to prove to the user that the software meets all requirements. The acceptance test demonstrates proper

Computer-Aided Test Software:
Do You Want To Do A Better Job?

functionality and is not a substitute for the test plan that should be used during code development.

Both documents must be approved by the user. This has significant side-effects:

- * No further functionality may be added to the software after the analysis phase.
- * The software is considered finished as soon as it meets the acceptance test specification.

These benefits help to correct one of the worst problems seen in many a programs, the never-ending project.

THE DESIGN PHASE

A separate design phase is required because of the exceptionally high percentage of bugs in software designs compared to those found in code. TRW, for instance, found in one project that 64% of the 220 errors discovered during or after acceptance testing were due to design flaws.¹

The tools used at this stage use the functional specification as input to create a structured model of the software to be written. The model, known as the design, will consist of a hierarchy of modules. Its purpose is to spell out the module definitions and their interfaces to each other. Once created, it can be improved according to a set of standard design rules.

After the software design is done, a test plan is written. It outlines the order in which the modules are to be implemented (coded), and how they will be tested.

THE IMPLEMENTATION PHASE

Because of all the preparatory work, the implementation phase becomes very straight-forward: Code the modules laid out in the design, in the order defined by the test plan. Testing and

¹ James Martin, Carma McClure. *Structured Techniques For Computing* (Englewood Cliffs, NJ: Prentiss-Hall, Inc., 1985), pg. 402.

integrating the modules into the rest of the software is also done according to the test plan.

When the software is completed, the acceptance test plan (from the analysis phase) is used to show that it meets the functional specifications.

To finish the project, the documentation is completed. This documentation may include a user's manual. The documentation, listings, and backups, along with the code, are part of the completed project.

OTHER

Other phases may be added to the life cycle. An OEM or software supplier, for example, may add a product evaluation phase after the implementation phase that calls for extensive alpha and beta site testing.

STRUCTURED TECHNIQUES

In software development, structured techniques are pretty much what they sound like--a methodical approach to software problem analysis, design, and development. The following quote gives a more formal definition:

Structured techniques are a collection of techniques, methodologies, and tools whose common objective is to build high-quality, low-cost software systems. They include programming methodologies for analysis, design, coding, and testing, project management concepts, and documentation tools.²

The structured techniques used in this methodology utilize structured analysis during the survey and analysis phases of

² Ibid., pg. 42.

the life cycle, structured design in the design phase, and structured programming in the implementation phases.

Where possible, graphical tools serve as the backbone of the structured techniques. The system structure is shown graphically, while algorithms and data definitions are documented with text. This keeps the wordy portion of documents to a minimum.

A closer look at the techniques and associated tools will clarify.

STRUCTURED ANALYSIS

The goal of structured analysis is to reduce a problem's complexity into its simpler parts. Decomposition of the problem starts with viewing the system in a high-level manner to make sure that all of the system interaction with the outside world is taken into account. The system is then broken down into logical, or functional, parts level by level, until each level is simply defined. Graphical tools used to document this process are data flow diagrams and control flow diagrams. Textual tools supporting the DFDs and CFDs are mini specs and a data dictionary.

TOOLS OF THE STRUCTURED ANALYSIS PHASE

DATA FLOW DIAGRAM (DFD)

Data flow diagrams consist mainly of bubbles and curved-line arrows. The bubbles represent processes (or tasks) that operate on data, hence the name "data flow diagram." The arrows are used to show data movement, or flows, from one process to another. (See Figure 3.)

The context diagram is a special case DFD which shows the proposed software's interaction with the outside world. It consists of one bubble with data flows to rectangles that symbolize physical sources and sinks of data. For example, a test system would source measurement data for a CAT program. (See Figure 2 for an example.)

Computer-Aided Test Software:
Do You Want To Do A Better Job?

The advantage of the DFD is that it forces the analysis to concentrate on data, and its transformation into information, without worrying about control and sequencing.

The diagrams tend to be intuitively understood, so that they can be used as a communication tool with those people who may not be entirely familiar with their purpose. This makes them ideal for use in project documents, so the proposal from the survey phase includes a context diagram and first level DFD while the functional specification of the analysis phase contains a levelled set of data flow diagrams.

CONTROL FLOW DIAGRAM (CFD)

Control flow diagrams emphasize the step-by-step flow of control between processes previously defined by the DFDs.

Although somewhat like flowcharts, CFDs don't use such complex symbology. They consist primarily of labelled rectangles and arrows. Each rectangle represents a process. Each arrow represents a transition between processes. If the transition is conditional, a notation is made on the arrow. (See Figure 4 for a sample.)

The advantage of control flow diagrams is their emphasis on process sequences. No data shows up on a CFD. In a formal document, where it is necessary to show both, such as a functional specification, a given DFD would be located on opposing pages with its corresponding CFD.

Another advantage of CFDs is that they are better suited to showing an interrupt-driven system than industry-standard DFDs and flow-charts.

MINI-SPECS (PROCESS SPECIFICATIONS)

All the diagrams need to be backed up by some kind of explanation of what the processes are to do. This is the function of the mini-spec, or process specification. (See Figure 5 for an example.)

Mini-specs are written only for the most primitive processes (those not further broken down on another diagram). This way, if process definitions change, only a limited amount of text requires modification.

Computer-Aided Test Software:
Do You Want To Do A Better Job?

Simple mini-specs are included in the project proposal to support the DFD. Detailed mini-specs are used in the functional specification.

DATA DICTIONARY (DD)

The data dictionary is the one place where all data paths and miscellaneous terms are described. It serves as a backup document to all the other documents. Like the mini-spec, it keeps the definition text to a manageable size, all in one place.

Commonly used data dictionary symbols are below:

=	means	"consists of"
+	means	"and"
	means	"or"
{ <i>item</i> }	means	"iterations of <i>item</i> "
[<i>item</i>]	means	"optionally <i>item</i> "

One data dictionary entry for the bandpass filter example used in the figures might look like this:

FILTER DATA = {Gain + Measurement Frequency} for the filter under test.

meaning that FILTER DATA consists of iterations of Gain and Frequency pairs for the filter under test.

A data dictionary can be found in both the project proposal and the functional specification.

STRUCTURED DESIGN

Structured design picks up where the analysis left off. It is the disciplined process of converting the functional specification into a blueprint for the software.

Computer-Aided Test Software:
Do You Want To Do A Better Job?

TOOLS OF THE STRUCTURED DESIGN PHASE

The structured design process uses a combination of tools consisting of the structure chart (graphical tool), and the supporting module specifications and design data dictionary (text).

THE STRUCTURE CHART

A structure chart looks much like a company organization chart. It consists of labelled rectangles and connecting arrows. Unlike the organization chart, though, it shows the communication paths, or the interfaces, between upper and lower levels.

Each rectangle represents an independent software module. The arrows represent the interfaces between modules and may include data (shown as smaller accompanying labelled arrows). The data flows probably will be turned into pass parameters at implementation time. (See Figure 6 for a sample structure chart.)

The advantage of a structure chart is that, right away, you can see which module depends on what others to perform its task and what data it requires as input and what it produces as output.

THE DATA DICTIONARY

The structure chart does not say enough about the composition of the data flows between modules. The design data dictionary rigorously defines these and any other terms used in the design documentation that need special clarification.

The differences between the design data dictionary and the data dictionary from the analysis phase come from the difference in emphasis between the two phases. The design phase is very close to code which means that data paths must acquire specific storage types. (For example, should a data path called Reading be integer, real or complex? What units will it be in? What range should it be in?). Also, new data items may be needed to improve the design, so the design data dictionary requires more entries than the data dictionary in the functional specification. (See the section below on improving the design.)

THE MODULE SPECIFICATION

Computer-Aided Test Software:
Do You Want To Do A Better Job?

For every rectangle on the structure chart, there is a supporting module specification written to explain how that module should work. Module specifications are as close as it gets to code in the design phase and are written in pseudocode, or structured language (similar to pseudocode).

TECHNIQUES FOR DESIGN IMPROVEMENT

Since structure charts show the relationships between modules in a graphical form, they can be evaluated fairly easily. Good designs have certain characteristics, including a large number of very cohesive modules with high fan-in and low coupling. A quick look at these concepts will explain why they're important.

Fan-in. It is highly desirable for modules to be reusable, since it can be tiresome and counterproductive to continually re-invent working code. Reusability can be indicated by how many other modules depend on it (use it). On the structure chart high fan-in appears as a number of arrows converging from above like a fan.

Coupling. High fan-in is often the result of a well-planned interface. This means that the data flowing in and out is generalized as much as possible. Also, it's an indication that other modules have little need to know how the module works. It might be said that the module is highly independent of (loosely coupled with) the rest of the design. In the jargon of structured design, this is known as low coupling--a desirable trait.

One author explains that

Coupling is a measure of strength of the connections between various modules of a program. The greater the amount of coupling between modules, the higher the probability that modifying one will have an effect on another. On the other hand, the looser the coupling between modules, the more independent the modules become. Thus a low degree of coupling is desirable.³

³ Greg DeWilde, "Developing Modular Programs," *Computer Language*, vol. 4, no. 1 (January 1987): 52.

There are many types of coupling that appear in software. Among these are data, stamp, control, and common. The most desirable is data coupling, since that is just the data that the module needs to get its work done. The least desirable is content coupling. An example of this in BASIC is the subroutine that sometimes is executed from its beginning line and other times from some line in the middle. Debugging such code can be a nightmare.

Cohesion. The author quoted above defines cohesion as a quality measure of the "relationship between different processing elements in a module."⁴ In other words, it is a measure of how well a module "hangs together." Types include coincidental cohesion, logical cohesion, temporal cohesion, procedural cohesion, and sequential cohesion.

The worst type of cohesion is coincidental cohesion, where all the parts of a module appear to have been thrown together in a completely random fashion. The best type of cohesion is functional cohesion. If a module is functionally cohesive, then every part of that module works toward accomplishing a single task.

Design improvement. The process of improving the design starts with a critical evaluation of the structure chart. This evaluation should point out areas that could be improved, according to the concepts discussed above. The chart is then re-arranged for a better design, while making sure that it still fulfills the functional specification.

It should be noted that this process is not often rote and requires a certain degree of creativity. It can also take a great deal of time. Project managers should set reasonable time limits to the design phase to ensure that enough time will be left for the implementation phase.

STRUCTURED PROGRAMMING

One of the first of the structured techniques to be invented was structured programming. It consists of writing code in a modular fashion and implies that it was designed through a process known as step-wise refinement (another name for functional decomposition).

⁴ Ibid., pg. 55.

Code is written taking advantage of whatever structured constructs are available in the language. HP 9000 Series 200/300 BASIC, for example, has LOOP/EXIT IF/END LOOP which can be used to avoid using GOTO statements to jump out of loops before normal termination.

Structured programming also implies that the modules were written and put together in a certain order. This order might be starting at the top of a structure chart and gradually working down towards the lowest layers (top-down), or it might be starting at the bottom and working up towards the top (bottom-up). Either way, each module is coded and tested independently. It is then added to the growing program (integrated). The advantage to coding in this manner is that you can always trust that each module has been fully tested before integration.

STRUCTURED WALKTHROUGHS

Walkthroughs are similar to engineering design reviews. A walkthrough consists of going over documents methodically to try to spot errors. They are done with others present to gain fresh input into the problem-solving process. A structured walkthrough is the formalized version of this same process.

Walkthroughs may be done at any stage in the life cycle and with any document.

Note that finding errors is the goal of a walkthrough, not finding fixes for them. Problems to be watched for include omissions, extraneous details, overcommitment, undercommitment, and logical mistakes. This kind of help can save a lot of time later on in the process.

AUTOMATED TOOLS

By now, you may be wondering whether or not all of these tools aren't too much to deal with manually. If you intend to use implement structured analysis and design in full, and you are working on a large project, the documentation work would be overwhelming. There are some software packages available to support the structured techniques. One example is a software package distributed by Hewlett-Packard, known as HP Teamwork.

**Computer-Aided Test Software:
Do You Want To Do A Better Job?**

HP Teamwork has a number of modules. HP Teamwork/SA is the core product and provides the structured analysis tools. HP Teamwork/SD is an optional module for structured design work. HP Teamwork/RT has real-time extensions to support structured analysis for interrupt-driven applications.

STANDARDIZED ARCHITECTURE

Structured techniques can greatly improve the software development process. However, most of us write programs that tend to be very similar, e.g., testing the same sort of part with different equipment or different test options. Having a lot of the work already done would speed up the implementation phase tremendously.

Pre-written software can come in two forms: as utilities (tools), or as a pre-defined structure (architecture).

Utility modules are very useful. Some may perform commonly required computational tasks, such as FFTs. Others might make standardized reports from test results. These modules can be used anytime. An example of a software tool package is HP's DACQ/300. Written in HP 9000 Series 200/300 BASIC, it provides subprograms (modules) for computation, graphics and reports.

A software architecture goes beyond utilities. It defines classes of modules and standardizes their interfaces. An architecture may have a number of modules that work together to create a user interface (menus) or to upload information to a host computer. The Taipan software architecture is for general CAT applications written for HP 9000 Series 200/300 BASIC Language Workstations.

TAIPAN

Taipan⁵ is an outgrowth of the software methodology developed by the HP investigation. It four layers of modules: test

⁵ *Taipan* means "Big Boss" in Chinese and symbolizes the controlling nature of CAT software. It's also the name of an extremely poisonous snake in Australia.

Computer-Aided Test Software:
Do You Want To Do A Better Job?

executive, test procedure, measurement procedure, virtual instrument driver. (See Figure 7 for illustration.)

Test executive (TE). The test executive in Taipan is responsible for managing the whole show. It controls all of the non-test related functions, such as communicating with users, and executes tests as directed. The TE is custom-assembled for each specific application. (See Figure 8 for a sample structure chart.)

Test procedure (TP). This layer is responsible for test sequencing specific to a particular unit under test (UUT). It handles all test-specific information, such as test conditions, connection points, and test results.

A test procedure manages a single test related to the UUT. It utilizes measurement procedures to perform the test, and virtual instrument drivers to ensure that connections have been made.

Measurement procedure (MP). Measurement procedures are responsible for generalized measurement algorithms. They are not involved with the specifics of a UUT, nor are they knowledgeable about specific instruments. Instead, an MP is instrument-class specific (e.g. digital oscilloscope versus dynamic signal analyzer), since measurement algorithms are likewise dependent.

Virtual instrument driver (VID). The only layer allowed to communicate with the instrumentation and the UUT, VIDs isolate the rest of the architecture from specific instrumentation knowledge. VIDs are written to perform specific functions, such as setup or taking a measurement.

Comparison with FTM/300. A lot of interest has been generated by the appearance of an HP software product known as FTM/300. "FTM" stands for Functional Test Manager. Where Taipan is a general-purpose architecture, FTM is a software product designed to manage functional test systems. It has pre-defined menus and unit pass/fail reports. It also provides statistical analysis utility modules which can be used on test results for statistical quality control.

Code development for FTM is restricted to such areas as test sequencing, customized UUT and instrument handling, and specialized graphics and data analysis.

Computer-Aided Test Software:
Do You Want To Do A Better Job?

It is advisable to combine FTM/300 and Taipan for complex functional test applications by using FTM as the test executive and much of its equivalent test procedures and utility modules.

CONCLUSION

To summarize, you have been introduced to a methodology for CAT applications and a software architecture for rapid implementation.

The methodology is based on a phased approach to the software development process. The phases are survey, analysis, design, and implementation. Structured techniques are used to produce working documents at each stage. Where possible, graphical tools are used to portray the software system. Text is used in a supportive manner.

The methodology is appropriate for any small to medium-scale project environments. Attendees to the HP 05096A SA/SD course who use the HP 9000 Series 200/300 Pascal Language Workstation and those who use HP-UX⁶ found the methodology to be quite useful.

The Taipan software architecture is a general-purpose structured design for CAT applications running in the HP 9000 Series 200/300 BASIC Language Workstation environment. It consists of four layers: test executive, test procedure, measurement procedure, virtual instrument driver.

Taipan may be used in conjunction with FTM/300 for functional test applications, with FTM taking the place of the Taipan test executive layer.

The benefits to be gained from using the methodology and architecture described here should be clear: You can do a better job with them. The question is where to start?

⁶ HP-UX is Hewlett-Packard's implementation of AT&T's UNIX operating system. (UNIX is a trademark of AT&T in the USA and other countries.)

Computer-Aided Test Software:
Do You Want To Do A Better Job?

STRATEGIES FOR INTRODUCTION

Start slowly. Take the training courses earlier. Try using context diagrams, first level data flow diagrams and control flow diagrams right away. Allow for low-risk practice by using these diagrams as communication tools. This has the added benefit of cross-training others. Implement code with structured programming techniques. Build and test software according to informal test plans. Start using informal walkthroughs.

Discuss the benefits of the structured techniques and standardized architectures with others. The best strategy involves finding out what the others would like to fix in the software development process then show or discuss a structured technique that would address the problem. For example, management is generally interested in cutting costs. Using these techniques will reduce costs by identifying risks earlier, providing ways of predicting resource investment, and cutting down implementation time. All of these will lead to lowered software development costs over the long run.

Select a pilot project. It should be a of reasonable size with an estimated deelopment time of three to six man-months. Be sure that the project will result in a useful software package. Also make sure that it is visible, yet low-risk.

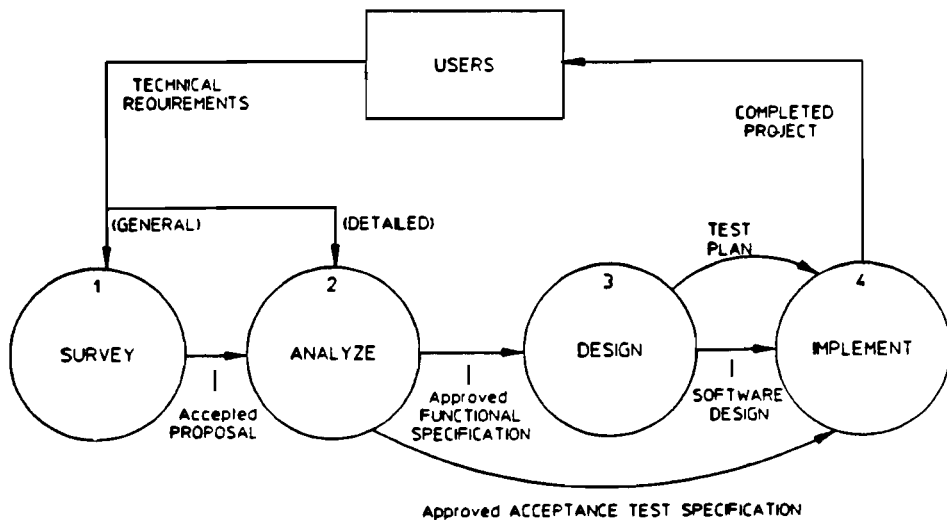
Continue to learn. Use the following list of references as the starting point for a reading list on structured techniques.

When projects get big enough, look into the utilization of automated tools. These tools may include drawing packages and word processors for small projects and organizations. Larger projects and organizations may wish to invest in one of the automated structured analysis and design tools, such as HP Teamwork.

Use common sense! Too often the methodology is implemented too rigidly or with projects that are too large and visible. Only you know what makes the most sense for your situation.

Computer-Aided Test Software:
Do You Want To Do A Better Job?

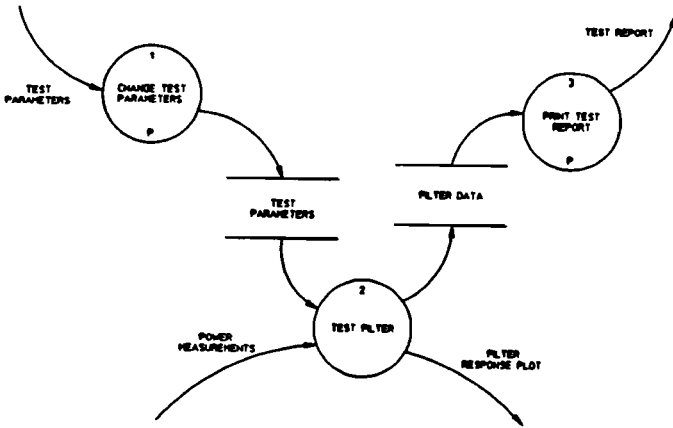
FIGURE 1: INSTRUMENT SYSTEM SOFTWARE LIFE CYCLE



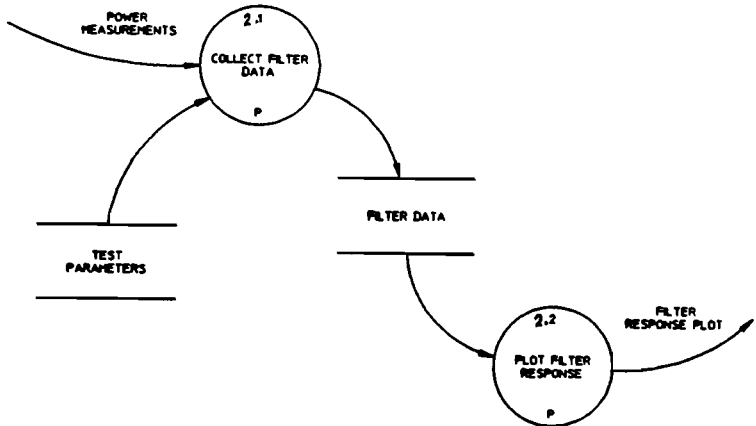
Computer-Aided Test Software:
Do You Want To Do A Better Job?

FIGURE 2: SAMPLE DATA FLOW DIAGRAMS FOR A BANDPASS FILTER SYSTEM

DFD 0



DFD 2



Computer-Aided Test Software:
Do You Want To Do A Better Job?

**FIGURE 3: SAMPLE CONTEXT DIAGRAM FOR
THE BANDPASS FILTER TEST SYSTEM**

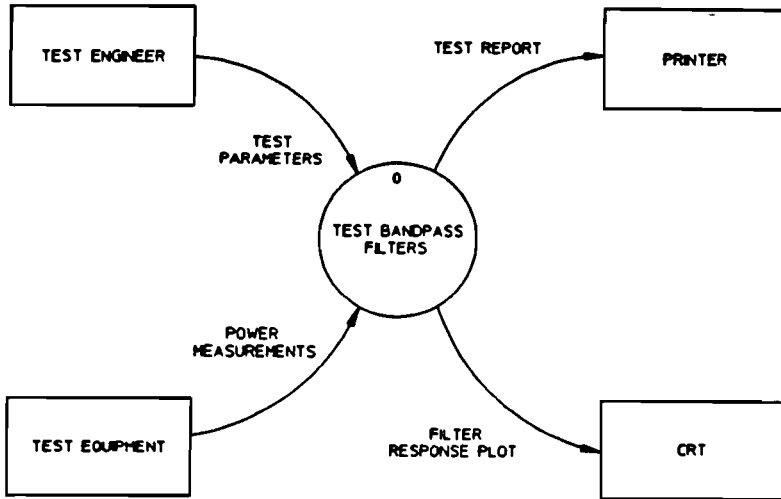
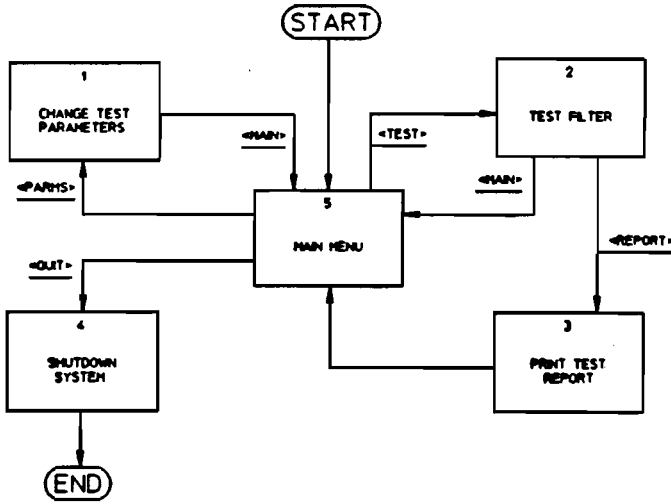
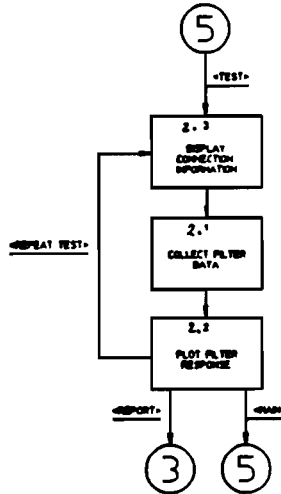


FIGURE 4: SAMPLE CONTROL FLOW DIAGRAMS FOR THE BANDPASS FILTER TEST SYSTEM

CFD 0



CFD 2



Computer-Aided Test Software:
Do You Want To Do A Better Job?
21

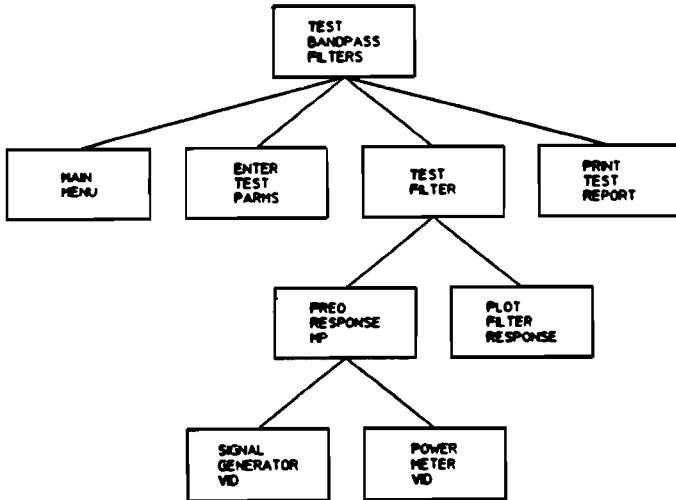
**FIGURE 5: SAMPLE MINI-SPEC FOR
THE BANDPASS FILTER TEST SYSTEM**

The mini-spec below might correspond to Process 1, *Change Test Params* on DFD 0 in Figure 2 above.

Note that the "< >" symbols are used to indicate operator actions as well as keys that are pressed.

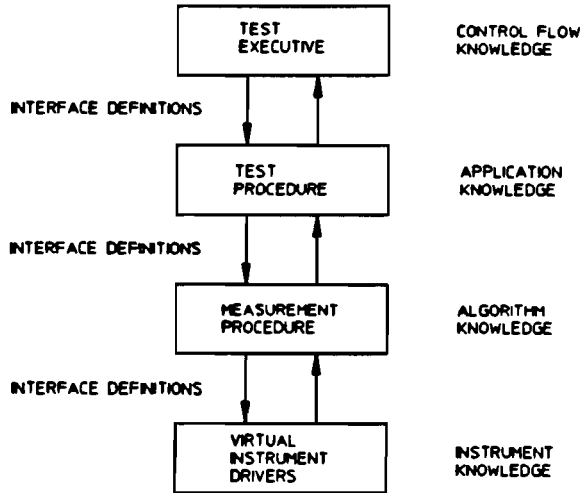
```
REPEAT
  Display the Test Parameter Menu, using current values
  SELECT Operator Action
    CASE 1: <Operator moves arrow (using knob on the
             keyboard) to the Test Parameter which is to be
             changed and presses <SELECT>>
      REPEAT
        Prompt for new value of selected Test Parameter
        <Operator enters new value>
        Check to see if new value is valid
      UNTIL valid value is entered
    CASE 2: <Operator presses <DONE>>
      Set Exit = True
  END SELECT
UNTIL Exit = True
```

FIGURE 6: SAMPLE STRUCTURE CHART FOR THE BANDPASS FILTER TEST SYSTEM



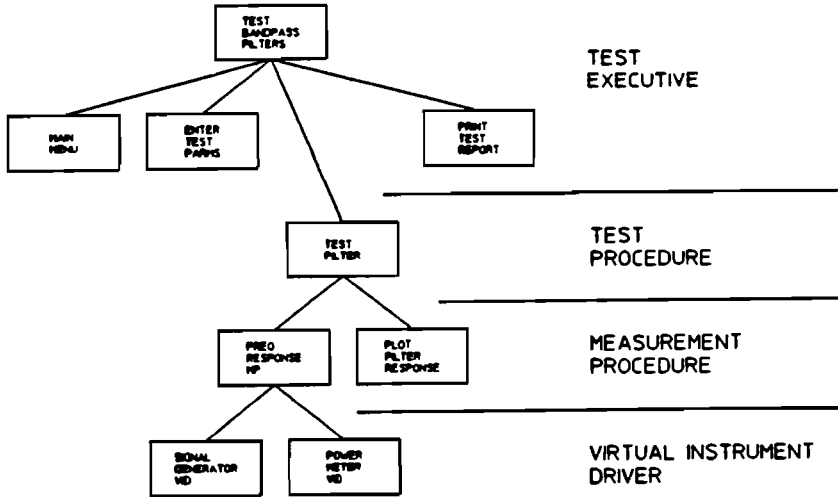
**Computer-Aided Test Software:
Do You Want To Do A Better Job?**

**FIGURE 7: AN ILLUSTRATION OF THE TAIWAN
SOFTWARE ARCHITECTURE LAYERS**



Computer-Aided Test Software:
Do You Want To Do A Better Job?

FIGURE 8: AN ILLUSTRATION OF MODULES WITHIN THE TAIPAN SOFTWARE ARCHITECTURE LAYERS



Computer-Aided Test Software:
Do You Want To Do A Better Job?

REFERENCES FOR FURTHER STUDY

Fred Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley Publishing Co., 1975.

Joe Celko, "Alternatives to flow charts: I. Data Flow Diagrams." *Computer Language* Vol. 4, No. 1 (January 1987): pp. 41-43.

Robert N. Charette, *Software Engineering Environments: Concepts and Technology*. New York, NY: McGraw-Hill, Inc., 1986.

Tom DeMarco, *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentiss-Hall, Inc., 1979.

Oscar DeWilde, "Developing Modular Programs." *Computer Language* Vol. 4, No. 1 (January 1987): pp. 51-55.

Gail Higgins, "Alternatives to flow charts: II. Structure Charts." *Computer Language* Vol. 4, No. 1 (January 1987): pp. 45-48.

James Martin and Carma McClure, *Structured Techniques For Computing* (Englewood Cliffs, NJ: Prentiss-Hall, 1985).

Meilir Page-Jones, *The Practical Guide to Structured Systems Design* (New York, NY: Yourdon Press, 1980).

Lawrence H. Putnam, *Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers* (New York, NY: Computer Society Press, IEEE Computer Society, 1980). IEEE Catalog number EHO 165-1.

Wayne P. Stevens, *Using Structured Design* (New York, NY: John-Wiley & Sons, Inc., 1981).

Edward Yourdon, *Managing the Structured Techniques: Strategies for Software Development in the 1990's*, 3rd ed. (Englewood Cliffs, NJ: Yourdon Press, 1986).

Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979).

Computer-Aided Test Software:
Do You Want To Do A Better Job?