RSTS PROFESSIONAL

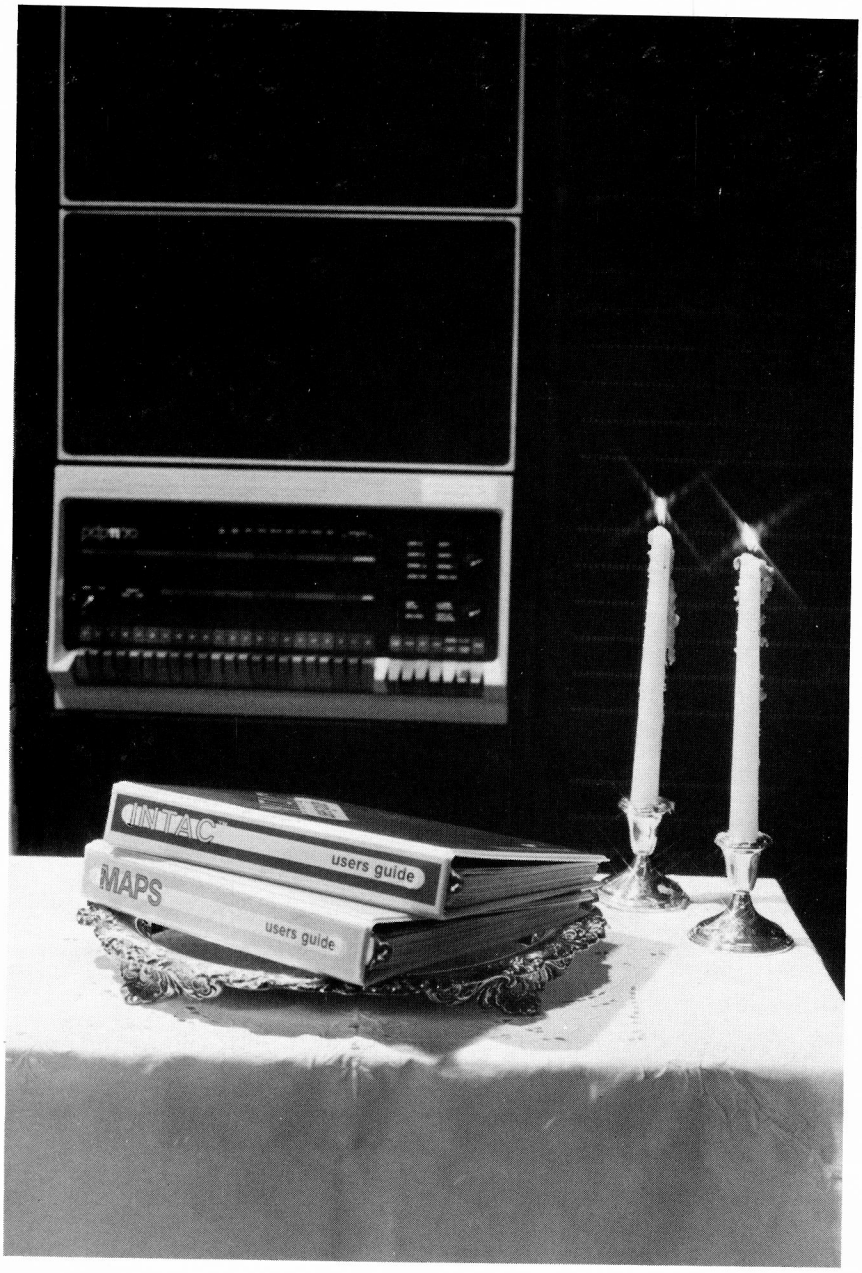
Volume 2, Number 2

May/June 1980

$7.50/issue, $20.00/year

INSIDE:

RSTS PROFESSIONAL



Two Distinguished Products for PDP-11 Users...

INTAC™
MAPS™

Interactive Data Base Management

INTAC is a new concept for data storage and retrieval that features an easy-to-use question and answer format, built-in edit rules, multi-key ISAM data access, interactive inquiry and a unique report generator.

Financial Modeling

MAPS, recognized worldwide for over five years as a leader in financial modeling and reporting, is used to construct budgets, financial forecasts, consolidations and "what if" analyses.

Ross Systems, with over seven years of proven capability, now offers these two products to current and prospective PDP-11 users. INTAC and MAPS enable business managers to produce instant reports themselves, and relieve DP managers from the pressures of special requests.

Ross Systems offers these management tools on our timesharing service, for license on existing computers and as part of a complete, in-house timesharing installation.

Call us collect for more information.

ross systems
incorporated

1900 Embarcadero Road, Suite 208, Palo Alto, CA 94303 • (415) 856-1100 • Other offices in San Francisco and Los Angeles

If you're looking for speed, a fast track and reliability, enter the Southern 600.

Enter the 600 line-per-minute band printer that costs less than somebody else's 300 lpm printer.

The Southern Systems 600 lpm band printer is about as good as you are going to get.

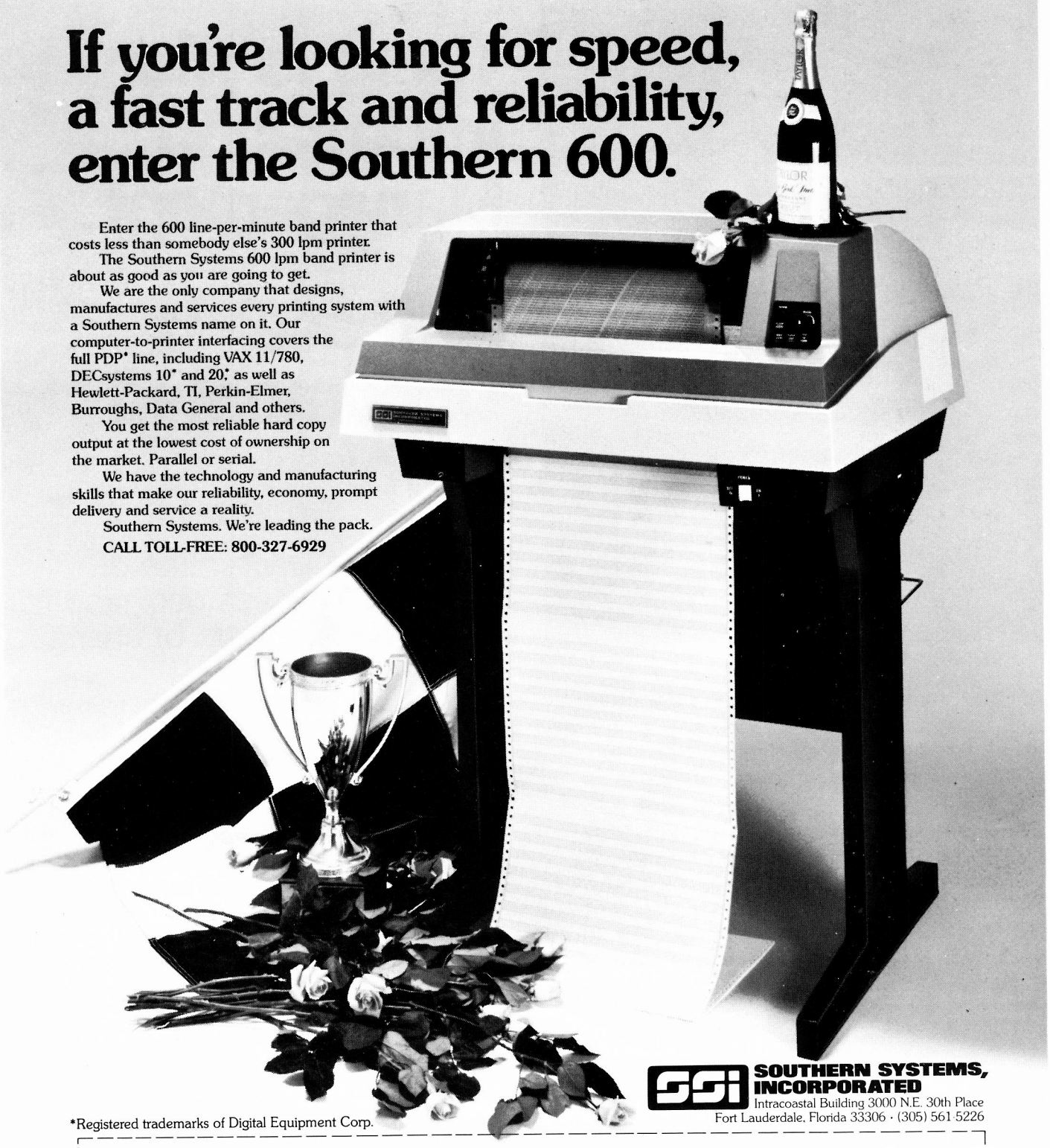
We are the only company that designs, manufactures and services every printing system with a Southern Systems name on it. Our computer-to-printer interfacing covers the full PDP* line, including VAX 11/780, DECsystems 10* and 20,* as well as Hewlett-Packard, TI, Perkin-Elmer, Burroughs, Data General and others.

You get the most reliable hard copy output at the lowest cost of ownership on the market. Parallel or serial.

We have the technology and manufacturing skills that make our reliability, economy, prompt delivery and service a reality.

Southern Systems. We're leading the pack.

CALL TOLL-FREE: 800-327-6929



SOUTHERN SYSTEMS, INCORPORATED
Intracoastal Building 3000 N.E. 30th Place
Fort Lauderdale, Florida 33306 · (305) 561-5226

*Registered trademarks of Digital Equipment Corp.

O.K., Speedy, tell me about the following printer system(s):

- ☐ 200 lpm impact matrix
- ☐ the B series (300 or 600 lpm band)
- ☐ the 2200 series (300, 600, 900 lpm drum)
- ☐ the 1200 series (600 to 1200 lpm Chain Train)
- ☐ the 2550 (1500 lpm charaband)
- ☐ Serial Interfacing
- ☐ Parallel Interfacing

My computer is a _____

My requirements are: ____ immediate ____ 3-6 months ____ information only.

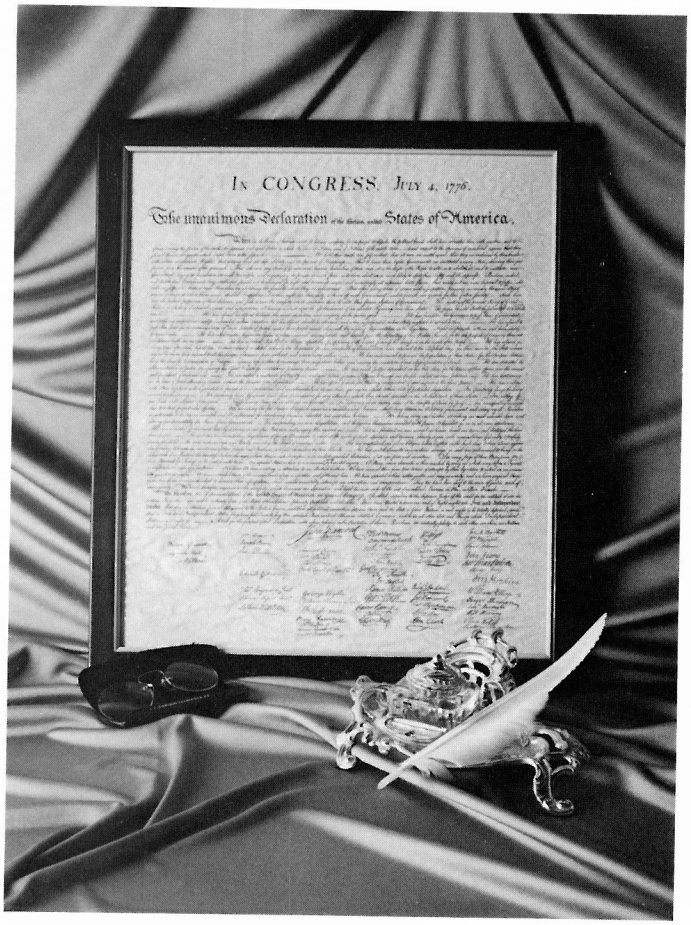
Name _____ Title _____

Company _____

Address _____

City _____ State _____ Zip _____

Telephone ( ) _____



There is More to Information Management . . .

Ambase® is Documented Proof

Just as The Declaration of Independence was designed to accommodate years of change...

AMCOR feels an application development tool using data base management structures must also be designed to accommodate change—the changes in your business.

Our combined DBMS and Application Development System, AMBASE, embodies this philosophy. With the use of AMBASE, you realize a savings in development time of 50%-90%, and automatically both ease of modification and maintenance are built into your system. Providing for change today is imperative to the management of your information resource.

State-of-the-art sophistication and human engineering have never been so uniquely merged into a Data Base Management System.

For complete information on AMBASE, or any of AMCOR's complete line of DATA BASE ORIENTED applications, give us a call or use the coupon provided below.

amcor®

1900 PLANTSIDE DRIVE • LOUISVILLE, KY. 40299 • 502/491-9820

SOFTWARE FOR DEC*/RSTS SYSTEMS

Please forward information on the following systems:

amcor

1900 PLANTSIDE DRIVE
LOUISVILLE, KY. 40299
502/491-9820

- ☐ AMBASE
- ☐ Accounts Receivable
- ☐ Accounts Payable
- ☐ GL/Financial Mgt. (AMFACS)
- ☐ Payroll
- ☐ Order Processing
- ☐ Inventory Control
- ☐ Sales Analysis

Name _____

Company _____

Address _____

City _____ State ___ Zip _____

Telephone _____

Computer Type _____ Operating System _____

*DEC is a TRADEMARK of Digital Equipment Corp.

Contents

In Next Issue...
August/September 1980

- In-Depth String Handling Analysis
- More Basic-Plus-2
- More Performance
- Trouble with V7.0!
- How & Why TECO?
- New Programs

The following items were to appear in this issue but space would not allow it. However, they will appear next issue.

- How to Add ON/IN Memory
- Continuing RSTS Communications: Multiplexers and Modems
- More Disk Anatomy
- The World of 300 MByte Disks
- Site Profile: The RSTS Credit Union System
- More . . .

The RSTS Professional Magazine, June 1, 1980, Vol. 2, No. 2. Published quarterly. Single copy price $7.50, $20.00 per year. Published by M Systems Inc., 753 Johns Lane, Ambler, Pa. 19002, telephone (215) 542-7008. Send all correspondence and change of address to: Box 361, Ft. Washington, Pa. 19034. Application to mail at second class postage rates pending at Ambler, Pa. 19002.

From the editors...

I'm confused by what I hear around the DECUS meetings about funding the various LUGs and SIGs. For the uninitiated, Special Interest Groups (SIG) and Local User Groups (LUG) form the heart and soul of DECUS. When you attend a DECUS meeting, the RSTS SIG has set up the program within the meeting for you. They do this with lots of unsung hard work and planning and they do it very well. The RSTS PROFESSIONAL has attended DECUS meetings in the U.S. and Canada this year and we can attest to the skill of the SIG planners. The Local User Groups (LUG ...) vary from sedentary to dynamic and have between 1 and 500 members.

Now, How do these groups exist within DECUS? . We don't pay dues to DECUS; DEC, we presume supports it. Then DECUS supports the SIGS and the LUGS. This support is changing. The SIGS and LUGS are going to be responsible for more of their own funding. How can this work? Let me give you an example of a successful LUG.

The New York Metro Lug has a membership in excess of 300 and meets once a month for dinner in The Big Apple complete with after-dinner speakers and lots of good company. They have a newsletter that is published once a month by their leader, John Runyon, called Computers-R-Digital. They fund themselves by dues, dinners, and other innovative ways. They are a viable, valuable and interesting group (if you are in New Jersey, New York, or Connecticut, let them know and join the LUG; send your name to us if you can't find them). John does lots of work from home on his HOME Computer running RT-11 (he emulates RSTS).

There is an independent LUG forming in New England (see their article, "DECUS-PLUS or Independence for the RSTS Community"). They presumably will not be tied to any DECUS rules or limitations. They won't have DECUS support (mailing lists, etc.). If all LUGS are required to pay, then maybe they won't need or want DECUS jurisdiction.

How does your local LUG stack up? Poorly, I'll bet. It's going to get worse! How in the world do you fund a non-commercial, non-profit venture like a LUG. If that's a problem, think of our SIG with its 8500 estimated members; how will that be funded?

I am a member of the TECO SIG. There is a neat little card that has all the TECO commands on it and can be a big help to TECO programmers. It is out of date, needs revision and is being sold at $1.00 each by DEC. The RSTS PROFESSIONAL offered to print new cards (up to date, of course) and sell them with all profits split between us (we'll make the investment and do the work) and the SIG. Does anyone hear me? Or are we too commercial?

What will become of the substructure of DECUS without monetary support? Should the SIGS charge dues? Should DECUS charge dues? Should the LUGS charge DUES? I'm more confused than ever!!

Does DECUS owe us an explanation? It is our user group ... isn't it?

Carl B. Marbach, Editor

I just got back from DECUS Chicago. I have expended lots of energy (mental and physical) over five years trying to make DECUS into something it isn't. What is DECUS? It's DEC's trade show. They own it lock, stock and barrel and that's that. Someone wise once said, "It's much easier to ride the horse in the direction he's going." This magazine was created from that wisdom. DECUS is for talking with DEC. The RSTS Professional is for talking to everyone. DECUS is a context designed by DEC to achieve their corporate aims. These include user feedback to developers, wish lists, and other pulse-taking techniques. It is a priviledged, non-public forum where manufacturer talks directly to customer in a safe environment. It is highly commercial, but only for DEC. It survives only because the user benefits enough from the interchange that he or she is willing to pay for the session. For the last several meetings I have felt a growing dissatisfaction with the context of the meetings. DEC is always very unwilling to talk about the next release so soon after a new release. With no release-related gossip to distract us, we were left with the choice of say-nothing product panels or hearing for the third or fourth time that RSTS has a large future in being small. If this doesn't change, users will stop coming and DEC will lose a great thing. I personally feel that DECUS is excellent and worth having. I am involved in the RSTS SIG steering committee and am willing to do my part to improve the next meeting in San Diego. Now that I am clear about DECUS — what it is and isn't, I can make a list of all the things DECUS isn't now and may never be, and see what we can do to help some of them happen.

Some things DECUS isn't:

- A free forum
- A place to see non-DEC alternatives
- A trade show
- A place to sell magazines

Things DECUS should be and has not been lately:

- A two-way conversation with DEC
- A forum for top-quality user papers

This magazine already is many of the both lists, but it is only a one-way conversation with DEC.

There are a number of successful NON-DECUS lug's in both the U.S. and Canada. They, like this publication, are successful because they do the things that DECUS can't or won't do.

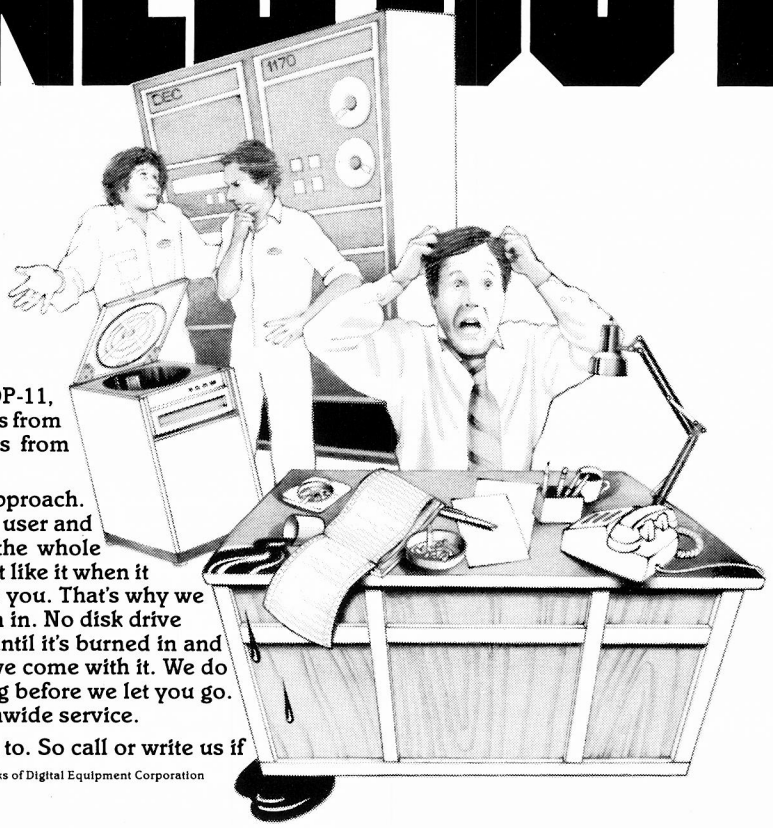
I hope to gather a bunch of these independents together for a meeting soon and see what we can do together.

R.D. Mallery, Editor

RSTS PROFESSIONAL*

Editorial Information: We will consider for publication all submitted manuscripts and photographs, and welcome your articles, photographs and suggestions. All material will be treated with care, although we cannot be responsible for loss or damage. (Any payment for use of material will be made only upon publication.)

BURNED IN OR BURNED OUT



When you order a disk drive for your DEC* PDP-11, 11/70 or VAX, you expect it to work. If it comes from Nordata you can be sure it will. If it comes from anywhere else . . . well, there's no telling.

So what's so special about Nordata? The whole approach. We've been in this business a long time, both as a user and supplier. We know how frustrating it is to have the whole system go down because of a rotten disk. We didn't like it when it happened to us, and we don't want it to happen to you. That's why we offer the industry's longest and most rigorous burn in. No disk drive — 300,200 or 80 megabytes — leaves our plant until it's burned in and error free for at least a week. And when it leaves we come with it. We do the installation, so we know you're up and running before we let you go. And then we back the whole system up with nationwide service.

At Nordata we take the beating, so you don't have to. So call or write us if you're tired of being burned. *DEC & PDP are registered trademarks of Digital Equipment Corporation

Nordata. Your one stop DEC* shop for peripherals

NORDATA
4433 27TH AVENUE WEST
SEATTLE, WASHINGTON 98199
(206) 282-1170

LETTERS to the RSTS Pro...

Dear Editors:

Please consider the accompanying article for publication. We expect it to be of considerable immediate interest to your readers. Thanks.

Sincerely,
Howie Brown & Monica Collins
SE New England RSTS Users Group
Pawtucket, R.I.

We are in receipt of your article, "DECUS-PLUS or, Independence for the RSTS COMMUNITY" and would like to use it in the May-June issue of THE RSTS PROFESSIONAL.

Thank you for submitting it to us. We are always looking for comments and opinions from the RSTS community.
(See Mr. Brown and Ms. Collins' article, "DECUS-PLUS . . .", in this issue.)

—

Dear Sir:

Congratulations on your new publication. I like it, the boss likes it, and so do the programmers. How about an article on measuring system performance. We're running version 7.0 of RSTS and have tried using the programs STATUS and QSTATS.

The output from these two are not very compatible and they seem somewhat at odds. I have tried DEC, DECUS and other users for information on QSTATS, but to no avail.

Perhaps you know of someone who knows about QSTATS, and would be willing to give me his name. With the advent of data caching it would be helpful to get some statistics on how well it performs.

Sincerely,
Wendell Peterson
United Industry, Inc.

Thank you, Wendell. We'll try to help you soon. Stay tuned!

—

Dear Sirs,

Enclosed is a subscription application and a check for $20.00 in payment for a year's subscription to your journal. I have just finished reading the November/December issue (borrowed from a colleague) and wish to congratulate you on such a fine publication, long needed in the RSTS community. I would like to purchase a back copy of all issues to present, if that is possible. Please provide a list of available back issues and cost.

Thank you,
Edward T. Keegan
New Haven, Conn.

Back issues of the RSTS PROFESSIONAL are now available. Information is provided on the subscription cards and throughout this issue.

Dear Dave,

Enclosed is a copy of an open letter to RSTS users which I would like for you to consider publishing in the next edition of the **RSTS Professional**. It concerns the Spooling Package Task Force which was created at the Chicago DECUS Symposia.

I would also be interested in any input you might have concerning our project.

Sincerely,
Jerry Kiestler
The University of Tenn. at Martin
Computer Center, Martin, TN 38238

See Mr. Kiestler's Open Letter in this issue.

—

Dear Mr. Marbach:

I received your letter, concerning advertising in your publication, with a great deal of interest. I received the first 2 copies of the journal and am very impressed with the high level of professionalism they exhibit.

I have sent material for an ad in this issue.

Cordially,
Bobby E. Sharp, General Manager
Plycom services, inc.

We thank Mr. Sharp for his positive comments and support.

—

Gentlemen:

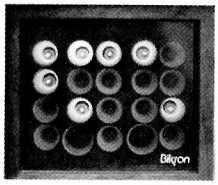
We shall be happy to subscribe to your magazine just as soon as you bring subscription prices down to a reasonable level.

Sincerely,
John Muller, Data Proc Mgr
Zipatone, Inc.

We're sorry, John, but you'll have to keep on reading a borrowed copy of the RSTS Professional. There's no such thing as a free lunch.

—

DO YOU REMEMBER THIS?



(Photo contest, RSTS Professional, Vol. 2, No. 1, p. 64.)
A RSTS T-shirt is on its way to the readers listed below who have correctly identified the above photo as a BINARY CLOCK which "says" 3:51:14. We thank all who participated for making this feature fun.

Photo contests will appear in the RSTS PROFESSIONAL occasionally and readers will have until the last day of the second month of publication to submit their answers (ex.: February/March 1980 readers will be required to have their answers postmarked no later than March 31 of that year). We may, from time to time, limit the number of correct answers eligible to receive prizes (ex.: the first 10, etc.).

Here are the entries in order of receipt:

. . . continued on page 31

Due to time and space limitations, Chapters Three and Four of Cathy's article will be continued in the next issue of the RSTS Professional, August/September. This action was taken to avoid cutting any portion of this very useful manual.

NEW USER'S MANUAL FOR RSTS/E

By C. Galfo

ACKNOWLEDGEMENTS

I would like to thank all of the people who encouraged me on this project and I hope they will be gratified by the result. My only regret is the small amount of time I had to spend on this manual, which continually forced me to narrow my choice of topics. Barry Gershenfeld, my right hand and brain at work, did an excellent hardware overview in Chapter One — **thanks Gershe!** For your comments, etc., our address is University of Virginia, Division of Biomedical Engineering, Box 377 Medical Center, Charlottsville, VA 22908.

PREFACE

This manual is written for system managers and system programmers new to the RSTS/E operating system and the PDP-11 family of minicomputers. The real system manuals, though considered "pocket guides" compared to non-DEC manuals, currently consist of a three foot mountain of information not cross-referenced between volumes. As a result, it is often difficult to find answers to questions posed by new users. In an effort to make the task of learning easier, I propose a more relaxed approach offering common sense, problem-solving techniques, and humor. What follows is a loose formalization of working experience compiled over several years and from many sources. The author does not wish to be held accountable for technical information appearing in this document, though praise, suggestions, and questions are encouraged.

INTRODUCTION

What is a system manager?

If you are a newcomer to DEC computer systems, then the term "system manager" will probably be unfamiliar to you. The titles of "system programmer" and "application programmer" are well-known throughout the industry, but, as I am learning in my job search, the non-DEC world does not understand my current function. Attempts have been made to pigeon-hole me as "a four year BASIC-PLUS programmer", who has, therefore, not used a "hard" language and who has, as a consequence, no value in the marketplace. I resent that description, and am fighting for the right to use those managerial skills I have had to have to make my system reliable and efficient. Here is, from my resume, a definition of a system manager's job: "responsible for the daily administration of PDP 11/70 mini-computer running twenty-four hours a day, seven days a week, year-round, . . . Duties include the tailoring and maintenance of system programs, operating system conversions, data base management, and the supervision and training of software engineers, programmers, and data entry personnel. Consult with users, coordinate user activities, and produce user programs and manuals . . .". In the non-DEC world the above would represent a mythical cross between the jobs of computer operations manager and DP manager.

The typical route to the top is an abrupt promotion from programmer (taking orders) to system manager (giving orders), a switch which requires a great mental change. The purpose of this manual is to remove some of the fear of new responsibilities, by presenting lessons from collective experience. Hopefully, your work will be made more enjoyable in the end.

Barry Gershenfeld, author of TAPE, RSTS Professional, Vol. 2, #1, presented some of this material at Spring 1980 U.S. DECUS.

A Not-So-New User

Chapter One

Hardware for System Managers

In this chapter I will attempt to familiarize you with the physical parts of your computer and how they are put together. Perhaps you have looked inside the cabinet and wondered at all the seemingly extra boxes and all those wires which look more like ribbons than cables. You will not become a hardware expert, but hopefully you will learn enough that you can refer to your device manuals for detailed information, or talk somewhat intelligently to your hardware service rep. So let's examine the insides of your machine, working from the end of the UNIBUS back to the central processor unit.

UNIBUS. The UNIBUS (DEC trademark) forms the backbone of any PDP-11 computer system. It constitutes a 56 wire party line on which any device of the system can talk to any other device. The ribbon cables interconnect each device from one to the next. Inside each device, the UNIBUS takes the form of a backplane into which the various circuit boards are plugged. Eighteen of the bus lines carry address information; another 16 lines are for the data. The rest of the lines are used for synchronization, handshaking, interrupts, and initialization signals.

Bootstrap Terminator. Electrical signals travel at the speed of light down the bus and will bounce right back when they come to the end of the cable. For this reason, a terminator card is placed at the last position of the UNIBUS to soak up any echoes that will try to come back and foul up things. DEC also locates the bootstrap program on this card, since, as long as you have to have this card, why not put something else useful on it? The terminator is a bunch of resistors; the program is in the form of serveral read-only memory chips.

Devices. The central processor is, of course, the most important device on the UNIBUS. Although the bus originates in the CPU, the CPU does not have unique control of it. Any of the other devices can "get on" and request data. This is the reason for so many control signals.

The peripheral devices that can be added to the UNIBUS include circuits to interface terminals, readers, punches, printers, mass storage devices, instrumentation, arithmetic units, and anything you might wish to build yourself. The 1976 Peripherals Handbook lists some 64 devices.

Controllers. The controllers are devices which exist between a physical device (what you see) and the interface inside the computer. They are found on mass storage devices that need lots of preprocessing of data. For instance, a disk drive reads and writes bits on your disk pack, but the controller groups them into words, counts the words as they are moved, checks the parity, and even corrects the data if it can. It also contains all the control and status registers that you see in an ERRDIS printout. Controllers are also called formatters. Simpler devices, such as terminal interfaces, have the registers right on the interface card and do not process data.

Memory. Main memory is a read-write data storage device which has one of the fastest access times of any of the storage devices. "Fast" in this case is less than a millionth of a second! This is where the action is. A good deal of CPU activity involves reading and writing memory. Memory can be thought of as a big data array occupying the lowest UNIBUS addresses, starting from address 0. Anytime your job is active, it is stored in main memory. Main memory takes the form of either magnetic core memory, or semiconductor ("MOS") memory, the latter being made entirely from integrated circuits. The reason it's called MAIN memory is to distinguish it from hardware cache memory, or from mass storage which is sometimes called disk memory.

Cache Memory. The memory found in top-of-the-line processors has an even faster access time thanks to cache memory, an additional amount (1K) of high speed memory interfaced with main memory. Anytime a word is read from memory the hardware looks first to see if it's in the cache. If it is (a "hit"), everything is fine and you have your data in 1/4 the time it would have taken otherwise. If you "miss", then you have to get the word from main memory. Meanwhile the new word is added to the cache, the reasoning being that a program spends most of its time re-accessing data since it uses many loops. On writes, the cache is updated to agree with memory if the old word is presently in the cache. The hit rate is 80 to 90 percent, giving an average access time improvement of 3 times the speed of main memory alone.

Parity. In any system where the error rate is low, you can add an extra bit to a data word (or byte) and be able to verify if any bits change. For instance, main memory really is made up of 9 bit bytes, or 18 bit words. You count the bits and if there are an even number of them you set your extra bit to make the number odd. If there are already an odd number, you leave this bit off. This is called odd parity. You can just as well reverse the rules and shoot for an even number and then you have even parity. The trick, you see, is that if you change any bits, including the parity bit, you no longer have an odd number of bits, and therefore, an error condition. If 2 bits change, then you are in trouble, but if you have that kind of error rate, you're in trouble anyway. Using more than one parity bit per word, you can even tell which bit changed. This gives rise to an error correcting code, which is used in ECC memory and on some disk drives. Most of the devices on the system use some form of parity, including magtapes, DECtapes, and terminals.

Memory Management. One of the most useful hardware items a timesharing system could want is something to help it keep track of all those jobs in memory. Actually it relieves the operating system from having to worry about much of this activity. Memory management features three main operations: relocation, segmentation, and protection. Relocation allows the operating system (and the user) to treat each job as though it occupied the lowest locations in memory. A relocation register offsets the memory address referenced so that the actual location is then automatically computed from these two. Indeed, the physical memory limit imposed by a 16 bit machine is 32K words; through the use of relocation, up to 2 million words can be addressed. Segmentation allows blocks of memory to be divided up so that a job's work area does not need to be contiguous, although through the use of relocation registers it can be made to look contiguous. Protection takes the form of several processor modes (User, Kernel, Supervisor). Using the various modes, a job can be limited to reading only its own allocated space. In addition, certain instructions can be made privileged (so you can't execute a HALT, for instance.)

CPU's. The central processing unit is a whole book in itself. Consequently, your computer should have come with a PDP 11/?? Processor Handbook. I will briefly run through the different procesors of the PDP-11 family.

The LSI-11 and LSI-11/23 processors are part of the 11 family in that they make use of the Macro-11 instruction set, however their bus is not a UNIBUS and therefore not compatible with PDP-11 computers.

The 11/04 processor is the smallest CPU and its logic is all on one circuit board. The memory is interfaced directly from the UNIBUS.

The 11/34 has the added feature of memory management and is the smallest processor which will support the RSTS/E operating system. A 1KW memory cache can optionally be added to this CPU.

The 11/45 has memory management, and in addition, features a separate memory bus so that memory cycles can be performed independent of UNIBUS cycles for added speed.

The 11/55 processor features two separate memory busses, one for core and one for solid state memory.

The 11/60 has the features described so far, and besides its unique front panel it also has microprogrammable instructions allowing you to add your own custom instructions to the normal DEC instruction set. The 11/60 also features a 1K memory cache.

The 11/70 is the mainstay of the large minicomputers. It has the separate memory bus, memory caching, and memory management that its brothers have (but not the microprogrammable machine code of the 11/60), and while the other machines could address up to 128K words of memory, the 11/70's memory management permits addressing of up to 2 million words! Furthermore, it has a direct high speed interface to memory from certain mass storage devices, which is explained next.

High Speed I/O. The RH-70 interface is used typically to interface magtapes and disk drives in the 11/70 CPU. It connects the UNIBUS to the device controller as usual, but when blocks of data are read or written they are transferred over a separate bus called the MASSBUS, which is a direct connection from the device controller to main memory. This is what helps make the 11/70 the fastest computer of the PDP-11 family.

Chapter Two
The Language of RSTS/E

2.1 CONVERSATIONAL RSTS/E

I'm fond of quoting an infamous definition made by an old friend, who, at the time of this remark, knew very little about computers. When asked to define the term "software", he answered, "a person or persons who program a computer." The persistent confusion surrounding computer related terminology inhibits communication not only between people representing different vendors, but among people using systems from the same company. For RSTS/E there exists a large collection of jargon, and an equal number of glossary type terms. In order that we might better understand each other at conventions and on the telephone, here are some candidates for each category. Several "words" appear in both lists, so that the reader can choose the definition that best fits the conversation or audience.

RSTS/E	Jargon
Sysslash-"C":	Where C is one or more letters. Request for a SYSTAT.
Directoryslash-"C"	Where C is one or more letters, each separated by a "slash". Request for a directory of an account.
Percent:	Variable delimiter. Also denotes an integer.
FOO:	DEC variable name that replaces DUMMY.
Pip-it:	How to get a printout from the computer.
Queue-it-up:	To send messages to batch processors.
Spool-it:	How to get a printout. Same as "pip-it", but your KB is not tied up while printing is in progress.
KB:	Your terminal, "KB colon", or someone elses, as "KB ten".
Console:	The terminal you never use, but always has messages printed out on it, usually in the middle of your printout.
Of:	Computer programmer jargon for "(".
F:	Referring to floating-point numbers.
Bayospool:	The correct pronounciation of BAOSPL, the first batch receiver name.
Ristis:	It is never pronounced R S T S!
BP2:	BASIC-Plus-2 was too much to say.
Dotdotdot:	Any RSX job.
Force it:	Do nasty things to some else's job.
Kill it:	Get rid of the offender, and/or his file.
Clear-the-system:	Kill all jobs that don't dump their own accounting statistics. Also, remove data entry jobs.

Dump: Write to disk. Usually refers to a large number of blocks output to disk, usually for diagnostic purposes.

Minus-"N": Where N is a number between -128 and 127. Denotes job priority.

Takedown: Shutdown of system, as in "do a takedown".

Crash: Unscheduled takedown. The unexpected termination of a job or timesharing.

Renault: Jargon for RNO.TSK, the text formatter with a fascinating hyphenation algorithm.

RUNOFF: The original text formatter CUSP, still unsupported.

BATCH: Package of programs similar to DEC X11, in that it tests the load capacity of a RSTS/E system.

EDIT-11: Your better BASIC editor; easy for beginners.

EDT: An editor that has taken one small step back for man, and a giant step forward in support.

TECO: The editor for programmers with big egos. DEC's answer to APL.

Error-copy: For ERRCPY, the CUSP that logs user and system errors (sometimes).

Tittyset: For TTYSET, the CUSP that sets terminal characteristics.

Beepus: Binary Program Update Service. Supplies Software Product Dispatches (SPD), which are commonly known as "patches".

Core exceed: To exceed the runtime system (16K for BASIC-Plus) or system swap max (up to 31K). All variables and arrays are zeroed.

Garbage collection: The reason why BASIC-Plus programs using lots of strings may run slowly. The repacking of strings still in use to make room for new strings.

Glossary

CCL: Concise Command Language—used to issue system commands. To invoke a program without using the RUN command.

CUSP's: Commonly Used System Programs—BASIC-Plus versions of system commands.

EDIT-11: The text and program page editor with a medium-sized buffer (2-4K Bytes).

EDT: MACRO line editor with a small text buffer.

TECO: Complex editor/word processor considered a language by some. Text buffer can be as big as the system swap max.

RNO: MACRO version of RUNOFF, but with lots of little differences and still unsupported.

Protection codes: Eight bits in a file's directory name entry that determine the file's read and write accessibility. They are useful once learned. In general, RSTS/E automatically protects to a higher degree than you would think.

Cache: Generally, any area of memory reserved for the temporary storage of data. A PDP 11/70 has a physically separate cache which augments main memory; however, most references to caching refer to the area in main memory called XBUF.

Cache hit: The next data that the CPU or user wanted was in the cache area.

Cache miss: The next data that the CPU or user wanted was not in the cache area and had to be fetched.

Run-time system: A preprogrammed set of instructions that make your program work, and which can communicate with the monitor. One step above a resident library.

Resident library: A read-only set of instructions that can be shared by different users.

SPR: Software Performance Report—form sent to DEC when reporting software and documentation problems.

Supported: If a program or system bombs, DEC will answer an SPR on it. They are legally bound to resolve the problem to your satisfaction.

Unsupported: If a program bombs, DEC may choose to help you but they are not legally liable for any malfunctions.

Source: Any file that can be listed and edited, and can be run or transformed into a runnable form; for example, files with the extensions of .BAS, .MAC, .FOR, and .B2S.

Private pack: Disk pack restricted to privileged users. Cannot be used by the system without explicit instructions.

Public disk: Any disk available to all users, and on which the system is free to create files and accounts.

Reorder: Term referring to the restructuring of disk directories to reduce directory look-up time. Two actions are possible: restructure only, or restructure and sorting of files by date.

Swapping disk: A non-file structured disk used exclusively for the storage of non-running jobs.

Swap: To copy from memory to disk (swap out), or vice versa (swap in), the contents of a user's job area.

Swap slots: Areas in a swapfile, each the size of the current swap max. SYSTAT shows the location of any swapped out job by A10, B05, etc.

CTRL/T: One line SYSTAT for the job attached to your terminal.

Disk bound: Program continually in I/O wait for disk (DB'ed, DF'ed, DK'ed, etc. in SYSTAT).

Tape bound: Program continually in I/O wait for (the movement of) tape (MM'ed, MT'ed, MS'ed, etc. in SYSTAT).

Idle time: People who still have front panel lights can watch the computer not doing anything.

Lost time: What is left over after you add up the percentages of computer time used by USER, I/O, Idle, and EXEC, and the total is less than 100%.

Sleeping: The temporary suspension of program execution (SL), which leaves the job eligible for swapping ("sleeping on disk"). Similar to the action of the PAUSE statement in FORTRAN.

Hibernating: A detached job attempting to perform I/O to channel 0 will stop execution and appear in SYSTAT as HB.

2.2 THE SYMBOLIC LANGUAGES OF RSTS/E

There appeared in a recent issue of a computer industry trade journal an article on the number and complexity of languages facing today's programmers. An average of a dozen syntactical and conceptual languages can masquerade as one high-level language and operating system. The mastery of each of these mini-languages, referred to as "symbolic languages" by the author, is required for the successful completion of each step in a data processing task, such as data base definition or documentation.

I thought it would be worthwhile to apply this theory to the RSTS/BASIC-Plus environment, so that new users would have another perspective on what, and how much, there is to learn. There is an additional benefit for experienced users, system managers, and their superiors: the information below can be used as a yardstick for measuring the technical skills of junior programmers. For those of you running the other DEC languages, it should not be hard to develop a similar evaluation tool.

Ten Symbolic Languages for RSTS/E

1. Algorithmic Language — IF..THEN..ELSE, GOTO, FOR-..NEXT, GOSUB, ON..GOTO, ON..GOSUB, etc.

2. External Data Description and Conversion Language — MAT READ UNIT$, DIM KB.ASSIGN%(10%,10%), FIELD, etc.

3. Internal Data Language — CVT$%, SWAP%, %, CVTF$, RAD$, CHR$, ASCII, etc.

4. Job Control Language — OLD, COMPILE, RUN, etc.

5. Monitor Command Language — SYS calls, PEEK, /MODE, < 248 > , all of which are MACRO-like functions for BASIC-Plus.

6. System Utility Language — PIP, TTYSET, LOGIN, SYSTAT, QUE, DIRECT, all of which are BASIC-Plus versions of the fifth language.

7. Debugging and Diagnostic Language — Immediate Mode, BPCREF, /DUMP option, CONT, MAT PRINT, etc.

8. Editors — TECO, EDIT-11, EDT for programs and documentation.

9. Text Formatters — RUNOFF, RNO for information retrieval and documentation.

10. Plain English — for communication with users.
The most skilled programmer will be able to work in all ten languages, but if you aspire to a system manager's job, you will need at least two more symbolic languages.

11. Hardware Configuration Language — CPU, DH11, etc., and device functions.

12. System Generation and Library Language — The CUSP's are each a language unto themselves, making up a myriad of "dialects" that direct system operations.

Finally, a system manager has all of language twelve stored for instant recall in his or her head; that is, every last bit of command syntax and system trivia. Even though DEC issues more manuals with each new version, the goal is to be as free of them as possible.

. . . continued in next issue.

CANADIAN DECUS

Toronto

By Carl Marbach, Editor, RSTS Professional

Why would anyone go to the Canadian DECUS instead of the Spring bash in Chicago? Because it's closer, friendlier (??), has a tutorial the day before the meeting, and the Canadian Sister of the American DECUS has a flavor that we found more genuine. The American DECUS seems to be hung-up in its own politics (see Editorial) while the northern counterparts are interested in information exchange and the well being of its members.

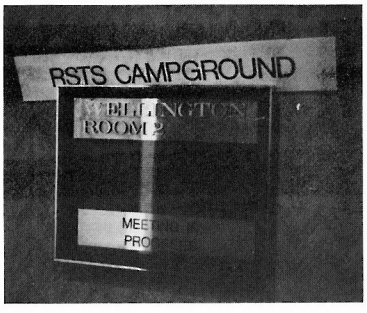
"Didn't realize Toronto was this close", my co-editor remarked as we passed over Lake Erie descending to 1500 ft. and the Toronto Island airport in Downtown Toronto. It was just under 2 hours and 15 minutes since we left the northern limits of Philadelphia in our light plane. A short ferry and taxi ride left us at the Toronto Hilton in time for the 9:30 beginning of a RSTS tutorial on performance. While most people arrived via scheduled airline at the Toronto International, and limo'd to the Hilton, we were energy efficient (22 gallons, U.S.). [The value of light aircraft in transporting people quickly and efficiently has been lost amid the recent energy mania, but that's another story.]

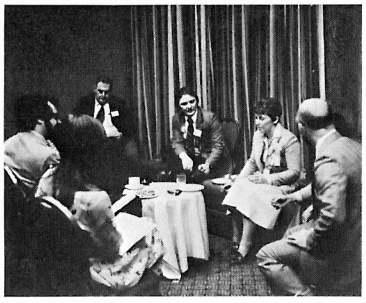
The Canadian counterpart to the U.S. DECUS meetings offers the same benefits to all comers. The machine room (complete with VAX that crashed every time the elevators started or stopped; neat power fail!), and had the usual array of hardware for hands-on experience (games?). The ever-present DECUS/DEC bookstore did the usual land office business and gave away free(?) rulers.

The meetings were three days of dawn to dusk workshops, lectures, tutorials, and discussions in all the hardware and software worlds of DEC. Many of the sessions were similar to those presented in San Diego or Chicago, but with enough new thoughts to satisfy us that the venture north was well worth the effort. After dinner there were more meetings, campground discussions, and at last the FIRST 10th birthday party for RSTS. Simon, and many of the original and current development team were present and told secret stories about the real inside world of RSTS. Lest we ever doubt the emotional involvement that these people have, Simon was rumored to have bought his own airline ticket to the party! Thank you, Simon. Have any of you seen an 11/20 running RSTS (or at least the null job)? It was doing just that at the birthday party; remember RSTS began as a small sharing-BASIC-only system on an 11/20 (and they say 11/23 can't run RSTS!).

The Canadian members of DECUS are sometimes thought of as secondary members. They are FIRST CLASS. Their applications, expertise, and organizational skills are something they can be proud of. It is time that their U.S. counterparts started understanding this fact. The American manufacturers should also pay more attention to the Northern RSTS Customers; they could use more Canadian representatives and sales offices. From our subscription list, we know that there are a significant number of Canadian RSTS sites. Go to it!

We won't miss another Canadian DECUS if we can help it. The atmosphere and people are just too good. We need that feeling to help us along throughout the year. Thanks Canada!





PRESCRIPTION FOR AN OLD PROGRAM

By Rob Davidson, President, Timesharing Consultants of Pennsylvania, Inc.

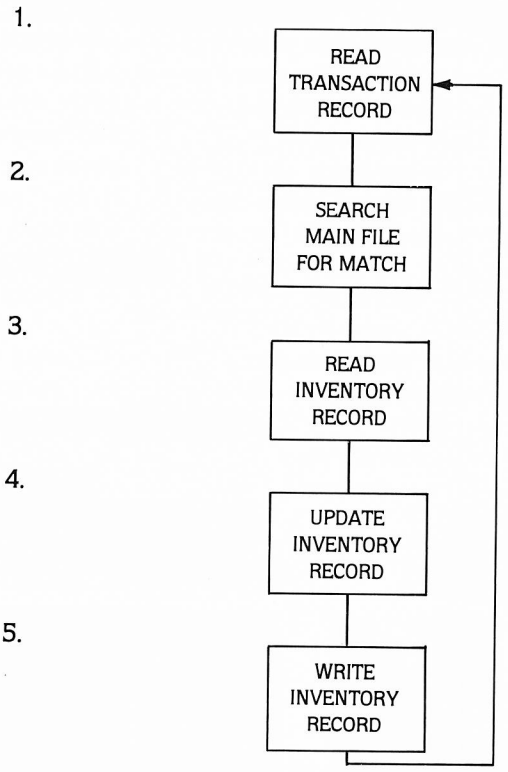
TCI serves dial-up timesharing customers on two RSTS systems in eastern Pennsylvania (11/35 and 11/70). The firm specializes in conversion of programs from other commercial vendors such as Call/370. Rob is also a contributing editor to the **Auerbach Computer Technology Reports** where he evaluates software packages.

My company has been providing remote timesharing services on RSTS systems to customers in the eastern Pennsylvania area for the past 5 years. Prior to that many of my customers had used the Call/370 Time Sharing Service. As a result, a rapid conversion technique was required to bring programs and files over from the 370 system to RSTS. Because of my modest programming abilities and the constraints of time, many of the file oriented Call/370 applications were converted using virtual array files as their basis under RSTS. Virtual arrays are very easy to use in accessing disk files on a random basis and simple to understand for the novice programmer. Single virtual string arrays were used with numeric data packed in using the CVT functions. Most of these programs have been running without problem for 5 years.

THE PROBLEM

An inventory program has had major and minor alterations over the five year period. The number of inventory items has grown from 2,000 to over 5,000. Recently, I received a call from this inventory user that the updated program had taken over 5 hours to complete an update of only 100 items. My first assumption was that the system must be terribly overloaded or my user had been assigned the lowest of low priorities. Unfortunately, neither of these was the case. Could it be the fault of my program, which had performed flawlessly (and somewhat rapidly) for the past five years? It could!

As I have already mentioned, this program, used as its file access a virtual string array. This array was assigned a length of 256 characters and contained 49 subfields, a mixture of alphabetic and numeric data (both integers and floating point). The file is accessed on a random basis using an integer array as an index to the record sequence. This core array had been assigned a dimension of DIM I% (6999%). The program was currently at the system limit of 16K words or 32K bytes. Even a single additional line of code would cause a "maximum memory exceeded" error. In fact in a recent program change it was necessary to consolidate 5 lines of program code into one line in order to add a single PRINT statement. The structure of the program is illustrated below:



An analysis of each of these program blocks yielded the following results. In the Step 1. the simple reading of the transaction could not be improved since only a single INPUT LINE statement was used. Next, Step 2. utilized standard binary search routine finding most records in 5 seeks or less. Step 3. read the record identified in the previous step and broke it down in the 49 fields making generous use of MID and CVT functions to breakout the various alpha and numeric fields. The updating Step 4. simply exchanged or added to or subtracted from existing fields depending on a given transaction code. The final Step 5. then rewrote the virtual string array element by re-assembling the forty-nine fields in statements like the following

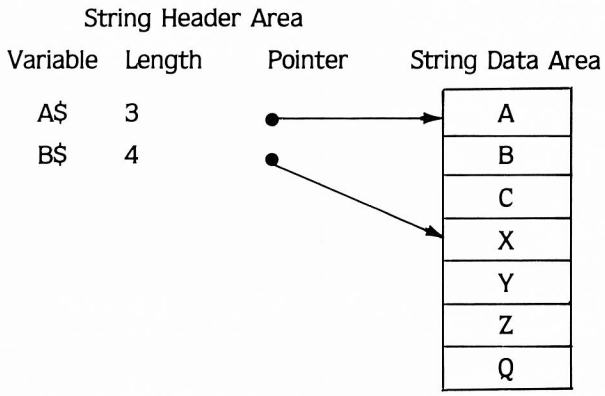
```
I$ = A$ (1)+A$(2)+A$(3)+A$(4)+. . . . .
```

HOW DO YOU SPELL RELIEF?

The problem called for a two-phase solution. The quick solution was to reduce that index array I% (6999%) to a size more closely aligned to the actual record count. Thus a new DIM I%(5499%) saved 3K bytes and brought the program under its 32K threshold. This reduced the run time for the update program from 5 hours to 45 minutes. But better results were yet to come.

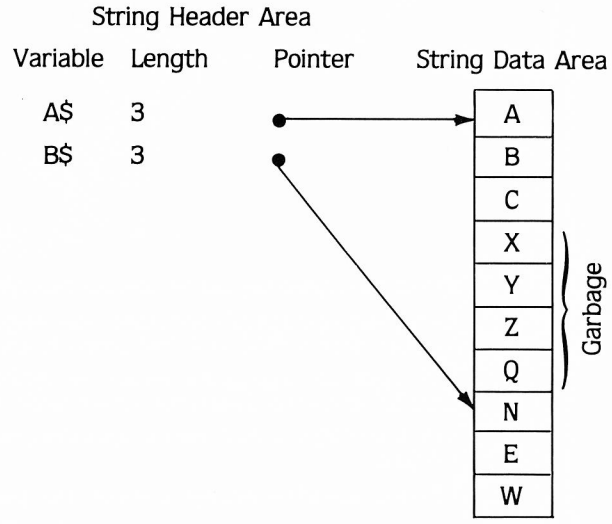
First let us consider the problem of the RSTS/E Core Recycler sometimes called the "Garbage Collector". Borrowing from T.R. Sarbin's File Processing Guide (known to be a hand-out at DEC's Introductory Basic Course), here is a short course in **RSTS internal** string handling. When a string identifier is used a string header holds the length of the string and a "pointer" which locates the start of the string in the string data table in core.

```
10 A$ = 'ABC'
20 B$ = 'XYZQ'
```



Each time a string is changed or re-assigned, the old string becomes garbage and the new string uses a new set of positions in the data table. See the change as shown in the chart below:

```
30 B$ = 'NEW'
```



As the program proceeds, garbage is created in the string data area. In fact, this area gradually fills up until there is **no more room** for strings to be created. When this happens, the RSTS/E Core Recycler is called and the strings are all collected into the beginning of the string data area.

When the program is at the 32K core limit and the string data area is therefore restricted, then the core recycler must

be called in more frequently resulting in the five-hour run time of the inventory updating program.

By reducing the size of the index array I%( ), enough core space was made available to reduce the number of times the "garbage collector" was called. The second phase in the renewal of the up-dating program reduced the amount and nature of string handling.

To complete the final changes needed, a null buffer was opened and set at the exact size of the inventory record.

```
OPEN 'NL:' AS FILE 7%, RECORDSIZE 256%
```

Next this buffer was fielded to match the 49 fields of the record

```
FIELD #7%, 15% AS A$(1%), 4% as A$(2%),. . . . .
```

with a final overlapping variable for the entire buffer

```
FIELD #7%, 256% AS I9$
```

These steps reduced the amount of program code in two ways. First there was no longer any need to make extensive tests on the length of various alphabetic strings to make sure they would fill the virtual array properly. These statements:

```
A$(1%) = input string
Y% = LEN(A$(1%))
IF Y% < 15% THEN A$(1%)
  = A$(1%)+SPACES(15%—Y%)
      ELSE
IF Y% > 15% THEN A$(1%) = LEFT(A$(1%), 15%)
```

were replaced with:

```
LSET A$(1%) = input string
```

This could be done because once the length of string variable has been fielded, then any LSET statement either automatically truncates any excessive characters or blank fills the field (on the right with LSET or on the left with RSET). Secondly, the record reading routine was reduced to:

```
LSET I9$ = I$ (Z%)
```

where I$ (Z%) is the virtual array. A series of CVT functions were added for the numeric data in the record.

The final program alterations were to change the A$(x) assignment statements in the updating portion to LSET A$(x). These changes caused a result of 5 minutes run time for a 100 transaction record update or a **6000% improvement in throughput!!**

SUMMARY

A 32K byte virtual array updating program using an oversized core index integer array and excessive string manipulation was taking 5 hours to update a 5000 record file with 100 transactions. The program response was improved by reducing the size of the core index array initially. Further significant improvements were made by reducing the string handling through the use of a null buffer, FIELD statements, and LSET assignment statements. These changes eliminated the need to call the RSTS Core Recycler for resetting the string data area and yielded a sixty fold improvement in program throughput.

BOOK REVIEW

By Carl B. Marbach

A GUIDE TO PROGRAMMING IN BASIC-PLUS
Bruce Presley, Stuart Hayes,
Theodore Graham, Harriet Morill
Lawrenceville School, Lawrenceville, NJ $10.00

If your installation is typical, there are lots of users who would like to know more about the computer and how to work it. In my career, more than one data entry person has graduated to Programmer. There is a book that is an elementary text written specifically for RSTS and BASIC-PLUS.

The authors allow that this text was written for the "classroom" environment. That is, they expect it to be taught rather than used as a self-teaching text. The marketing philosophy has also been to sell class-sized quantities rather than single issues. Once we are past these philosophies, the book is just what we have needed for some time, and I have lots of ideas on what to do with the books when I get them; but first let's talk about the book.

The book is organized into three main parts: The first six chapters provide general instruction in the BASIC-PLUS language and some applications. Chapters seven to nine are for the more advanced mathematically or technically oriented person, and the last three chapters discuss business and related applications. Each chapter is followed by a section of exercises, to reinforce the concepts presented and to ferret out and problems or misconceptions the student might have picked up. These exercises also "desensitize" students to the computer by letting them see that, at their level, they can't yet destroy large amounts of data or the whole computer by a single RETURN.

Although this is not as large as DEC's two volume BASIC-PLUS teaching guide, it is complete. The explanations at the elementary level are clear and concise and the student is likely to be successful in both the exercises and learning basic programming. The guide is just a guide and not a reference manual. While the explanations are clear, they are not as detailed as the larger DEC Teaching Guide. This is less important earlier in the text, but as the concepts grow more difficult it could be a problem.

I gave it the ultimate test. Our controller has been itching to do some of his own programming, but he has never done any before. I gave him the book as a self-teaching device and gave him his own RSTS account. In several days he was doing elementary matrix manipulations and wanted to know how to store data using block I/O. He is now bogged down in the more difficult I/O but he has a beginning facility with the language. I think that is a pretty good result.

If there is enough interest, the RSTS PROFESSIONAL will purchase a large quantity of the books and distribute them to subscribers who are interested in single copies. Write to me at our box. You won't be disappointed in the book.

HOW TO JOIN DECUS

Send for a membership application to one of the addresses listed.
State that you want to join the RSTS SIG.
Tell them that you received this information in the ***RSTS Professional.***

Australian Chapter
DECUS AUSTRALIA
P.O. Box 384
Chatswood
NSW 2067
Australia

European Chapter
DECUS Europe
P.O. Box 510
12, Av. Des Morgines
CH-1213 Petit-Lancy 1/GE
Switzerland

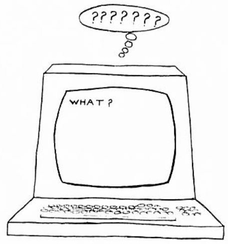
U.S. and
CANADIAN CHAPTERS and others
DECUS
MR2-3/E55
One Iron Way
Marlboro, MA 01752
U.S.A.

Great Newsletters

SORLUG
Linda Adams, Sorlug Treasurer
Alcan Canada
Box 269, Toronto Dominion Centre
Toronto, Ontario M5K 1K1

COMPUTERS-R-DIGITAL
Directory Database Inc.
Box J
Navesink, New Jersey 07752

DEAR RSTS MAN



DEAR RSTS MAN: My salesman always seems to want to sell me more core. Why?

Signed, Hardcore

Dear Hardcore:

The cure-all for RSTS performance is 'more memory'. If you have any doubt, add more. For example, a large 11/70 running 50 jobs can easily use up two megabytes.

One great way to use up all that extra memory is with XBUF, the 70 above could constructively use 128K + WORDS of XBUF — and that's with only a modest amount of data caching.

DEAR RSTS MAN: Everytime I open a file for output, the whole machine stands still for 30 seconds and my RP06 runs like a washing machine on spin cycle. What's wrong?

Maytag

Dear Mr. Maytag:

One of the worst 'machine eaters' is a very large account (> 150 files), actively used for output that is not reordered almost daily. I have seen systems lock-up for more than a minute while the disk drive goes crazy following all the linkages in such a UFD. Most poorly performing systems have several of these accounts.

There are several solutions. The disk shakes during such situations because the clusters that make up the UFD are scattered randomly all over the disk. The individual linkages of the directory elements may very well require head movement hundreds of times for even a few files. This is eliminated by pre-extending the UFD to its full intended size during the initial set-up of the disc. Thus, all clusters are probably in the same cylinder.

The system locks-up, because a directory look-up is a FIP function that might alter the directory from happening till the current action is done. We can't do anything much about that, what we can do is speed up FIP. Pre-extended directories, frequent REORDR and lots of XBUF accomplish this end. 'Large FIP' in 7.0 also helps with shared files.

Of course, the other solution is to never try to use the RSTS directory structure as an ISAM index. It really isn't!

DEAR RSTS MAN: When I use VTEDIT, the scope editor of TECO on my VT100, it acts just like a VT52? I want all the neat features of the VT100 like reverse video. How have I gone wrong?

Sincerely, Scoped Teco

Dear Scoped:

The VT100 has only one attribute different than the VT52 as far as TECO is concerned: XON. Set this using TTYSET and you're home (?) free.

DEAR RSTS MAN: We are trying to use BASIC PLUS 2 and RMS. But first we wanted to use a NON-RMS BASIC PLUS 2 program with the BASICS shared library. We PIPed the library and all other files to the correct areas and made sure the protection codes were correct. When we task build using this shared library we get many error messages. Why doesn't it work?

Brave Soul

Dear Brave Soul:

The RSTS MAN notes with interest that you did not BUILD the shared libraries using the approved BUILD procedure supplied with your kit. This BUILD procedure patches the BASICS.LIB file to make it compatible by changing some internal module name from BP2 to something else. You can find the BP2 reference with ODT (in radix-50 in one of the first 50 locations) and change it yourself or go through the DEC-approved BUILD.

DEAR RSTS MAN: I am having a problem with rounding. I need to round to Dollars and Cents. Do you have an easy way?

Round and Happy

Dear Round and Happy:

Dollars and Cents have always been close to the RSTS MAN's heart so here is what we use.

```
DEF FNR(X)=INT(X*10.**2%+.5)/10.**2%
```

Or you can work with Double precision integers (SEE Vol. 1, No. 1) and divide by 100 to get to Cents.

Please send your DEAR RSTS MAN questions to: DEAR RSTS MAN, P.O. Box 361, Ft. Washington, PA 19034.

AUTHOR!!!

The RSTS Professional wants you to be an author. Send us your article of interest to the RSTS community.

Be sure to mail back your Subscription Card in time for the next issue of the RSTS PROFESSIONAL

Special RSTS Instructions

AOI	Add and overflow integer
BSF	Branch and swap forever
CRAB	Convert Rad50 to Ascii and back
DQM	Destroy the que manager
ECC	Erase core common
FDD	Find and destroy data
GBQS	Go to Batch que and sleep
HPP	Hold for proper priority
IDS	Inspect and deschedule scheduler
JOM	Jump out of memory
KAJ	Kill all jobs
LDO	Leave data out
MMT	Mangle mag tape
NNN	No No No
OAD	Open and destroy
PSD	Put and scratch disk
QAG	Quit Adventure game
RSM	Remove swap medium
TAD	Take all devices
SWR	Seek wrong record
UAC	Uncouple all couplers
VGC	Vacuum Garbage collector
WLT	Watch lost time
XP	X-ray programmer
ZTB	Zippy task build

Editor's Note: This article was reproduced exactly as we received it from the author. This was done to show you just how good word processing output can look directly from a computer. Word processing can be done using DEC's WPS-8 (a PDP-8/Floppy) word processing system alone or connected to RSTS. Data Processing Design's WORD-11 runs directly on RSTS and is a superset of WPS-8. This article is applicable to both of these systems.

WORD PROCESSING WITH DEC COMPUTERS
HINTS AND KINKS

Lawrence H. Eisenberg
17141 Nance Street
Encino, California 91316

ABSTRACT

This paper discusses various handy hints and kinks associated with the use of Word Processing Systems on the PDP-8 and PDP-11 systems. While developed primarily with WPS-8, the routines and hints offered generally are applicable with WPS-11 and other commercially available systems currently utilizing the WPS-8 format. The discussion presented below consists of the various matters presented at the Spring 1980 Symposium in a Panel Discussion with Ms. Vicki Ann Rose, Digital Equipment Corporation, Merrimack, N.H., Mrs. Nancy Evans, Federal-Mogul Corporation, Southfield, Ml and this author.

LIST PROCESSING HINTS

FIELD IDENTIFIERS AND DATA PROCESSING

The "lists" which are developed in list processing often are useful for data processing activities as well as many of the Word Processing and List Processing purposes. For PDP-11 users, many of the data files developed under Word Processing may be addressed directly by data processing. However, for PDP-8 users the WPS-8 files (which are saved in a format similar to COS-310) cannot be addressed directly by COS-310 or OS/8. While the WS-200 series originally was disigned to provide for direct communication between Word Processing and COS-310, this feature no longer is supported and the WS-200, as with all other WPS-8 systems, requires conversion to utilize the files. (Conversion utilities for both COS-310 and OS/8 are available through the DECUS LIBRARY. These utilities transfer list processing type files between the various systems. The conversion procedures are not discussed in this paper.)

It is most helpful, therefore, to maintain the LIST FIELD IDENTIFIERS as upper case characters. While the DEC WPS manuals show the field identifiers (e.g. - <field1>) as lower case fields, such was not meant to be a required form for identifying the fields. The use of lower case by DEC was a throwback to computer manuals which used lower case to indicate operator decisions, as opposed to upper case which indicated mandatory acts.

Since each of the WPS-8 systems utilizes special characters to indicate lower (and upper) case shifts, any conversion program is going to require considerable additional (and wasted) time in order to perform the conversion, as each of the special characters will have to be stripped from the field before the data can be used by the data processing system.

If there is even the remotest possibility that your list files will be used in data processing, it is important to avoid the use of hard [the RETURN key] returns except at the end of a field identifier. In other words use one identifier for every line of text. For example:

DO NOT USE

```
<NAME>John Doe
<ADDRES>123 Any Street
Our Town, U.S.A.
    00123
```

DO USE

```
<NAME>John Doe
<ADDS1>123 Any Street
<ADDS2>Our Town, U.S.A.
<ADDS3>
<ADDS4>
<ZIP>00123
```

In many conversion programs, and nearly all data processing programs, the carrier returns within a field will be read as a terminator, and the information following the return will be lost during the conversion or use by the program.

While the use of several fields may appear somewhat cumbersome at first, the benefits soon become very apparent. Also, the more available fields, the easier it is to edit and to SORT!

SELECTION SPECIFICATION - TO SELECT ONLY IF SOME CHARACTER EXISTS

The DEC manuals fail to disclose the selection specification which can be used to select a record only if a field has information. The wild card specifications presented by DEC are <?> and <*>. The <?> is used to replace a letter (i.e., it must be preceded or followed by some character other than a <?>). The <*> is used to define a field as containing ANY OR NO characters.

From time to time it is necessary to select a record ONLY IF A GIVEN FIELD HAS SOME INFORMATION. There are two possibilities; the first example given is the most reliable:

```
(1)  if<field5> =<?><*>
     then process record

(2)  not if<field5> =
     then process record
```

(The use of lower case is for example, only. USE UPPER CASE!) The last example would follow other qualifiers; if used alone the results are not predictable.

DELETING UNUSED LINES FROM FINAL OUTPUT WHERE THERE IS NO DATA

Regretably this author allowed an article to be published upon this subject (12-BIT Nov. 1979) which contained an inaccuracy. The information which is presented in this paper is correct, and has been tested under several conditions. (The prior article presented a situation which would work only if the field size was known in advance.)

DEFINING THE PROBLEM: EMPTY FIELDS ON LINES WHICH SHOULD NOT BE PRINTED. The problem which often is encountered is how to eliminate blank lines which are printed when there is a field which is empty, but which has been defined in the form. We will use an address block as an example.

```
<NAME>
<TITLE>
<COMPANY>
<APT/SUITE#>
<ADDR1>
<ADDR2>
<ADDR3>
<CI/ST/ZP>
<DROP>
```

In the example presented it is obvious that several of the fields might not be present in the final printout. The individual may have no title; s/he may not be associated with a company; there may be no apartment or suite number; there may only be a single address line. However, if the FORM is created in the manner indicated, which, in the example (and only by way of illustration) would be the same as the LIST, the final output would be printed with blank lines for each line on which there is missing data.

<u>There is a solution.</u> It takes a little planning, but once understood, it is simple to apply to every situation. (Just keep in mind, however, that this solution will cause each missing field to disappear and to bring the following line up one line feed! <u>You must remember to allow for this, if the missing lines could affect other line-count features of your form</u>.)

The first step is in the creation of a FORM. To accomplish the desired result for any set of circumstances <u>it is necessary to create two FORMS.</u> The first FORM should include only the variable information, and will, itself, become the LIST which then will be used to create the actual FORM or PRINTOUT. **THERE CAN BE NO SPACES OR TABS ON ANY LINE WHICH MAY "DISAPPEAR", EITHER IN THE ORIGINAL LIST OR ON THE FORM.** (Adjust the Left Ruler in lieu of a single tab, if indentation is desired.)

The FIRST FORM is created to determine which, if any, fields are not present and automatically to create a "wrap", as opposed to a HARD RETURN, for each such field. It also is used to create the second LIST. To accomplish this, it is necessary to create "soft" returns on each line which may not have information upon a field. This is done by using dummy rulers <u>after</u> each line which reasonably is expected to "wrap". Using the LIST above, and assuming that EVERY LINE may possibly have a missing field, we could create a form as follows [NOTE THE RULERS!]:

```
L----------------------------------------------------------R-----------------
.   1   .   2   .   3   .   4   .   5   .   6   .   7   .   8
.......0.........0.........0.........0.........0.........0.........0.........0

<<NAME><NAME>
LT---------------------------------------------------------R-----------------
<<TITLE><TITLE>
L----------------------------------------------------------R-----------------
<<COMPANY><COMPANY>
LT---------------------------------------------------------R-----------------
<<APT/SUITE#><APT/SUITE#>
L----------------------------------------------------------R-----------------
<<ADDR1><ADDR1>
LT---------------------------------------------------------R-----------------
<<ADDR2><ADDR2>
L----------------------------------------------------------R-----------------
<<ADDR3><ADDR3>
LT---------------------------------------------------------R-----------------
<<CI/ST/ZP><CI/ST/ZP>
----------L------------------------------------------------R-----------------
<<DROP><DROP>
LT---------------------------------------------------------R-----------------
<<>
```

Note that each of the rulers is identical, except for the dummy tab which follows every alternate ruler. The only purpose for the tab is to create a new ruler which can be imbedded. (If the rulers were identical, they would all disappear, and the method described could not be used.) Also note that the last line, DROP, has been indented by changing the left margin. **The "indent" feature may be used on any line and is used to avoid the insertion of**

tabs or spaces which necessarily will defeat this utility. Also note the "<<>" identifier to create the new list! (Down arrows indicate hard returns which may be observed with GOLD VIEW.)

Using the blue EDIT keys, proceed to the beginning of each line AFTER A LINE WHICH MIGHT RESULT IN AN EMPTY FIELD. Use the Blue LINE key to travel from line to line. With the cursor on the left margin, strike the RUB CHAR OUT key ONCE. (This will delete the hard return and, upon a GOLD VIEW, will disclose a funny looking circle at the end of the sentence, instead of a down arrow.) Repeat for each line which might reasonably be expected to have an empty field. [If it becomes necessary to edit the last letter, back the cursor to the end of the line -- this will place it under the last letter -- and insert the new characters. The last letter will continue to travel and, if undesired, must be deleted.] Run the List Processing feature, creating a document. The document created by this feature will, itself, become the LIST for the second part of the program.

Upon completion, you will have created a form which, when operated with the List Processing Feature, will result in a new LIST which will have "wraps" in each empty field, between a ruler. YOU MUST BE CAREFUL TO AVOID TABS OR SPACES IN EMPTY FIELDS AND IN THE FORMS or this utility will not work properly.

The SECOND STEP is to create another FORM, which is identical to the first, except for the special field indentifiers.

From the following illustration (on the next page), note that that extra field identifiers have been removed. This will be the final list and will eliminate the spaces between lines which otherwise would have been created as a result of unwanted fields.

If you should find spaces between lines, the problem most likely will be that tabs or spaces were imbedded in either the FORMS or the original LISTS. Check them carefully.

The following form is such an illustration:

As with the first FORM, line feed to the beginning of each line AFTER the field which may not be present, and enter a RUB CHAR OUT to delete the hard return. (If you merely copied the document, be careful, as you may delete a character from the preceding line. To edit this problem, BACK UP to the preceding line (you will be on the last character). Re-type the last character (the one which is above the cursor) and any character which was deleted. Finish with a hard return. Delete the remaining character above the cursor, which should remove the hard return, also. (Check with GOLD VIEW.)

Now, USING THE NEW LIST CREATED BY THE LAST FORM AS YOUR LIST DOCUMENT, run the list processing again. This time, the new document (which also can be a direct PRINT) will cause all of the empty fields to "fold" upon themselves, so that all of the rulers with "soft" returns will collapse and final output will be without lines between information. While all rulers will appear on the screen, there will be no returns within them; the printer will skip to the next line of text without printing the "empty" lines.

If it appears that there is a space between rulers on which there was no data, check to see if there had been a space or tab on either of the FORMS or LISTS used for the procedure. Check your original LIST with GOLD VIEW. Each empty field's right arrow should be followed by an immediate down arrow (without a space).

Remember, you only have to create the two forms ONCE. They can be used for every processing run. (Actually, you need create the form only once, and then add the extra field identifiers to one of the forms. If you should get a line wrap, because of the extra space required by the new field identifiers, don't worry. The program automatically will adjust.)

PROGRAMMING NOTE: Although you can use the same **selection specification** for both forms, you also can use the simple specification of "process record" for the second run, as you already have specified the records to be used.

```
L-----------------------------------------------------R------------------
.    1    .    2    .    3    .    4    .    5    .    6    .    7    .    8
.....0.....0.....0.....0.....0.....0.....0.....0.....0.....0.....0.....0

<NAME>
LT----------------------------------------------------R------------------
<TITLE>
L-----------------------------------------------------R------------------
<COMPANY>
LT----------------------------------------------------R------------------
<APT/SUITE#>
L-----------------------------------------------------R------------------
<ADDR1>
LT----------------------------------------------------R------------------
<ADDR2>
L-----------------------------------------------------R------------------
<ADDR3>
LT----------------------------------------------------R------------------
<CI/ST/ZP>
------------------L----------------------------------R------------------
<DROP>
LT----------------------------------------------------R------------------
```

COMFORT NOTE: Although this may appear somewhat clumsy, it actually is rather easy and once you get the hang of it, you will find the procedure very useful!

USING LIST PROCESSING TO CREATE AN INDEX OR TABLE OF CONTENTS

Presently there is little ease with which to create an index or a table of contents with the existing WPS-8 or WPS-11 systems. While 11-based systems will semi-automatically create an index and Table of Contents, and most dedicated word processing systems do the same, some ingenunity is required to acomplish this with DEC's systems (although we are assured that this, too, will change some day!).

For the time being, a fairly long document can become a LIST document using the following procedure.

First, copy the document over to another location (or on another diskette), as you are going to alter it (i.e., destroy it) considerably.

Second, decide on some easy shorthand for the catagories you are going to use with your index or table of contents. For example, you might wish to use <H> for headers; <N> for names, etc. Choose a character to be used as a dummy field identifier, e.g. <X>.

Enter a terminator and the dummy field identifier in the PASTE buffer, as you will be using it quite a bit during this exercise. (To enter it in the paste buffer, type it and then cut it.)

E.g.: <><X>

Start the document with the dummy field (e.g., <X>) and proceed to the first data which is to be used in the Table of Contents or Index. Let's suppose the first data is a header, which will use the <H> identifier. Enter a terminator <> and field identifier <H> immediately preceding the header and then enter the PASTE immediately after the header. Thus, the document would appear something like this:

```
---------- t o p  o f  p a g e ----------
<X>

  (miscellaneous data)

   <><H>TITLE OF DOCUMENT<><X>


<><H>First Subheading<><X>

  (miscellaneous data)
      <><X><N>(desired name)<><X>
  (miscellaneous data)

<> [entered as last character in document]
-------- b o t t o m  o f  p a g e ----------
```

In the same manner, identify the different titles throughout the document, such as names, subtitles, books, etc., until you have identified each item which will be used in your index or Table of Contents.

CAUTION: As you proceed through the document, enter the PASTE in a random manner (i.e., insert the dummy field identifier <><X>) about every 2/3 screen, or more often. This is necessary as no field may contain more than 1500 characters, and to avoid an error message you will have to insert the dummy field every so often. It doesn't matter how often you use the dummy field, as it never will be referenced during list processing.

At the very end of the document, be sure to enter a terminator <> or an error message will occur (it won't affect your program, but no error is more comforting than some buzz error which might leave some doubt).

After proceeding through the entire document, you can create a very simple FORM and SELECTION SPECIFICATION. The FORM may consist of a single entry (e.g., <H>). The selection specification may be "process record". Operating the List Processing, then, will transfer each data identified with the <H>, and will skip all of the rest. (If you have to format the output, it will be much easier to do so after running the list processing.)

Also, if you have the type of document which might require some form of sorting, such as alphabetical listings, you can perform some minimal alphabetical sorting by use of the wild cards in your selection specification. (This will require several runs through the list processing; e.g.: if <N>=A* then process record, will pick up every name starting with an upper case A, etc.) If there are only a few records, then use of the cut and paste feature will probably result in an easier, as well as faster, alphabetical processing.

Another feature, which will result in much faster opeartion if several field identifiers are being used, is to utilize the double LIST feature (i.e., create a new LIST with a single pass). To create a new LIST, set up your FORM (for the above example) as follows:

```
<<H><H>
<<N><N>
<<>
```

Processing the entire document will fill a new document with each field, in a random manner, and you then can run a second pass which will be more selective as to the order in which you want the items to appear. All of the dummy <X> field data will be omitted from the new LIST.

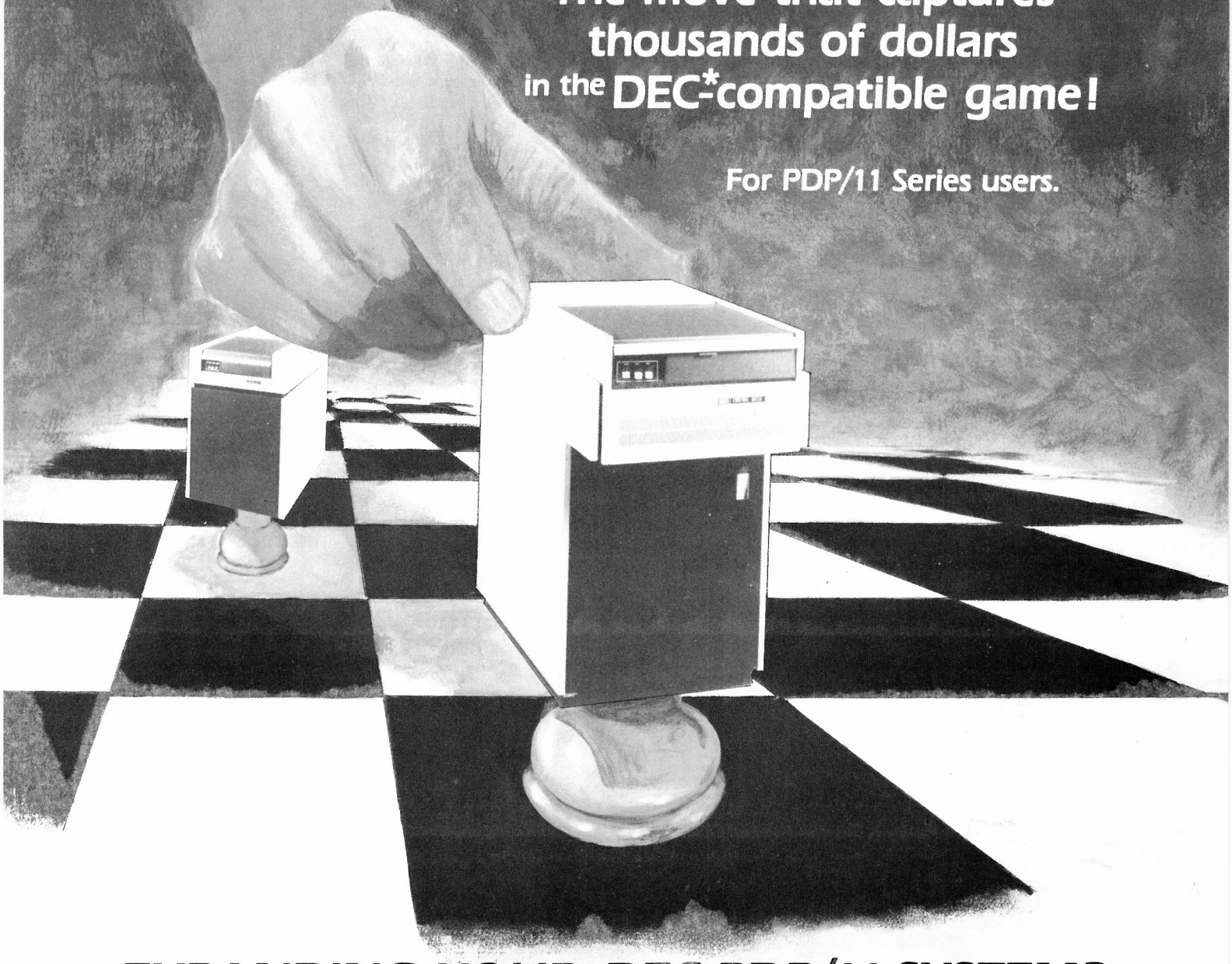
USING LIST PROCESSING TO LOCATE DISKETTE INDEX INFORMATION

It is not unusual to want to find information from a diskette index, and to avoid going into the index (where there is a danger of losing the document). The diskette index is set up as a LIST document, and can be used for many purposes. (It even can be alphabetized, or otherwise sorted, if care is used, by using the procedures set forth above, or with the SORT program available to WPS-8 users.)

DEKMATE!

The move that captures thousands of dollars in the DEC*-compatible game!

For PDP/11 Series users.



EXPANDING YOUR DEC PDP/11 SYSTEM?

For less than half the cost (see chart), our equivalent DEKMATE† expanded RP06 disc subsystem delivers:

- 50% more data storage,
- 50% increased data transfer rate, and
- 20% faster seek times!

When these features are combined with our short delivery time and the following DEC compatible factors:

- is software transparent to all DEC Operating Systems, Diagnostics, and Drivers,
- is fully media compatible (we use the same disc manufacturers as DEC), and
- is 11/70 Cachebus, Unibus, or Q-bus compatible

you have the winning cost/performance strategy in the DEC-compatible game!

The extra plus–we service what we sell!

Our large staff of DEC-experienced service engineers is always ready to install and maintain your DEC systems.

† DEKMATE subsystems are industry proven, high quality CDC, Memorex or Ampex disc drives and Emulex controllers. Also offered are high performance, reliable magnetic tape subsystems, DZ11's, DH11's, high speed MOS memories, printers, and other hardware. Call or write us for your hardware requirements.

* DEC, PDP/11 are registered trademarks of Digital Equipment Corporation.

COMPARE OUR PRICES

DEKMATE			DEC		
DM06	(300MB)	$19,950	$44,000	RJP06	(200MB)
DM06/70	(300MB)	$21,700	$44,000	RWP06	(200MB)
DM02	(80MB)	$12,950	$24,000	RJM02	(80MB)
DM03/70	(80MB)	$14,700	$25,000	RWM03	(80MB)
DM-300	(300MB)	$12,950	$34,000	RP06	(200MB)
DM-80	(80MB)	$ 7,950	$18,000	RM02	(80MB)
DM-77	(125IPS)	$10,995	$28,000	TJU77	(125IPS)
DM-45	(75IPS)	$ 9,995	$23,000	TJU45	(75IPS)

ABOVE DEKMATE systems are expanded or media compatible versions of standard DEC RP06/RM02 disc subsystems and TU77/TU45 magnetic tape subsystems. These subsystems are 100% compatible with all DEC software, diagnostics, and drivers. Expanded capability may require patches.

ADVANCED DIGITAL PRODUCTS

P.O. BOX 22657 • SAN DIEGO, CA. 92121
(714) 455-9150

The INDEX LIST for each diskette is formatted:

<n>title data <#>5<>

Using <n> for your selection specification, you can seek any type of sequence desired. E.g., if you want to find if "John Doe" appears, and on which document(s), the selection specification could be set:

if <n> =<*>John Doe<*>
then process record

The FORM would be set:

<n> <#>

Running the List Processing will show the presence of the requested data, and the document number, each time it appears. On long or several indexes this can save time, and is more positive than using the I command to examine every page of every index.

LINE NUMBERING USING WORD PROCESSING

Presently there is no easy way to number the lines on a document under WPS-8. Perhaps some day the powers to be will provide us with this feature, but for the time being it is necessary to use some planning in order to accomplish line numbering.

At the moment, the easiest way to number lines, whether starting with 1 and proceeding to nnnn, or repeating the same number of lines per page, is to do it by brute force.

Create your document in the normal manner, but allow sufficient extra space on the left margin ruler for the numbers to be used plus at least two spaces. Thus, if you ordinarily would use the left margin for your left ruler and expect to use three digits for the numbers, set your left ruler, initially, five spaces to the right. (NOTE: There will be a slight variance in this procedure for inside paragraphs. This is discussed below.)

Upon completion of the document, AND AFTER FINAL EDITING, the line numbers can be added by re-setting the left margin on the ruler to its normal location AND INSERTING A TAB AT THE FORMER LEFT MARGIN LOCATION. While this would ordinarily cause the text to "re-wrap", it will make no difference.

Proceed to the beginning of each line, using the BLUE LINE editor key.

Enter the line number and then TAB. Repeat this for each line to be numbered. Since the text already has been edited, the new line numbers will not affect your prior formatting, as you are using all of the extra space with the line numbers and tabs.

If you need to enter identical line numbers for each page (e.g., such as with legal pleadings of 1 through 28 or 32 for each page) then you can do this with a User Defined Key. Anything more, however, will use up all of the buffer space available for the User Defined Keys.

The use of the line numbers and tabs will not affect right justification, as each line number will follow a soft return. HOWEVER, SUBSEQUENT EDITING WILL BE VERY DIFFICULT. Therefore, try to avoid numbering the lines until the document is ready for final output.

INSIDE PARAGRAPHS

To use the line numbering feature on inside paragraphs, where the numbering is to remain on the left margin, use a W (wrap) in the ruler instead of the L for Left Margin. The first line of each inside paragraph will have to be double tabbed, but you will find it fairly easy to master after a few attempts. When you are ready to insert the line numbers, it will be necessary to remove the W from the ruler, and to replace it with a T (tab). When tabbing over from the number insertions, the text will remain formatted in the same location as with the W, and, as before, right justification will remain undisturbed.

If further editing may be expected, it may be easier to retain a copy of the document before line number inputting, especially where the editing may be extensive. The procedure indicated is not intended as a solution, but, rather, as a procedure which may make life somewhat easier for you.

INSERTION OF PORTIONS OF LONG DOCUMENTS, TOO LONG FOR "CUT AND PASTE", AND/OR WHERE ALL IMBEDDED MATERIAL IS DESIRED, AND/OR WHERE A "GO GET" ROUTINE IS NOT AVAILABLE BECAUSE THERE IS INSUFFICIENT ROOM REMAINING ON THE DISKETTE

It is not at all unusual to have the need to use a portion of a long document in a document presently being created. Quite often, also, the size of the required material exceeds the buffer space allowed with the "cut and paste" method (which often deletes a lot of the material you wanted); the remaining space on the diskette is insufficient to allow you to GO GET the old document, and then cut out the unwanted portions (even if all you want is in the first few pages) or you want to retain imbedded materials, such as rulers and page markers, and the cut and paste method won't retain them. Do not lose hope, there is a fairly simple remedy.

SOLUTION: Edit the old document to the portions desired. Enter a "boilerplate library" type of indicator at the beginning of the text to be copied, and a <> terminator. E.g.:

<<COPY1>>text material (may be as long as needed) <>

Use the same procedure for each section to be copied, but identify each portion with different names, e.g.: <<COPY1>>, <<COPY2>>, etc. (These identifiers can be removed, later, quite easily by using the blue <> key to advance through the document and rubbing out the identifiers.)

Note the drive and document number of the old document. Return to your new document and, with the Gold Menu (i.e., the editor menu) feature, change the boilerplate library to the drive and document number containing the old document.

Proceed to the portion of the new document which is to receive the old document's information, enter

GOLD LIBRARY and the name (e.g., COPY1, COPY2, etc.). The information will be transferred, including all imbedded materials, such as rulers.

After using this method, be certain to reset the boilerplate library, in the editor menu, for its proper location.

ABBREVIATION AND BOILERPLATE LIBRARIES

There is no end to the utility to which the system libraries may be appreciated by the Word Processing operator. Undoubtedly, these features are among the most important individual assets of the entire system.

Naturally, the needs of each user will be different. We believe that the following hints will be of interest to most users.

UPPER VS. LOWER CASE FOR FIELD IDENTIFIERS.

Again, as with List Processing, there is no requirement that you use lower case field identifiers for the libraries. In fact, upper case identifiers generally are much preferable, as reference to the library document may be made in upper or lower case and still retrieve the document, whereas if the library field identifier is in lower case, only a lower case identifier will retrieve it. This especially can be annoying if you are seeking an abbreviation library document (which does not echo the input on the screen) and you happen to have the caps lock activated.

LOCATION OF LIBRARIES

The Word Processing manuals and the self-paced teaching manuals for WPS-8 identify SYSTEM 2 and SYSTEM 3 as the location for the abbreviation and boilerplate libraries. Indeed, all the software for the Word Processing software comes with SYSTEM 2 and SYSTEM 3 initiated as the respective libraries.

There is no magic in the assignment of locations for the libraries and your own particular needs should dictate where these libraries are located, and even whether you might wish to change libraries during different operations (a very helpful and powerful feature).

In a client or job oriented operation, where each client or job is assigned an individual data diskette (or RL01 allocation) it might be most helpful to always have the boilerplate library as the first document to be created on that data diskette (which will always be document #2, as #1 is reserved for the diskette's index). If this is done, data which is repetitious for each client or job easily may be recalled by using the same abbreviations or identifiers for each diskette. For example, in our own operations we would identify the name and address block of our client with a field identifier of <<CLIENT>>. Since this information resides only on the diskette in use, every time the library identifier of CLIENT is used, the name and address of that client is displayed in the document. Naturally, the same is true with all information which applies to the specific account, but which is similarly identified for all accounts.

In this manner, the SYSTEM diskette's space is reserved for other needs, and many other libraries.

ALTERNATING LIBRARIES

There is no particular requirement that the library document always be in the same location. On the other hand, it often is helpful to be able to have several documents available on a given diskette which can be utilized as a library document for a particular purpose. This especially is helpful in creating new documents, where there is going to be repetitious use of some phrases. A new abbreviation library can be created, for these phrases only, and the phrases called with short entries. When completed, the library contents can be deleted (or retained, if desired) and the library document changed to the standard.

The use of such a "temporary" library especially is appreciated when one no longer has to search through the current document for specific phrases to be "cut and pasted" at a specific location.

Also, if a library document becomes too lengthy, then it takes a considerable period of time for the computer to find the phrases you need. To avoid this problem, you often can break your library documents into catagories, and, knowing the catagory desired, assign that document as the library (abbreviation or boiler plate) document for the current assignment.

USE OF THE HELP COMMAND FOR LIBRARY CONTENTS

As use of library documents increases it becomes increasingly difficult to remember field identifier assignments, and hard copy reminders become antiquated, misplaced, or unhandy. There is, however, an on-line solution, and that is a HELP COMMAND.

When creating a library document, the first field identifier should be <<HE>> for the abbreviation library and <<HELP>> for the boilerplate library. (Entering "help" will call the field in both cases, although the extra letters ("lp") will appear on the screen after an abbreviation library call.)

Prepare a Table of Contents which identifies each field identifier and its meaning, which can be called by the HELP command. As each new abbreviation is added to the library, the HELP section also is updated with the new command information.

To seek and examine the HELP information, which only can be accomplished while editing a document, the operator simply (1) enters the SELect key; (2) enters GOLD ABBREVIATION or GOLD LIBRARY and the word HELP (although only HE is required for an abbreviation); and the HELP information is displayed upon the screen. [Reference to "sub-help" libraries may be followed with another GOLD LIBRARY command.] After examining the displayed information, the operator (3) strikes the CUT key and all the displayed information is removed from the screen to the position where the SELect was inserted and the library may be accessed for the desired field.

By no means is the information provided here exhaustive of the potential for the HELP library. One may use HELP as a key to provide the operator

. . . continued on page 72

CONVERSION TO VAX ("native mode") BASIC

By Kenneth Ross, President, Ross Systems, Inc.

SUMMARY

A "proper" migration path to the VAX from RSTS has been discussed since the first VAX was announced some years ago. The RSTS community has screamed about compatibility (or the lack thereof), yet DEC has not really come to grips with the problem of an exact conversion path. DEC has developed VAX-BASIC which is remarkably similar to BASIC-PLUS II. Our firm, for reasons presented below, decided to acquire a VAX and to operate it in "native mode" to gain the benefits of a virtual machine. As of this writing (April 1, 1980) our 3MB VAX is due in June, and a representative sample of our software (all written in BP+/BP2) has been successfully converted to VAX-BASIC. The results are discussed below.

OUR BUSINESS ENVIRONMENT

Ross Systems is involved in management consulting, computer time-sharing and the sale of proprietary software products. We presently have a staff of 36 people and offices in Palo Alto, San Francisco, and Los Angeles. We have 3 PDP-11/70's with the VAX on order. We have 2 proprietary software products, both written in BASIC-PLUS II that make up the bulk of our timesharing users and that we offer for sale.

MAPS™ — Recognized worldwide as the leading product for financial modeling, reporting and consolidations. MAPS combines ease-of-use with the flexibility to handle large, sophisticated financial problems.

INTAC™ — A new concept for interactive data management, features an easy-to-use question and answer format combined with a unique report program generator and a screen formator/transaction program generator. INTAC can be used by business managers to create applications and by programmers to reduce programming time.

As our timesharing business grew, it became increasingly obvious that we had to be able to add bigger "chunks" of computer capacity than an 11/70. In addition, the processing requirements of our larger corporate clients operating financial modeling and consolidation systems needed the capacity of a "virtual machine". Some of our larger financial models are in the order of 700 rows by 40 columns and we felt they could be run more effectively on a VAX.

Clearly a VAX operating in compatibility mode or with an emulator would not solve our basic problem, so we began the process of converting enough of our software so that we could be sure of success.

OUR SOFTWARE STRUCTURE

When we began to work on RSTS machines, we made the decision to minimize the use of system dependent functions (SYS calls) and to write in BASIC that was as "vanilla" as possible. Essentially, all of our products are composed of multiple programs that CHAIN between themselves to perform a series of particular functions. We use a few rudimentary SYS calls e.g. get/put core common, and get a job number. The most RSTS dependent function contained in our software is the use of Pseudo keyboards for the automatic compilation and linking of generated programs. We make extensive use of virtual arrays both as file structure and as a method to pass information between programs. All of the programs are written in BASIC-PLUS II.

GENERAL CONVERSION COMMENTS

In general, the overall conversion was extremely easy. We were working with the first field-test version of VAX-BASIC, and we did encounter a few bugs, each of which were due to be fixed in later field test versions, and none of which were not solvable by some other method.

VAX-BASIC combines the ease of use of an interpreter with the speed power of a compiler. Programs can be developed quickly, interactively similar to BASIC-PLUS yet they can also include linkage to compiled modules both in BASIC and in other languages. Most of the RSTS type BASIC statements are available such as FIELD and CVT $% etc. Data file structures default to 8 byte floating point and 2 byte integers so that they are compatible with RSTS files. For INTAC, we developed our own, sophisticated file structures and we had no problems in transfering INTAC files from our RSTS machine to the VAX to be read by INTAC converted to VAX-BASIC.

On the VAX, large systems such as MAPS would tend to be linked together into one large program using the extensive subroutine facilities rather than CHAINing amongst multiple programs. The compiler/interpreter features allow this to be easily done.

SOME CONVERSION POINTS

1. CHAIN is supported but only without line numbers.
2. Core common is not supported.
3. All SYS calls must be changed to VAX equivalents (if available).
4. When a subroutine is called, variables are not reinitialized.
5. Virtual arrays must be opened with the VIRTUAL option.
6. Clustersize cannot be used on an OPEN.

TIMING

The following **CPU second** timings were done on the initial version of VAX-BASIC. It is our understanding that VAX-BASIC has been "tuned" for performance since that version.

TASK	CPU 11/70 BP	CPU 11/70 BP2	CPU VAX
1. Large Compile	21	159	22
2. Execute an INTAC program	NA	5.5	5.5
3. Virtual array processing	NA	190	356
4. FIELD statement	NA	31	35
5. PRINT 50 spaces on disk	NA	35	29
6. INPUT LINE	NA	49	28
7. INSTR	NA	57	28
8. Integer arrays	NA	27.9	9.6
9. MID	NA	52.3	15.2
10. MOVE FROM	NA	263.2	56.7

CONCLUSION

We believe that VAX-BASIC will offer us a high degree of compatibility to our RSTS systems, while providing us the performance increases that we require. For those of you who were involved with the release of BASIC-PLUS II in 1977 (?) and the horrible performance characteristics that if offered, VAX-BASIC seems to remedy most of those complaints and offer the solution that we originally wanted for the PDP-11.

RSTS Disk Optimization

By Mike Dash, John Fluke Mfg.

Probably every RSTS user, at one time or another, has tried to get a little more performance out of his/her already loaded-down system. Often, because time-sharing systems tend to be disk-bound, this means optimizing the structure of the disks.

Disk optimization is not easy, and it can require very careful analysis and study. However, version 7.0 has added some good tools, and makes it substantially easier to build and maintain well-built disks. As we explore the uses of these new tools, we will be able to extend the RSTS community's growing body of disk knowledge; in this way, we can all continue to help each other learn more and more about the best ways to use our systems.

Therefore, we are re-printing a DECUS symposium paper titled "Building a well-structured disk". The paper was written for RSTS 6C; this means that the material on analyzing a disk is still applicable, but the actual building methods and techniques do not reflect the new power of 7.0.

We hope that this reprint will stimulate your thought, experimentation, and contributions on how to build disks under 7.0.; it also expresses our thanks to DEC for responding to us, and our hopes that they will continue to expand RSTS's capabilities for analysis, control and structure of disks. In future issues of the **RSTS Professional**, we will publish a compilation of your methods, comments and suggestions on disk building.

BUILDING A WELL-STRUCTURED DISK

I. Introduction

A. Why bother?

RSTS systems are often disk-bound. Therefore, performance can often be improved when the disk structure is improved. The disk should be tailored to fit the requirements of your installation, and RSTS offers some of the necessary tools. (For an excellent discussion of disk internals, see Mike Mayfield's article in the RSTS newsletter, vol. 5, # 1.)

Is it really worth the trouble? Mayfield calculated that the worst-case file-open takes 5000 disk seeks! This is 4999 more than the best case; at 30 msec per seek (on an RM03), this is over two minutes of unnecessary disk bashing. As a less extreme example, consider an application with an average of 10 accesses per second on an RM03 disk. If the disk is unstructured, then each access has an average seek time of 30 msec. However, if the files are well-placed, each access could be as short a 6 msec; this is a total saving of 240 msec. Thus, each second has had 24% of waiting-time removed, and the overall system speed could be 24% greater.

B. What this paper is; what this paper is not.

This paper presents some of the issues to consider in building a disk. This includes analyzing your application, planning the build and performing it. This is not a cookbook or a specification of how your disk should look; each environment is different and requires a different optimization strategy for disks.

C. What is 'well-structured'?

The goal of disk optimization (for speed) is to minimize the number of head accesses and the distance moved on each access. This means that heavily-used files (such as swap files and directories) should be placed together, directories should be contiguous, and data files should be optimized for minimal directory overhead.

II. Planning

A. Choosing the pack characteristics.

A RSTS disk pack has a 'pack clustersize'. When a pack has a small clustersize, the directory overhead goes up (since a given file is then composed of many clusters). If the pack clustersize is large then disk space is wasted (there will be an average of one-half cluster wasted per file). If you have a lot of transient files (and can afford to waste some space), try using the next higher clustersize than the required minimum for your disk.

You can also set 'new files first' (NFF) as a pack characteristic. This means that directories will become more tangled (and the associated overhead will increase). If you work with a continually changing set of files, NFF is probably a good idea. If you have a fairly stable set of files — for example, a general ledger system — then you may not want NFF. In either case, use REORDR often to untangle the directories.

B. Finding the center of the disk.

The most heavily-used files should be in the 'center' of the disk. This minimizes the average distance that the disk heads have to move. Where is the center? Divide the number of

occupied blocks in half (the disk-allocation algorithm does a good job of filling the disk from the bottom up; this means that the used blocks are not spread evenly across the disk. For example, 2000 free blocks on an RK05 means that 2800 blocks are occupied; therefore the 'center' of the disk is at block 1400.) If you want to check, try using DSKDMP (from the UETP tape in the 6C kit) on SATT.SYS to see where the occupied blocks are on your disk.

C. Drawing the disk map.

Decide which files are most heavily-used on your system; these files should go in the center of the disk. Some of the obvious candidates are listed below, but each site will have differing requirements. Decide which files you access the most often and then draw a disk layout which has these files in the center.

SATT.SYS: accessed every time a block is allocated or released (when a file grows or is deleted)

swap files: accessed on every job swap (see SGM (System Generation Manual), p.3-30)

UFDs: accessed at login, logout, and on all OPENs, directory lookups, RUNs and file deletions

[1,2] library files: accessed implicitly by CCL commands, by login/logout, and by RUN commands

RTS files: accessed every time a non-resident RTS is swapped in

OVR, BUFF, ERR (See SGM, page 3-32)

any heavily-used task or data files at your site.

D. Avoiding cylinder boundaries.

For files which are used very often (such as swap files), you may want to ensure that the file does not cross a cylinder boundary. This just takes some study of the disk layout (see the Peripherals Handbook). Crossing a cylinder boundary requires a repositioning of the head; on an RM03 this takes 6 msec.

E. Planning the directories.

Directory planning is very worthwhile because blocks allocated to a directory will not be de-allocated (*). Therefore, a contiguous directory (pre-allocated) will stay contiguous. (Note, however, that even a well-built UFD should be REORDRed often.) Furthermore, the MFD is never touched by REORDR, so any initial order in the MFD will be preserved — if the MFD is ordered by account number, it makes SYSCAT listings very easy to read. Create an ACCT.SYS file with account numbers, ordered, in it.

Decide how big you want each directory to be. How many accounts will you have? This determines the MFD size. How many files (and how many clusters per file) will you have in each account? This determines the UFD sizes. After choosing the directory sizes, divide each size by 7 and round up to a power of 2; this is the clustersize that the directory should have. For more detailed information on directories, see Mayfield's article; for general guidelines, read SGM on DSKINT (page 3-7).

Finally, you may want to order the files in [0,1] and [1,1]. These accounts are never reordered and therefore the initial ordering will be preserved. This is only worthwhile if you have a lot of files in these accounts.

III. INIT

A. DSKINT

During DSKINT, set the pack clustersize, NFF (if you want it), the SATT.SYS location and the MFD clustersize. Do not let DSKINT create the [1,2] library account! Therefore, if you are creating a public pack, say NO when it asks 'create library account?'. Private packs are created properly, but system packs are a problem. Don't use the SYS option; create the pack PRI and build your own system disk (see below).

B. REFRESH

Create and place any of the following that you need: swap files, OVR, BUFF, ERR, CRASH. Create and place a dummy file (say, PLACE.UFD) to take up room and force the UFDs to occupy whatever part of the disk you have selected.

If you are building a system disk, then allocate and place INIT, the SIL and the BASIC run-time system. It is hard to build a structured disk if this is the first sysgen; generally this should be done after you have generated a running system. You will need to know the sizes of INIT, your SIL and the BASIC RSTS. (An additional complication during the first sysgen is that COPY places INIT.SYS too low on the disk to allow the MFD to be pre-extended.)

IV. Building the Directories

A. MFD

Run $REACT on your ACCT.SYS file and enable the accounts. Pre-extend the MFD by enabling dummy accounts until the MFD is full (this happens when the MFD occupies 7 clusters); then delete the dummy accounts. Ideally, this is all done from a running system on a different disk; if you have only one disk then you must run the program from a tape. The disadvantage of this method is that a temp file will be created on the disk you are using; the temp file may get in the way of the directories you want to create.

B. UFDs

UFDs are pre-extended in a similar way to the MFD. OPEN files in each account, until the UFD is full, and then delete the files. The files should be zero-length (name only) so that no other disk space is allocated while the directory is being built.

The easiest way to do this is to have a program read the ACCT.SYS file. On each account it should OPEN dummy files until the directory is full; then it should kill the files. When you're done with this, delete PLACE.UFD.

V. The System-Recovery Medium

A. Minimal RSTS systems.

If you are building a non-system disk then it should be saved at this point (see V. B, below). If you are building a system disk, you have nearly created a crash-recovery medium.

To make a crash-recovery medium, you must first finish building a minimal RSTS system. First put INIT.SYS, the SIL, ERR and BASIC.RTS into the files you have already created (this should be done with the /UP switch in PIPSML or PIP.SAV). Then make the pack bootable, using HOOK.SAV (see the RSTS/E software dispatch, article 16.1.2).

Boot the disk, INSTAL the monitor and tailor it. Install (and place) the BACKUP package (or whatever you use for file save/restore). It's a good idea to put UTILITY and UTILT1 on the disk, too. This is now a minimal RSTS system.

B. Building the recovery medium.

Make a copy of the disk. Use an image-mode copy utility (that is, ROLLIN or the SAVE/RESTORE facility that will be coming with version 7.0 of RSTS). If the disk you just copied was the system disk, then you have make a 'system recovery medium'. Whether or not the disk is a system disk, you should make and save a copy.

C. How to use the recovery medium.

If you have a catastrophic disk error, and lose the system disk, you now have a relatively simply recovery method. Copy the system recovery medium onto a pack; this gives you a minimal RSTS system on a well-structured disk. Then run BACKUP, read in the files from your archive tapes, and you have your system back on the air.

VI. Installing Files

A. Contiguous files.

Contiguous files have minimal directory overhead but cannot be extended. Task images (.BACs, .SAVs, .TSKs) are highly suitable for contiguous files. A patch in BACKUP could allow automatic contiguous creates for task images. . .

B. Clustersize optimization.

A file that fits in 7 clusters (or less) also has minimal directory overhead. The disadvantage of this approach is that the file will have, on the average, half a cluster wasted. The advantage is that the file can be extended (note that PIP will not automatically preserve clustersize on a file copy operation — you must explicitly do so in your commands or in your application programs).

You may have files that you want clustersize optimization for. This is hard to do with BACKUP. To kluge around this, you could make a BATCH job or indirect command file which copies the files with explicit clustersizes.

C. File placement.

It is bothersome to place files with RSTS 6C or earlier; use dummy files (created by REFRESH) to control file location. The rules of disk-space allocation are: first block of a file located as low as possible (starting at the bottom of the disk); successive blocks are allocated as low as possible (starting with the last block now in the file).

Starting with RSTS version 7.0, we will have explicit mechanisms for file placement.

(*) That is, directory blocks are not released when files are deleted. However, the DELETE function in REACT, and the /ZE switch in PIP will de-allocate directory blocks; don't use these functions. If you do, then the work you've done to structure the UFD will be lost.

```
1         EXTEND
2       !                                                              &
        !       CONTIG                                                 &
        !                                                              &
        !       UTILITY TO CREATE CONTIGUOUS FILES - RDM 1978          &
        !
5       INPUT 'TOP OF DIRECTORY <Y/N>:'; Y$
6       IF Y$ <> 'Y' THEN M% = 0% ELSE M% = 1536%
7       INPUT 'CLUSTERSIZE <256>: ';C%
8       C% = 256% IF C% = 0%
10      INPUT 'FILESIZE TO TRY FOR: ';F
11        GOTO 10 IF F = 0       &
        \ F = F - 65536. IF F > 32767.  &
        !

20        PRINT 'FILE NAME: ';          &
        \ INPUT LINE F$                 &
        \ F$ = CVT$$(F$,-1%)            &
        !
21      GOTO 20 IF F$ = ''
30      OPEN F$ FOR OUTPUT AS FILE 1%, CLUSTERSIZE C%, FILESIZE F, MODE 16%+M%
40      CLOSE 1%
50    END
```

LETTERS TO THE RSTS PRO . . .

. . . continued from page 6

Dear Sirs:

Regarding your "What is this?" on page 64, Vol. 2, #1. Could it be a Binary Clock? (Time 3:51:14.)

Keep articles like *RSTS Directories* [Vol. 1, No. 1, p.30 and Vol. 2, No. 1, p.45] in your plans for future issues. They're a big help in understanding RSTS.

Sincerely,
Steve Huth, Systems Manager
Computer Tech, Pittsburgh, PA.

Congratulations, Steve! Yours was not only the first correct answer, it was the first answer at all.

Scott Banks and Disk Directories are not in this issue but will be with us again in coming issues.

—

Sirs:

The item on page 64 is a clock which reads 3:51:14 as the time.

Jim Isaacs
Kentucky Machinery Inc., Louisville, Ky.

—

Dear Dave:

Congratulations on your new magazine. There has been a longtime need for a magazine like yours.

In regards to the picture on page 64 and what it is: It is a clock and it says 3:51:14. Probably PM as I would not wait around until almost 4:00 AM to take a picture of the darn thing.

What's the prize?

Yours very truly,
Kurt McRae, Vice President
Computers Unlimited, Inc., Billings, MT.

—

Dear RSTS PRO People:

The device on page 64 of the February-March issue of TRP is clearly a Binary Clock (BI-KRON), and it appears to say "3:51:14" though I see no indication of AM or PM.

It is too bad that I didn't get to see the issue until today, because I am sadly certain that I must be about the 55th person to give you the answer . . .

Cheers,
Jon Singer
Colorado Video, Inc., Boulder CO

Cheer up Jon, you were 4th!

—

Dear Mr. Mallery,

The second issue of the RSTS Professional came last week. It is a pleasure to read articles geared towards a specific audience of which I can proudly say I am one. Your fine magazine has been shuffled, routed, re-routed and it appears to have been dropped in the snow before making it to my desk. If wear and tear is indicative of readership then you'll do well.

The quiz picture on page 64 caught my interest. It appears to be a binary clock reading 3:51:14. If my guess is correct then I will pray that my prize is a free subscription. It would be nice to have a clean copy in the office. One last thing. My congratulations to you for having on your contributors staff Mr. Scott Banks. His articles on the internal directories have been informative and most importantly readable. Good luck to you all.

Sincerely,
Peter L. Hart, Coordinator
Administrative Computing
PSC, Plymouth, N.H.

Your prayers are indirectly answered. Your prize is not a free subscription, but our free T-shirt will be a reminder for you to order your subscription!

—

Gentlemen:

In answer to your question, the box of lights is a binary clock. The time shown is 3 hours 51 minutes and 14 seconds.

Sincerely,
Neil V. Sibley
Senior Programmer/Analyst
The Federal Land Bank of Spokane
Spokane, WA

—

RSTS Professional
Re: p.64, Vol. 2, Num. 1
It's a clock:
3:51:14

You didn't mention whether one had to be the first to answer.

Mark Emerson, Analyst/Programmer
SPU Computer Services, Seattle, WA
P.S. Tim's Wrong!

—

RSTS Professional
Re: p.64, Vol. 2, Num. 1
It's a clock:
3:58:42.

You didn't mention whether one had to be the first to answer.

Tim Rue, Operations Manager
SPU Computer Services, Seattle, WA
P.S. Mark's wrong!

Mark, you're right, Tim's wrong. Tim you're wrong, Mark's right.
P.S. You're both right in that we didn't mention whether one had to be the first to answer. Mark A. knows what we did say.

—

Gentlemen:

The object on [op.cit.] is plainly a binary clock displaying the time 3 hours, 51 minutes, and 14 seconds. Send prize to: Perry Locke, Hughes Aircraft Co., . . ., Fullerton, CA

[sic]P.S. RP articles are fair but the cartoons lack both humor and professionalism.

Sincerely,
Perry Locke

Dear Perry, Mark and Tim are right, you're wrong! The P.S. goes after the closing.

"RSTS Prof."
Page 64, FEB/MAR 80.
3:51:14.

C.H. Haring, President
Computer Hardware
Maintenance Co., Inc.
Newtown, PA

Dear Mr. President:
We heard what was said, but what's it all about!

—

Dear Sirs,

I noted with great interest your "contest". I have therefore determined that the object in question is a BCD clock reading 12:58:52. Since you specified that ANYONE who can guess what the object is will receive a prize, I eagerly await your response.

Sincerely,
Mark Anacker
SPU Computer Services, Seattle, WA
P.S. Both Mark and Tim are wrong.

Dear Mark A.,
Re: Mark E. and Tim, you're both right and wrong.
Re: Perry, you're right.
Re: Your free prize, you're wrong!

—

Dear Mr. Marbach,

I gather from the name on the mystery panel that the device is a binary chronograph (Bikron). If this is the case, I suspect that it says "12:58:42".

When it comes to clocks, I prefer the KW11-L.

Sincerely,
Rick Richmond, Systems Officer
Pikes Peak Library District
Colorado Springs, CO
P.S. Many thanks for your fine magazine.

Dear Rick, your time is in the wrong zone, but at least your P.S. is in the right place.

—

Dear Sir:

The answer to the question is: 1) it's a Binary Clock, and 2) it says 3:51:14.

I will be anxiously awaiting my prize.

Thank you,
Jim Corkey
Spokane, WA

—

This is a binary clock with the time of 3:51:14.

William E. Elstermeyer
SLT Warehouse Co., St. Louis, MO

We received this one on May 17. That's late enough. We'll have to say that William is the last winner.

—

Now that we've ended that contest, perhaps the folks at SPU Computer Services, Seattle Pacific University, can get back to work!

Send letters to: Letters to the RSTS Pro, P.O. Box 361, Fort Washington, PA 19034.

PLEASE IGNORE THIS NOTICE

By Joel Schwartz, M.D.

EVERYBODY loves to take a chance. Not you? You're reading this article aren't you?! Freud once said, "Every human being is inherently born with the organic and physiologic components necessary for the formation of a complicated internal system which, under the proper conditions, can lead them to situations involving chance." Unfortunately, no one understood what he meant and the elucidation of why individuals involve themselves in situations of chance continued to be mystery. However, this article will shed no further light on that mystery. It will deal with six games of chance found in the computer.

Russian Roulette, the first and simplest, takes one minute to play and twenty seconds to forget. The directions are easy. Here is a revolver. Hit 1 to spin chamber and pull trigger, hit 2 to give up. There's not much more to it and if I tell you what happens, you won't get fooled into playing it like I did, so I'll go right on to the next game.

Roulette. In order to play this game, you need a six-week intensive course to learn to understand the directions. There are fifty possible deaths and I'll bet you'll give up before you play.

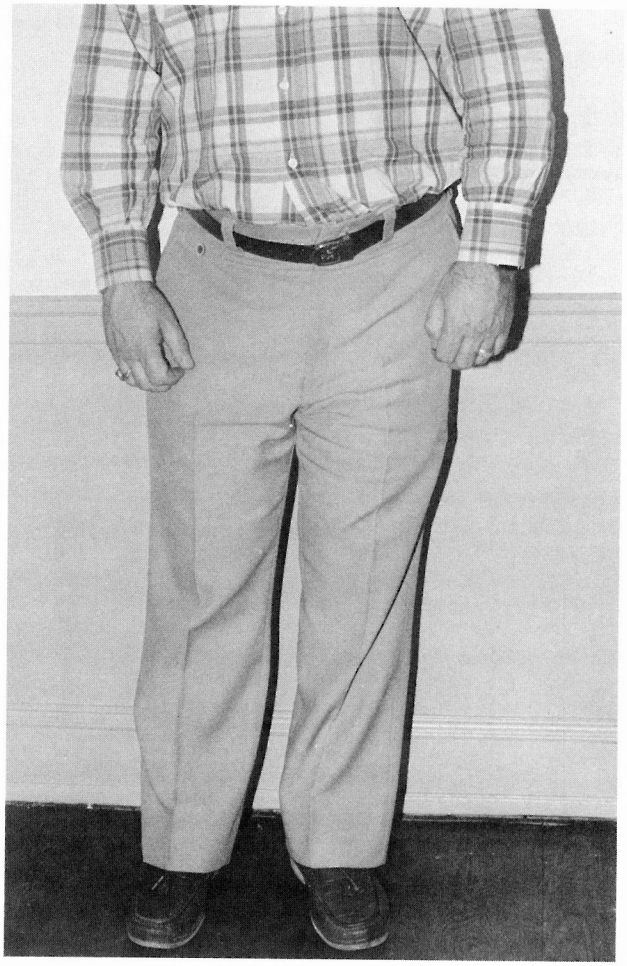
Moving right along, there's **Black Jack.** It is just like the real game, including doubling down, splitting pairs and endurance. It was so real, in fact, that after fifteen minutes I had lost $1,000.

The fourth game I played was **Craps.** Sometimes the computer rolled the dice and you bet on the times it was vise versa. I got bored of the game after about ten minutes, but if you like Craps, I guess you'll like this.

The fifth game was **Keno.** Keno is a game where the player chooses eight different numbers, from 1 to 80 inclusive, and bets $1.20 on each. The computer then selects 20 numbers and compares them with yours and you win for each match. The game would have been terrific if the programmer had not been fixated on ringing the bells. Unfortunately, as the computer searches out the 20 numbers and then matches them with yours, the bell rings incessantly, forcing you to pop Excedrin, salivate intensively or reach for Control C.

The last game was the one I liked the best. **Horses** is a racing game involving a horse. The odds are posted at the beginning of the race and after you place your bet you can sit back and watch your horse fall farther behind. I played 10 times and lost $100. The price of admission was right and there was no fighting traffic to get on the expressway to go home.

In conclusion, I have a letter here from a lady from Madison, Wisconsin who writes,



Dear Dr. Schwartz:

Would you please show us a picture of you without your mask on?

Sincerely,
Eleanor Schwartz
Madison, Wisconsin

MUMPS as a Language

By Peter Clark

MUMPS which stands for Massachusetts General Hospital Utility Multi Programming System was developed as a tool for handling clinical and laboratory data in a time sharing environment. The language (first developed in the mid 60's) although developed in the medical community, is a general programming language which is a valuable tool in many fields. However, being developed at MGH, it has gotten wide acceptance in the medical area.

MUMPS has several features which make it very useful as a language for developing data base systems.

1. Powerful string handling and pattern matching
2. Simple access of data in data base
3. Several types of indirection
4. Interpreter

String Capability

There are 5 string operators used in the language.

1. "_" Concatenation i.e. A_B will concatenate string B to string A.
2. "=" String equality.
3. "[" String follows, i.e. A["XYZ" returns "TRUE" if string A contains the string "XYZ".
4. "]" String follows, i.e. A]B returns "TRUE" if the string A follows (alphabetically) the string B.
5. "?" Pattern match. This operator allows checking for patterns of upper case, lower case, numeric, etc. characters in a string.
   Example: A?3N1"—"2N1"—"4N returns "TRUE" if string A is in NNN—NN—NNNN format.

The following string functions are also part of the language:

1. $EXTRACT Extract substring from a string. The beginning and/or ending positions of substring within main string are specified.
2. $FIND Returns end position + one of a substring within a string.
3. $JUSTIFY Returns a value of an expression right justified within a field. Used for formatting numeric values.
4. $PIECE Selects a substring from a string between specified delimiters.

Data Base Access

Variables in a data base are accessed in the same way as other variables except the variable name is prefixed by an " ↑ ".

Examples:

```
SET ↑ A=1                  ;sets "global" variable A = 1.
SET ↑ A(1,2,3) ="XYZ"      ;sets "global" variable A(1,2,3) = to
                           ;string "XYZ"
SET B= ↑A                  ;sets "local" variable B = "global"
                           ;variable A.
```

Global (and local) variables may have any number of subscripts. There are functions in the language for returning next subscript at any level, information about any node (i.e., does it contain data, pointers, etc.). MUMPS arrays are sparse arrays, i.e., every element of the array contains a pointer. This allows one to have a very wide range of subscripts without preallocation of a large amount of space as required by other languages (DIM in BASIC).

The multiple levels of subscripts allow the data base designer to minimize disk accesses for any node in the data base.

Indirection

MUMPS allows the programmer to use several types of indirection.

1. Argument indirection
   Example: the sequence
   ```
   SET X="A=2+Y",Y=2
   SET @X
   ```
   will set the variable A=4.
2. Name indirection
   Example: the sequence
   ```
   SET X="B"
   SET @X=2
   ```
   will set the variable B = 2.
3. The EXECUTE command will allow you to interpret an entire command line.
   Example:
   ```
   SET X="SET Z=1+2"
   XECUTE X
   ```
   will set the variable Z = 3.

These indirection capabilities are very powerful tools for developing general inquiry programs, computer assisted instruction applications, etc. The fact that MUMPS is an interpreter makes this type of flexibility easy to provide in a language.

Interpreter

Since MUMPS is an interpreter, program development is much easier and faster than with a compiler. The developer doesn't have to wait for lengthy compilations and linking. Using breakpoints and examining variables during execution is easy to provide for the programmer. The indirection features mentioned above are also provided easily because MUMPS is an interpreter.

MUMPS allows the programmer to abbreviate all commands to a single letter. You can have multiple commands per line and most commands allow multiple arguments. A summary of the STANDARD MUMPS commands follows:

BREAK	Stop execution (i.e., breakpoint)
CLOSE	Close file, release device
DO	Begin execution at specified address. Return when QUIT executed. Can call routines from an external device.
ELSE	Execute rest of command line if $TEST is true. $TEST is set to true or false by the IF command or postconditionals (see below).
FOR	Specify repeated execution of commands following in the same line.
GOTO	Transfer control to specified address.
HALT	Suspend execution.
HANG	Suspend execution for specified number of seconds.
IF	Execute rest of command line if "IF" expression is true.
KILL	Remove specified variables from internal symbol table or global area.
OPEN	Obtain ownership of device.
QUIT	Exit a DO,FOR or EXECUTE command.
READ	Read data from current device.
SET	Assign value to specified variable.
USE	Designate a specific device to be used for "current" device.
WRITE	Output data and/or format control to "current" device.
XECUTE	Interpret MUMPS code during execution.

Most commands can be post-conditionalized to be executed only if the post-conditional expresion has a value of true.

EX:

SET:Y=1 X=1

This will assign the value 1 to X if Y has a value of 1. Otherwise the command will be ignored.

In the next issue I will discuss an implementation of Standard MUMPS on a DEC SYSTEM—10. Unfortunately MUMPS is only available on DEC 11's in a single language environment. There have been rumors from DEC that they would develop MUMPS to run under various 11 operating systems but I doubt that will happen now that there is such an implementation by DEC for the VAX.

What is going on here?

A prize if you can tell us exactly!

DECUS-PLUS

or, Independence for the RSTS Community

By Howie Brown and Monica Collins*

Here in RI, nearby Mass and Conn, an exciting experiment is underway — the building of a viable RSTS users group INDEPENDENT of DEC's official user structure, DECUS. SENERUG — the Southeastern New England RSTS Users Group — reflects the belief that a group of users can best serve its members by: 1) aggressive recruitment and active participation of user members; and 2) avoiding a paternalistic relationship with DEC.

Though apparently new to the RSTS community, independent user organization is an accepted fact of life elsewhere in the data processing world. System 32 users, for example, maintain the SHARE users group independently of IBM. Similarly, the HP General Systems Users Group specifically excludes Hewlett-Packard financial support and discourages participation by Hewlett-Packard employees.

What, you may ask, is wrong with the DECUS arrangement? In short, reliance on DEC's financial support and goodwill. Ask a DECUS officer why vendors of compatible hardware and software don't get invited to set up displays at DECUS symposia. Chances are, the answer will run like this: Whatever benefit might accrue would be outweighed by the potential jeopardy to the "special relationship" between DEC and DECUS.

There is a growing feeling here that the "special relationship" does all of us, RSTS users and DEC alike, more harm than good. When RSTS users come together the conversation inevitably turns to what's wrong with DEC, and the complaints seem to have a common basis: Digital's lack of response to its own market. Certainly DEC's sales position could tempt any corporation to take its market for granted. But (although comparisons are admittedly dangerous) one should point out that in spite of its dominant market position, giant IBM is unarguably market-driven.

Here is the root of the matter: what is wrong with DEC-supported user groups is that they REINFORCE DEC's unresponsiveness by discouraging user exposure to hardware and software not labeled DIGITAL. If DECUS organizes true "user" symposia, why are displays of compatible hardware and software excluded? And why are uncomfortable issues raised by users so often tabled into oblivion?

When the charter members of our organization first met 6 months ago to draw up the by-laws, the question of DECUS membership was addressed, and the overwhelming majority felt that they would be better served by an independent users group. SENERUG members are excited by the prospect of giving equal time to all hardware and software vendors of interest — DEC included, but not DEC alone. In the past, many of us had needs that DEC failed to meet adequately. For example, the long-awaited replacement for the discontinued VT52 terminal with printer controller. Explaining our needs to other vendors will result in a more responsive market for all of us.

But isn't DECUS support crucial to a local users group? The success of SENERUG has already answered that question with a solid "NO". If you're willing to invest some energy and time in organizing, you can prosper without DECUS. We identified more than 20 installations in our geographical area, generated publicity, and now additional sites are making themselves known to us. With a supportive membership and an enthusiastic board, we've organized monthly technical workshops, including guest speakers, and maintain a software library. The group does its own mailings, surveys, and record-keeping funded by modest annual dues.

While DECUS membership was voted out of our by-laws, cooperation with DEC is considered crucial by all our members. One of our earliest accomplishments was the establishment of a line of communication with DEC. When the situation warrants their presence, DEC representatives attend our meetings at our invitation. We can claim to have solved specific support problems for our members through such meetings. But, by having an organization able to function without ongoing support from Digital, we avoid worrying about earning DEC's displeasure at the pressure we might exert on DEC's support services, or at the attendance of "foreign" vendors at our meetings from time to time.

DECUS does serve a legitimate purpose as a disseminator of information between DEC and RSTS users. An independent user structure can serve another purpose, that of protecting and advancing the interests of users without regard to special relationships. So far SENERUG's experience has shown that an independent RSTS users group is viable, and we believe that the idea can be applied elsewhere. If we can supply you with further details, contact us — SENERUG, P.O. Box 3043, Pawtucket, R.I. 02861.

---

* Howie Brown is Chairman of SENERUG and Systems Programmer for Information Systems, Inc. of Pawtucket, R.I.

Monica Collins is Vice-Chairman and Program Coordinator of SENERUG and DP Manager for George Mann Co., Providence, R.I.

? Why TECO ?

By Carl Marbach, Editor, RSTS Professional

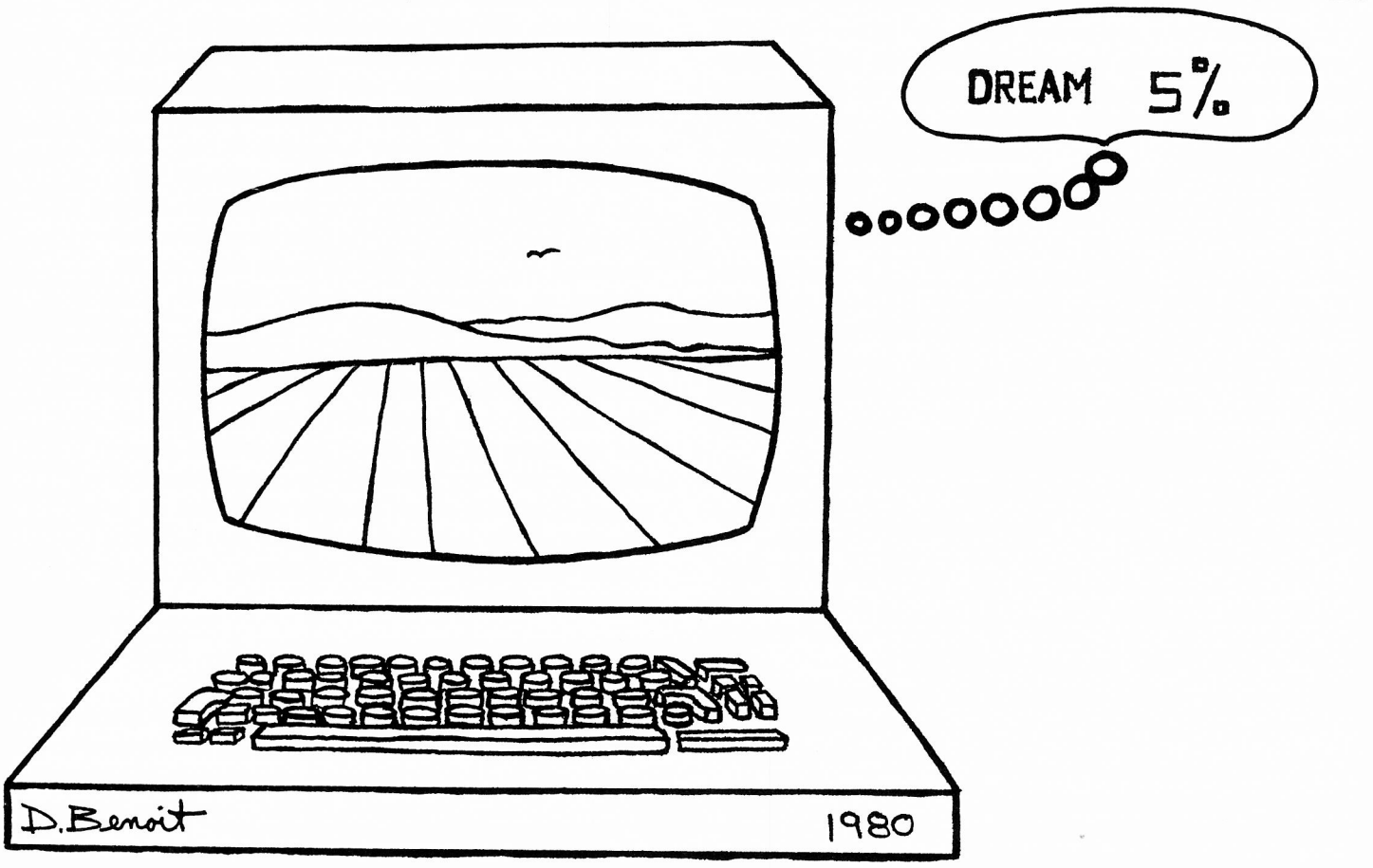
IN the last article on **How TECO**, Martin Pring noted that he tries to use Q-registers mnemonically by alluding to the fact that he keeps his limericks in Q-register L. We know that he keeps them in his head but we did manage to find a person who has written on structured TECO. Because of the many sessions that run concurrently at DECUS we didn't have a chance to hear her talk about it in person, but we did read about it in the Proceedings published about 6 months after the original talk (they use a computer no doubt). Jacquie has updated her original article and you'll find it in the **? How TECO ?** section which follows this.

How is it possible, then, that we can talk about "structured TECO" if TECO is not a language. Let's do away with any arguments and set down this author's premise that:

> TECO is a language for manipulation of text. It has a powerful "immediate" mode which allows on line editing of text. The language is interpretive on my PDP-11 but compiles on the PDP-10 I occasionally use.

Why do we need (or want) another language? I tell most of my programmers and other interested parties, that BASIC PLUS is much like FORTRAN, but more flexible and easier; similar to DIBOL and COBOL although the syntax and structure are different; and that when you know one well, the others can be mastered in a short time. Most of us, in fact, have migrated from some other language to BASIC PLUS (or others) proving this theory correct. There are some languages that are UNLIKE the ones mentioned above. Their very nature is different and they require different thinking modes and programming styles. SNOBOL (developed by Bell labs or thereabouts) is one of these. I once took a programming language course where we studied Assembly for the 360, ALGOL, BASIC, FORTRAN, COBOL, MACRO, FOCAL, JOSS (any old PDP-6'ers out there??), and .....SNOBOL. They were all the same, different syntax, but all the same except for SNOBOL. WOW! Get a book and see for yourself; things you NEVER thought about. Well, in some respects TECO is like that. The concept of an "anchored search" is lost on a BASIC PLUS or COBOL programmer. Matches, wildcard matches are cumbersome and unwieldy in most languages but easy and facile in TECO. Try writing SQUISH, the TECO macro for removing comments, spaces, redundent ESC's, and intelligibility from TECO macros, in any other language and call me next year.

If I can make a plea for TECO, it would be for it as a language, not an editor.



? How TECO ?
STRUCTURED PROGRAMMING IN TECO

By Jacquie Stafsudd, Hughes Research Laboratories, Malibu, California

Abstract

The concise nature of TECO's syntax necessitates the formalization of a "programming style" when writing complex TECO programs. This paper presents the basic ideas of Structured Programming principles, and suggests standards to help in implementing, debugging, and maintaining TECO programs.

INTRODUCTION

The use of TECO by programmers ranges from a nominal text editing utility to a sophisticated language in which complex editing functions may be written. These functions can be considered as "subroutines" which can be stored, later merged with other editing functions as part of a larger program, and driven by a mainline command string. The TECO "subroutines" are referred to as "macros", in that they are sequences of commands used to perform specified functions.

As a programmer attempts to create larger and more complex TECO macros, he is soon met with the obscurity that the concise syntax of TECO provides. The very nature of TECO's succinct command structure, where almost every ASCII character can represent a command, acts as a barrier to readability, understanding, debugging, or error free modification of all but the simplest macros by all but the most advanced TECO programmer.

Therefore, the utmost care and forethought must go into the writing of TECO macros to minimize programming errors and maximize program utility. Not only should these "subroutines" be thoroughly documented internally, but their structure should be as straightforward and understandable as possible.

The philosophy of Structured Programming as originated by E. W. Dijkstra meets the need of the TECO programmer. This paper attempts to provide an understanding of the principles of Structured Programming and apply those principles to the programming structures native to the TECO language. In particular, this paper focuses on the TECO-11 syntax and does not attempt to cover specifically all versions of TECO, although many of the Structured Programming techniques will still apply.

STRUCTURED PROGRAMMING PRINCIPLES

The goal of Structured Programming is to organize and discipline the program design and coding process in order to prevent most logic errors, make programs easily understood, and permit error free modification and maintenance. Structured Programming has three major characteristics:

1) Top down design
2) Modular programming
3) Structured coding

Top Down Design

Top down program design starts with a clear and precise statement of the problem and a determination of the major tasks involved. Then each of the major tasks are, in turn, subdivided into small modules until each module represents a distinct function that can be easily comprehended.

Next, the data structures must be defined and the major processes to which the data will be subjected must be described. And finally, the program should be documented while still in the design phase in order to further clarify the processing and logic flow.

Modular Programming

If the top down design of the problem has been properly done, then the task will have been partitioned into subtasks that can represent logical functions. This structuring is intended to:

1) insure that the actions of each module are well specified.
2) minimize errors by limiting the complexity of the particular function being coded.
3) isolate functions from each other so that the effects of any change or refinement to that function will be localized to that particular module.

The coding of a module should be such that it has one entry point at the top and one exit at the bottom. Within each module, there should be a minimum of paths to keep the structure simple. Following these guidelines will insure a minimum of complexity, a smooth flow of logic, and a maximum of module independence — hopefully leading to error free programming.

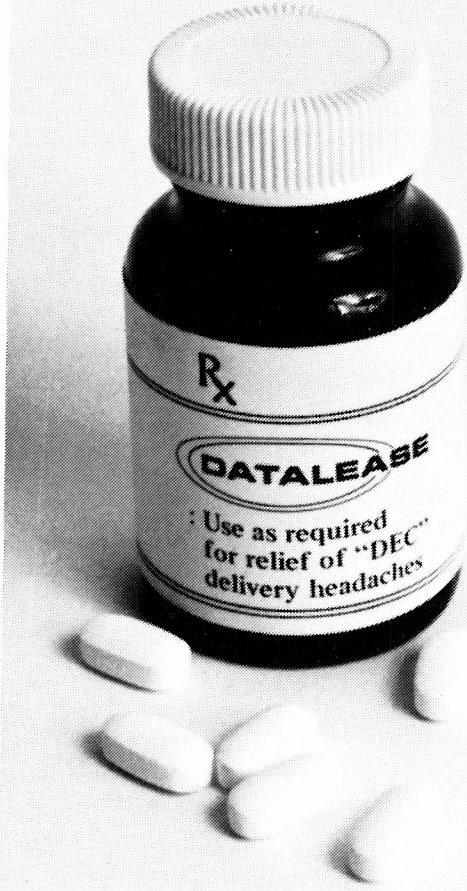
Structured Coding

Structured coding is a method of writing programs with a high degree of structure. It is based on some simple logic structures from which a "proper program" can be formed. A "proper program" is one with one entry point, one exit, and no infinite loops or unreachable code. The basic structures needed to write a "proper program" are:

1) Sequence — the idea that program statements are executed

PDP* 11 & 11/70 Disk Capacity Headaches

For Fast Relief . . . DMS Disk Sub-Systems Are The Best Prescription



COMPARE PRICE - PERFORMANCE

	RWP 06	DM 70 / 300
DRIVE CAPACITY (Mbytes)	174.4	253.7 (45% LARGER)
DATA RATE (K / words / sec)	337.9	491.5 (45% FASTER)
PRICE	$44,000.	$23,900. (45% LOWER)
	RJM 02	**DM 11 / 80**
DRIVE CAPACITY (Mbytes)	67	67 (EQUAL)
DATA RATE (K / words / sec)	337.9	491.5 (45% FASTER)
PRICE	$24,000.	$13,350. (45% LOWER)

COMPARE DELIVERY

DATALEASE MEMORY SYSTEMS can deliver and have your disk sub-system on-line in less time than it takes DEC* to process an order. (Typically within 30 days)

COMPATIBILITY

DATALEASE MEMORY SYSTEMS offers disk sub-systems that emulate the RM02, RM03, RP04, RP05, and RP06. The sub-systems are completely software transparent and best of all will run standard diagnostics.

NATIONWIDE SERVICE & SUPPORT

DATALEASE MEMORY SYSTEMS provides installation and maintenance nationwide through the Engineering Services Division of CDC. Our own highly experienced systems analysts supervise every installation and do not step out of the picture until you are completely satisfied.

Call toll free 800-854-0350

DATALEASE

DATALEASE MEMORY SYSTEMS
2770 East Regal Park Drive
Anaheim, California 92806
In California (714) 632-6986

***PDP and DEC are registered trademarks of Digital Equipment Corporation**

How to count your chickens before they hatch.

Surprises can be expensive. Even good news can cost money if you're not prepared for it.

Financial modeling lets you avoid surprises and allows you to plan calmly for whatever the future has in store.

FINAR is the financial planning system designed for businessmen in companies of all sizes.
You don't have to be a computer wizard.
You don't have to be a financial analyst.
You don't have to have a huge computer.

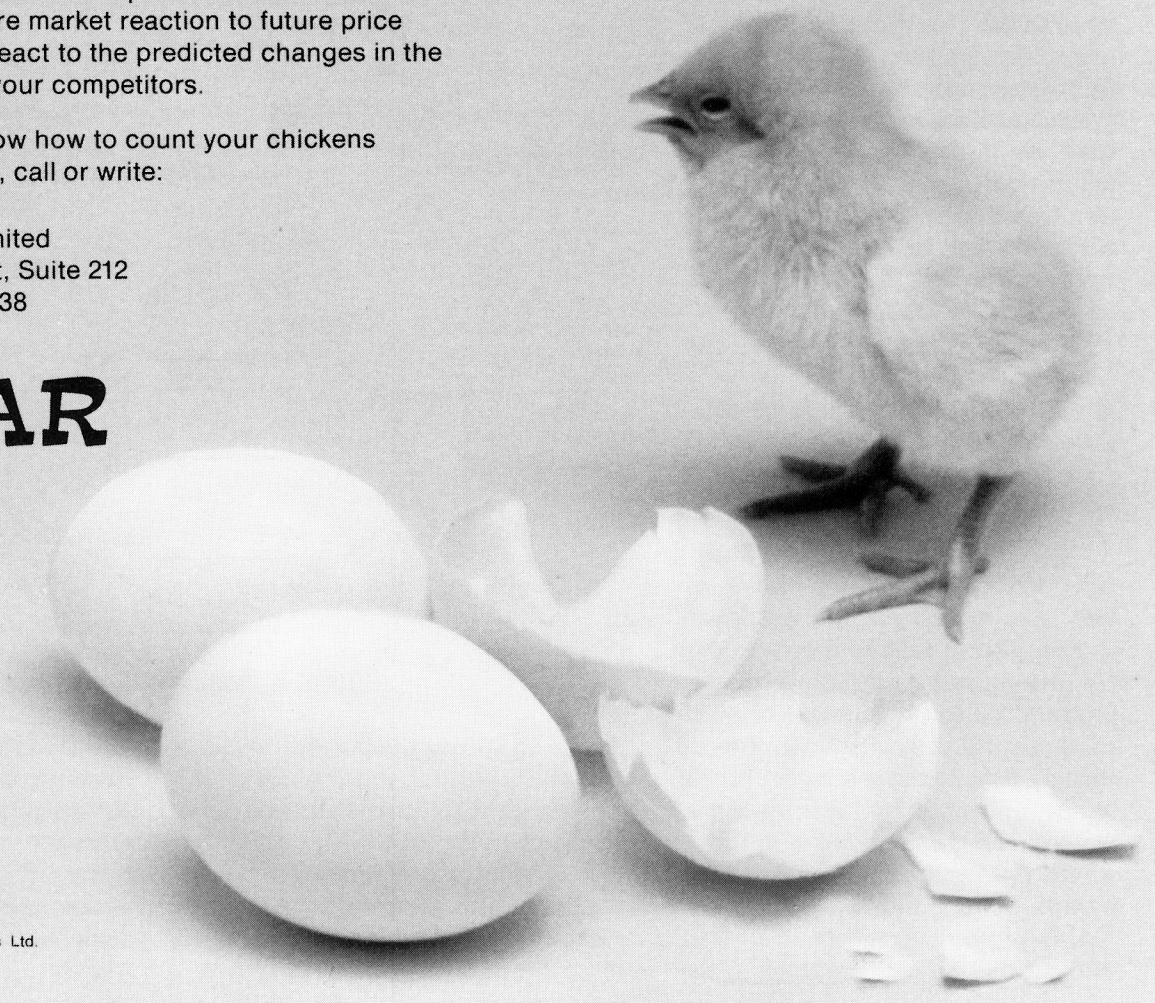
All you need is your own business know-how, a DEC PDP-11 RSTS computer and FINAR.

FINAR will help you to anticipate variations from budget, to measure market reaction to future price changes, and to react to the predicted changes in the economy before your competitors.

If you'd like to know how to count your chickens before they hatch, call or write:

Tony Kobine
Finar Systems Limited
132 Nassau Street, Suite 212
New York, NY 10038
(212) 222-2784

FINAR



in the order that they appear in the program unless something is done to change the sequence.

2) Selection — (called IF THEN ELSE) the choice of two actions based on some conditional.

3) Iteration — (called DO WHILE) used to execute a set of code repeatedly while a condition holds true.

Schematically, these logical structures are shown in Figure 1.

In general, although it is possible to write well-structured programs using only these three basic logic structures, programming becomes clearer if the repertoire is expanded to include the REPEAT UNTIL and CASE structures as shown in Figure 2.

The REPEAT UNTIL structure provides a type of looping similar to the DO WHILE but differs in that the conditional is checked after the statements are executed, and the loop terminates when the condition proves true. The CASE statement is a multi-branch construction used to select one of a set of functions for execution depending on the value of an integer expression.

To help identify these logical structures within a particular language, the use of "paragraphing" or "indentation" techniques should be considered. Paragraphing refers to the positioning of code on a page in an "indented" manner so as to best represent and identify the logic being used. To apply these principals to the TECO language, we must consider the syntax that TECO provides.

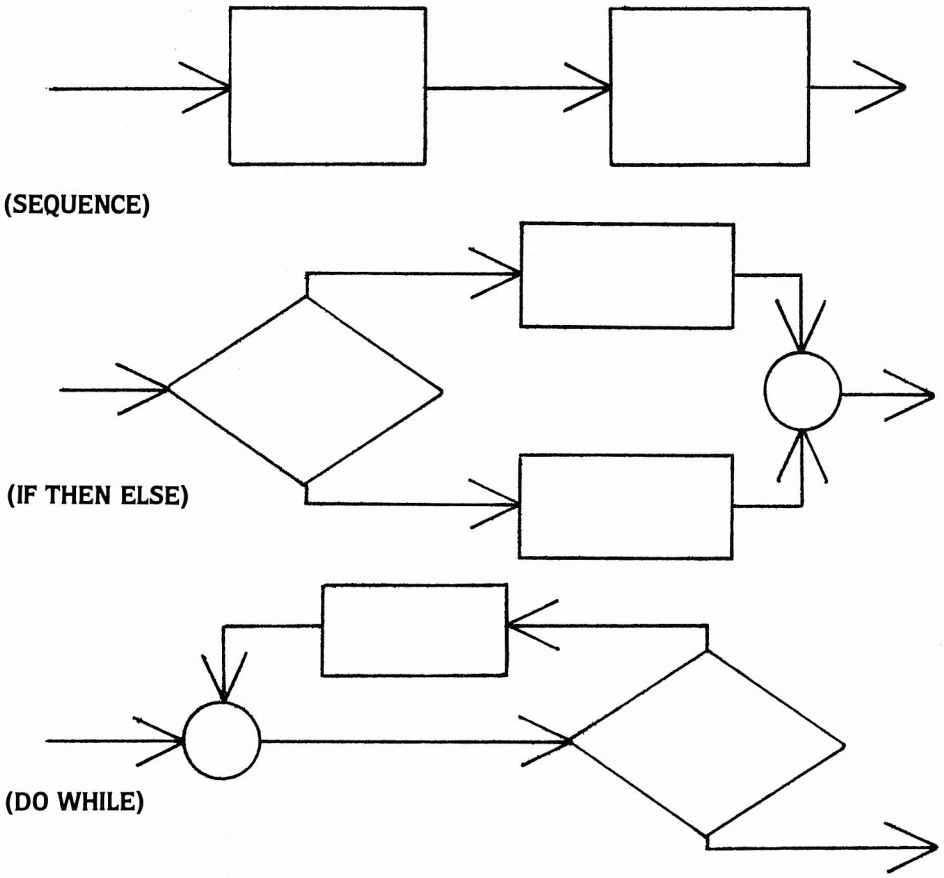


FIGURE 1. Basic Logic Structures

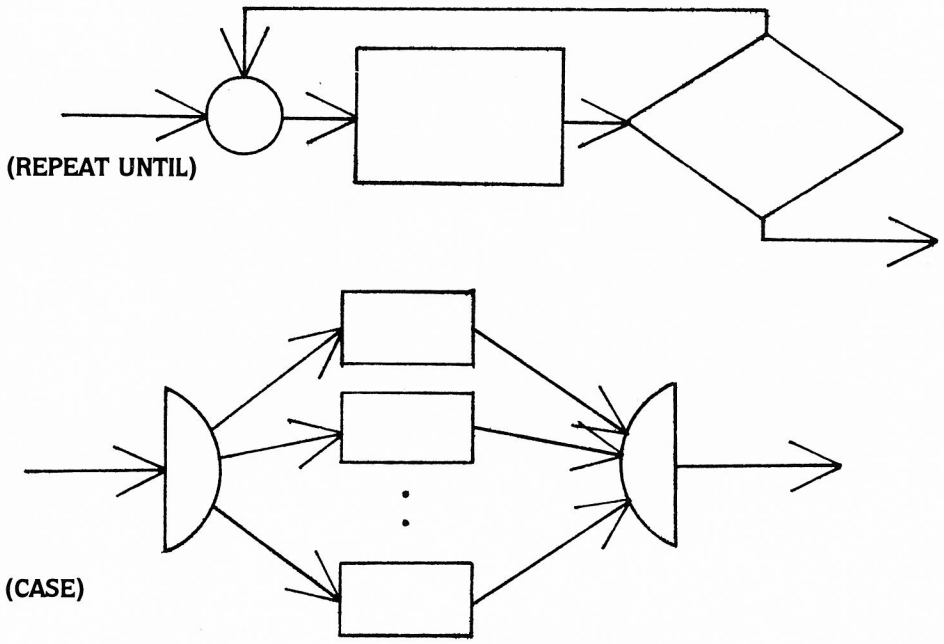


FIGURE 2. Additional Logic Structures

APPLYING STRUCTURED PROGRAMMING TO TECO

Preliminary Thought and Design

Much of the discipline of Structured Programming takes place outside the context of the particular language being used to program. The top down design specification of the problem should be done before even calling up TECO. The amount of forethought put into this preliminary phase will pay off immediately when coding is started. This is especially true in TECO, where the identification of variables, macros, buffers, and text is limited to the Q-register identifiers A-Z, 0-9. Without the ability to use longer mnemonics to specify the purpose or content of a Q-register, the significance of a particular TECO command can be easily lost or forgotten.

Therefore, once the top down design has been completed, the assignment of the data areas, buffers, and macros should be made according to the Q-registers they will reside in. Hopefully, this will be done according to some logical process — if only to store numbers in Q-registers 0-9, and buffers, text, or macros in Q-registers whose labels begin with a character somehow representative of their function.

TECO Logic Structures

The logic structures shown in Figures 1 and 2 may be found directly within the syntax of the language, or may be constructed by using more than one TECO command.

SEQUENCE — TECO command syntax follows the SEQUENCE principal in that the commands are executed in order of appearance unless a command is given to branch, loop, or test for a condition.

IF THEN ELSE — this structure is given directly in the TECO-11 syntax. It can be shown in an "indented" manner as follows:

```
n"X
  [THEN-COMMAND]
 ¦
  [ELSE-COMMAND]
 '
```

A simple example using this structure is:

```
(OA)"V
      (OA-32)I$ D
     ¦
      C
     '
```

This command sequence will replace a lower case alphabetic character (if it is immediately to the right of the buffer pointer) with its upper case counterpart. The command is translated by breaking it down into its component parts as follows:

(OA) returns the ASCII code of the next character in the buffer

"V tests IF the value is an ASCII code for a lower case alphabetic (a-z)

(OA-32)I $ D is the THEN command which inserts the upper case equivalent character in the buffer and deletes the lower case character

¦ is the beginning of the ELSE command

C moves the pointer past any non-lower case characters

' marks END-IF

DO WHILE — this structure can be set up using the iteration loop with a conditional LOOP-TEST at the beginning of the iteration:

```
< [LOOP-TEST]
  [COMMAND-SEQUENCE]
  [.................]
>
```

An example of the DO WHILE structure is:

```
< ^N:;
  HPW HK A
>
```

This is a macro loop which removes formfeeds from a file. WHILE the end-of-file flag is non-negative, the current buffer is copied to the output file (suppressing formfeed), the buffer is killed, and the next page is appended from the input file. When the end-of-file condition is met, the loop is exited.

REPEAT UNTIL — can be similarly constructed by using the looping command with a test at the end of the loop. The structure is given as follows:

```
<
  [COMMAND-SEQUENCE]
  [.................]
  [LOOP-TEST]
>
```

For example, the macro

```
<
  (OA)"V
       (OA-32)I$ D
      ¦
       C
      '
  .-Z;
>
```

uses the IF THEN ELSE code shown previously and puts it in a loop that is REPEATED UNTIL the end of buffer is reached.

CASE — this construction is now available through the use of the computed GO TO command. The logic structure for this command is the hardest to represent clearly. It can be formatted as:

```
nOtag0,tag1,....,tagk$
  [.......................]
  [.......................]
!tag0! [COMMAND-SEQUENCE]
!tag1! [COMMAND-SEQUENCE]
!....!
!tagk! [COMMAND-SEQUENCE]
```

The following macro makes use of this command to branch to one of many possible insertion strings. The function of the macro is to change all occurrences of single digits 0-9 to their written form. The search string demands that the digit be preceded and followed by a separator character.

```
<
.-Z;
:S$$ED$$'S
          2R (OA-48)U1 D
          (Q1)O0,1,2,3,4,5,6,7,8,9$
         ¦
          0;
         '
!0!IZERO$ F<
!1!IONE$  F<
!2!ITWO$  F<
!3!ITHREE$F<
!4!IFOUR$ F<
!5!IFIVE$ F<
!6!ISIX$  F<
!7!ISEVEN$F<
!8!IEIGHT$F<
!9!ININE$ F<
>
```

PROGRAMMING EXAMPLES

Having described the basic forms of structure that can be applied when writing TECO macros, let us now consider a complete macro function that can be coded in different ways. The function of the macro is to search the buffer, replacing control characters (↑A - ↑H) and (↑N - ↑Z) with a ↑ and their respective character (A-H), (N-Z), and to replace all ESCAPE characters with a dollar sign $. This function is useful in editing TECO macros so that they may be listed on devices that do not properly handle escapes or control characters. The first sample of this function is coded with the basic TECO commands that are available in all versions of TECO and makes use of the GO TO structure:

```
< (0A)U1
  (Q1-27)"G
          ONEXT$
         '
  (Q1-8)"G
         (Q1-14)"L
                 ONEXT$
                '
        '
  (Q1-27)"E
          D I$$ -C ONEXT$
         '
  D I^$ (Q1+64)I$ -C
!NEXT! C .-Z;
>
```

The second sample of this function is written making use of the TECO-11 flow commands and IF THEN ELSE:

```
< .-Z;
  (0A)U1
  (Q1-27)"G
         C F<
        '
         (Q1-27)"E
                 D I$$ F<
                '
         (Q1-8)"G
                 (Q1-14)"L
                         C F<
                        '
                '
         D I^$ (Q1+64)I$
        '
>
```

The third form of this function is written using the CASE construction:

```
< .-Z;
  (0A)U1
  (Q1)0A,B,B,B,B,B,B,B,B,A,A,A,A,A,
      B,B,B,B,B,B,B,B,B,B,B,B,B,C$
 !A! C F<
 !B! D I^$ (Q1+64)I$ F<
 !C! D I$$ F<
>
```

It should be noted that in the above computed GOTO command, there should not be a carriage return or spaces between the referenced tags (this was just done to fit page restrictions).

The execution timing of the function differs greatly depending on which coding technique is employed. It was found that the first sample using the GOTO code took approximately three times as long as the IF THEN ELSE construction, and the CASE sample took about 1½ times as long as the IF THEN ELSE. It should also be noted that when using the IF THEN ELSE construction, the path that is followed most frequently should be the THEN path, rather than the ELSE path, in order to minimize execution time.

Complete Examples with Comments

TECO programs often lack sufficient comments describing their purpose, the way to invoke them, the Q-registers used, or the logic being followed. These items must be commented to allow others to understand and use the programs. The problem with comments in TECO, is the fact that TECO does not recognize the difference between comments and labels, and therefore slows down its execution of macros when comments are included. To avoid this problem, the comments should be removed before executing the macro. A macro suited for this purpose is shown in Figure 3. The macro is highly 'structured', using many of the techniques recommended. Note that its function and execution are well commented, including comments that the macro itself writes out to the terminal. It notifies the user of the form of comment that it will remove and also writes out a message to the terminal when it is finished executing. Although it does not have a great deal of control logic necessary internally, its main loop is readily identifiable as a DO WHILE loop with a primary IF THEN ELSE structure controlling the deletion of comments.

Figure 4 shows the macro INPUT which will accept characters typed from the keyboard and insert them into the TECO buffer. It supports control-R (repeat line), control-U (delete line), control-Z (exit), RUBOUT or DELETE (delete last character typed), and carriage return or ESCAPE (exit).

This macro contains a great deal more internal use of logic structures. It was easy to write and debug primarily because of its use of indentation techniques that made the format of the IF THEN ELSE commands stand out. Note that neither of the examples needs a GOTO label because of the use of the structured logic forms.

CONCLUSION

For complex editing functions TECO is well suited, but the obscurity of the language's syntax must be treated with care to obtain the maximum efficiency of both the programmer's and machine's time. We have shown that TECO contains the syntactical capabilities for structured coding techniques. If properly utilized, these techniques along with proper labelling and commenting conventions, allow the programmer a clearer, more readable way to write TECO programs and in turn enable the programs to be debugged or modified more quickly and efficiently.

```
!* TECO macro: COMOUT.TEC
   This macro removes comments from TECO macros by allowing the
   user to define the difference between a label and a comment. The
   string that starts a comment (beginning with an exclamation point)
   can be any length; it is defaulted to be an exclamation point followed
   by an *. Similarly, the string that ends a comment can be any length
   but must end with an exclamation point; the default is an * followed
   by an exclamation point. The macro will NOT remove comments starting
   in column 1.
      The macro uses the following Q-registers:
          A - holds the start-of-comment string
          B - holds the end-of-comment string
          1 - points to beginning of current line
      To execute this macro:
          Load Q-registers A and B with your comment conventions if you
   want to override the default settings, put the macro to be acted upon
   in the buffer and execute the EI command as follows:
          EICOMOUT$$
*!
EI$
:QA"E                                        !* IF no comment string provided *!
    ^UA!$ :^UA*$                          !* THEN set up default comment format *!
    ^UB*$ :^UB!$
    '
^ACOMOUT will remove all comments with the format:
^A :GA ^A comment ^A :GB ^A
^A

< S^EQA$;                              !* LOOP-TEST:search for start-of-comment *!
 (.-:QA)U1 0L              !* save start position; move to beginning-of-line *!
 (Q1-.)"E                          !* IF comment starts at beginning-of-line *!
        C F<                            !* THEN go to beginning-of-iteration *!
       |
        S^EQB$ (Q1,.)K                !* ELSE find end-of-comment and kill it *!
       '
>
^A ***DONE***
^A
$$
```

FIGURE 3.

```
!* TECO macro: INPUT.TEC
   This macro accepts characters from the terminal and inserts them
   into the buffer. The input stops when the user types a carriage return,
   a ctrl-Z, or an ESCAPE character. A ctrl-R causes the character string
   to be typed out to the user; a ctrl-U deletes the character string from
   the buffer; a DELETE or RUBOUT deletes the last character typed.
      The following Q-registers are used:
         1 - contains original buffer position
         T - temporarily holds input character
*!

.U1                                                  !*Save initial buffer position*!
<
 ^TUT                                                      !*grab character into T*!
 (QT-27)"G                                          !*IF not a control character*!
        (QT-127)"N                                                !*IF not a DELETE*!
                 QTI$ F<                                 !*THEN insert into buffer*!
                |
                 (Q1-.)"E                  !*ELSE IF there are no char to delete*!
                        7^T F<                                  !*THEN send BELL*!
                       '
                 (ET&2)"G                                                 !*IF a crt*!
                        8^T 32^T 8^T                      !*THEN send bs-sp-bs*!
                       |
                        ^A\^A (-1A)^T                  !*ELSE no crt send \char*!
                       '
                 -D F<
                '
       |                                                      !*ELSE (a control-char)*!
        (QT-13)"E                                                        !*IF a cr*!
                ^TUT 0;                                !*THEN grab lf and exit loop*!
               '
        (QT-27)"E                                                         !*IF esc*!
                0;                                              !*THEN exit loop*!
               '
        (QT-26)"E                                                       !*IF ctrl-Z*!
                0;                                              !*THEN exit loop*!
               '
        (QT-21)"E                                                       !*IF ctrl-U*!
                (Q1,.)K 13^T 10^T F<                       !*THEN delete line*!
               '
        (QT-18)"E                                                       !*IF ctrl-R*!
                13^T 10^T (Q1,.)T                          !*THEN retype line*!
               '
        QTI$ F<                                    !*All else insert in buffer*!
       '
>
```

FIGURE 4.

REFERENCES

1. Basili, Victor and Baker, Terry, **Structured Programming Tutorial**, IEEE, 1975.

2. Cheatham, T.E. and Townley, J.A., "A Proposed System for Structured Programming" in **Programming Symposium**, edited by B. Robinet, Springer-Verlag, 1974.

3. **TECO User's Guide and Language Reference Manual**, TECO SIG, March 1979.

4. VanTassel, Dennie, **Program Style, Design, Efficiency, Debugging, and Testing**, Prentice Hall, 1978.

5. Zann, Charles T., "A Control Statement for Natural Top-Down Structured Programming" in **Programming Symposium**, edited by B. Robinet, Springer-Verlag, 1974.

6. Stafsudd, Jacquie, **Structured Programming in TECO**, Proceedings of the Digital Equipment Users Society, Fall, 1979.

7. Stafsudd, Jacquie, **Adding Structure to TECO Programs**, Proceedings of the Digital Equipment Users Society, Spring, 1980.

An Open Letter . . .

TO: RSTS Community

FROM: Jerry Kiestler
The University of Tennessee at Martin
Computer Center
Martin, TN 38238

RE: Spooling Package Task Force

At this past DECUS Symposia in Chicago, a task force was established to work on developing a document of the requirements that the spooling package on RSTS needs to meet in order to satisfy user needs. Due to the large number of wish list items which had been submitted at past Symposia concerning the spooling package, it was obvious that something needed to be done.

It is hoped that the task force will be able to develop a functional specification for a new or revised package. In order for this to be successful we will need the cooperation of as many RSTS users as possible in supplying us with input in the form of suggestions, complaints, wishes, etc. ANY user not satisfied with the current spooler system should send the task force any input they have on the topic.

At the Chicago meeting there seemed to be a very large majority who were dissatisfied with the package's current state, therefore, PLEASE send us your input.

We will try to put together a general outline of our initial findings for publication in the RSTS newsletter as soon as we can gather and sort the material.

BASIC-PLUS AS AN ENVIRONMENT FOR THE IMPLEMENTATION OF THE NAIVE USER INTERFACE IN A HIGH-LEVEL PROGRAMMING LANGUAGE

By Tony Kobine and Ed Taylor, Finar Systems Limited, New York, New York

ABSTRACT

FINAR, Financial Analysis and Reporting, is a language that has been designed to carry out a wide range of functions in financial modeling and budgeting. The system has been designed to fulfill some specific objectives, including that of being easily operated and understood by a completely inexperienced user after the minimum of instruction. Examples of techniques which achieve this are given.

In addition, because of the way in which FINAR is implemented, in BASIC-PLUS under RSTS/E, it is possible to make the operation of the computer itself almost transparent, which means that such functions as error-trapping and correction can be handled by the FINAR package itself. At all times, the naive user has complete control over the progress of his runs and is kept informed of what is happening next, in financial analyst's language rather than programming terms.

NEED FOR THE NAIVE USER INTERFACE

The Financial Modeler

The computer age has brought important benefits to our lives, not the least of which is that the authors and also most of the people who will read this paper make their living from computer technology. Everyone is affected by developments in data processing.

These developments have been paralleled by an unfortunate side-effect: the invention of a bewildering array of jargon, with a corresponding gap in communication between computer specialists and the rest of mankind. Some of it is inevitable — how could we talk about hardware or software without referring to cycle-time, bits, bytes, etc.?

What is the end-user to make of this? There are two choices available to a naive user who wants to take advantage of the processing power of modern computers. He can become an expert, and swap technical terms with the rest of us. Or he can try to find a computer system that he can understand, and which can be instructed in a language similar to the one he uses in his day-to-day work.

FINAR is such a language, and it has been designed for executives and managers who carry out forecasting, budgeting, modeling and other planning activities. These types of applications are related to the traditional jobs which are typically handled by a computer system such as the PDP-11: general ledger, accounts payable and receivable, payroll, etc., but are normally somewhat less time-critical. However, they are just as vital, even for a small company, especially in today's volatile business climate.

Programmers in the Boardroom

In many applications, there are well-defined methods of dealing with data input, formatting of output, selection of options, and specification of the logic to be performed. The various possibilities are generally selected by means of a question-and-answer routine, often depending on some sort of logical "tree", where the selection of one branch leads on to further selections until action can be taken. When combined with screen formatting and menu selection, this is a powerful and straightforward way of handling the user dialogue.

This technique is not, however, suitable for financial modeling. Although the question-and-answer technique is appropriate for choosing the action to be taken next, and has been used for this in a number of packages, the definition of the various actions themselves is entirely dependent on what the model is intended to do. A budgeting model might take figures for twelve months, produce totals and percentages, and then store data for use by an Income Statement. A project analysis could consider various alternatives for financing, with different calculations for lease versus buy, tax advantages and present values. Each model is different, and even the structure of reports for different departments of the same company can be different.

The financial modeler could have programs written to carry out every task that he might want to do. Each new application (for instance, to analyze the purchase of a new factory or the divestment of an affiliate) will require program modification or perhaps entirely new software. Even minor changes could demand substantial programming effort (for instance, to re-format a report or redefine a company ratio).

Many high-level executives still carry out their planning by hand because it is impossible to respond adequately to their needs with purpose-built software.

The obvious solution is to let the potential user program his own models, either in a general-purpose high-level language, or in a special-purpose language such as FINAR. Most financial analysts are unwilling to learn how to write BASIC-PLUS or COBOL, which demand an excessive amount of knowledge of computers, especially when handling such jobs as opening and closing files, defining reports and editing programs or data.

If we are then to expect boardroom-level personnel to write their own computer programs, even in a special financial modeling language, it is clear that the system must be easy to learn and remember, straightforward in operation, extremely powerful, and contain a wide range of possible options. In this paper we will try to explain how these requirements have been met in FINAR, and some of the features of BASIC-PLUS that have been used to implement the system.

WHAT DOES FINAR DO?

Financial modeling is generally concerned with the manipulation of tables of data, known as "worksheets". The size and shape of the worksheets are defined, along with text and abbreviations associated with each row (considering the data in lines across the page), and each column (considering the data vertically). There may be internal data groupings, perhaps by product or by division, and these groupings may relate to both rows and columns. Figure 1 shows a typical worksheet (albeit rather smaller than most real-life worksheets), and Figure 2 gives the FINAR instructions that would be used to define it.

Note the use of abbreviations to identify each row and column, along with more verbose text to be used in the reports, contained in parentheses. The grouping of the columns into threes is done by using a colon after the abbreviated name of each group, and before the abbreviated name of the items within each group.

The user next specifies the relationships that exist between the rows and columns. In data processing terms, these are usually not too complicated, but to a manager whose last brush with algebra was 25 years ago, even something as simple as a moving average can prove quite daunting. Thus FINAR supplies functions to carry out all types of arithmetic, plus finance-related calculations such as discounting, depreciation, rate of return, etc., each one specified by a form of words which is clearly understood by the financial analyst, for example:

360 Result = Internal rate of return of Cashflow
using abbreviations and defaults where required.

The presentation of results is extremely important, and both reports and graphs can be produced in FINAR. A very short series of instructions will cause default output to be produced, with more stylized formatting available if needed, again with English-like commands. Several reports can access

Cost Analysis for the Year

	Quarter 1			Quarter 2			Quarter 3			Quarter 4			Total		
	Act-ual	Bud-get	Vari-ance	Act-ual	Bud-get	Vari-ance	Act-ual	Bud-get	Vari-ance	Act-ual	Bud-get	Vari-ance	Act-ual	Bud-get	Vari-ance
Raw Materials	1243	1220	102	1319	1257	105	1399	1320	106	1484	1420	105	5445	5217	104
Manufacturing	3302	3310	100	3391	3468	98	3488	3492	100	3496	3544	99	13677	13814	99
Packaging	368	368	100	368	406	91	368	368	100	368	371	99	1472	1513	97
Transportation	876	1030	85	885	1030	86	900	950	95	903	749	121	3564	3759	95
Selling	1955	1800	109	2135	1980	108	2331	2166	108	2546	2382	107	8967	8328	108
Advertising	856	1144	75	856	1326	65	856	1062	81	856	931	92	3424	4463	77
TOTAL COST	8600	8872	97	8954	9467	95	9342	9358	100	9653	9397	103	36549	37094	99

FIGURE 1. A worksheet.

```
100  Rows  RM(Raw Materials)  Man (Manufacturing)
110  Rows  Pack (Packaging)  Tran (Transportation)  Selling
120  Rows  AD (Advertising)  TC(TOTAL COST)
200  Columns  Q1: (Quarter 1)  Q2: (Quarter 2)  Q3: (Quarter 3)
210  Columns  Q4: (Quarter 4)  Total:
220  Columns  :Act (Act-)(ual)  :Bud (Bud-)(get)  :Var  (Vari-)(ance)
```

FIGURE 2. Row and column definitions.

the same or different worksheets, allowing full results or summaries to be printed.

What has been described so far could probably be achieved with a competent typist and a calculator. The real merit of computerized financial modeling occurs when simulation is carried out — the response to "What if?" questions. In a typical budget preparation exercise, the figures might be changed over and over again, requiring the recalculation of the whole worksheet: this could involve thousands of calculations being carried out dozens of times, with new reports each time. It is clearly vital for a financial planning system to have adequate means of editing data and keeping copies, and also of editing the logic and reporting sections if this is required.

USE OF RSTS/E

Structure of FINAR

We have seen that a financial modeling system needs all the flexibility of a programming language, although restricted to a special set of applications. Flexibility, in turn, increases the likelihood and scope of users' errors, and places special demands on the implementation language. It is traditional to use Assembler, or at least a low-level language, to achieve the degree of complexity and monitor-interface required by general-purpose programming languages and run-time systems. FINAR has been implemented in BASIC-PLUS, and we feel that this has been sufficiently successful to warrant encouraging other PDP-11 users to consider writing their own English-language style applications in BASIC-PLUS.

Figure 3 is a simplified block diagram of FINAR. It will be seen that FINAR has a structure analogous to that used by BASIC-PLUS with its "BAC" files. The instruction editor serves also as the monitor of the system, analyzing requests for action in "direct" mode. A series of flags keeps a record of which parts of the FINAR source have been modified, thus obviating the need for recompilation of the whole model when something trivial such as a report title has been changed. This demands a limited amount of syntax checking by the editor at the time of instruction entry. The compiler operates in one pass, and generates two sorts of output:

1) Tables of text and pointers, which are normally stored on file as they are generated, although some are sorted: the row and the column abbreviated names are in alphabetical order, for example, to allow quick lookup.

2) Code which is directly translated from FINAR source. Since one FINAR instruction can generate a very complicated series of calculations, the amount of code per instruction varies greatly.

The run-time routines process the compiled code, producing worksheets as required, and printing reports on terminal or line-printer by user request. Different modules handle the case where the worksheets are large and have to be paged from disk, or are small enough to be manipulated in core, although this is transparent to the user.

Other routines (some not shown) deal with data editing, formalized simulation (sensitivity analysis and reverse iteration), transfer of data between models, and reordering of changed worksheets. The whole system has been designed in such a way that a user will not need to use any RSTS/E commands whatsoever, except for HELLO, RUN and BYE.

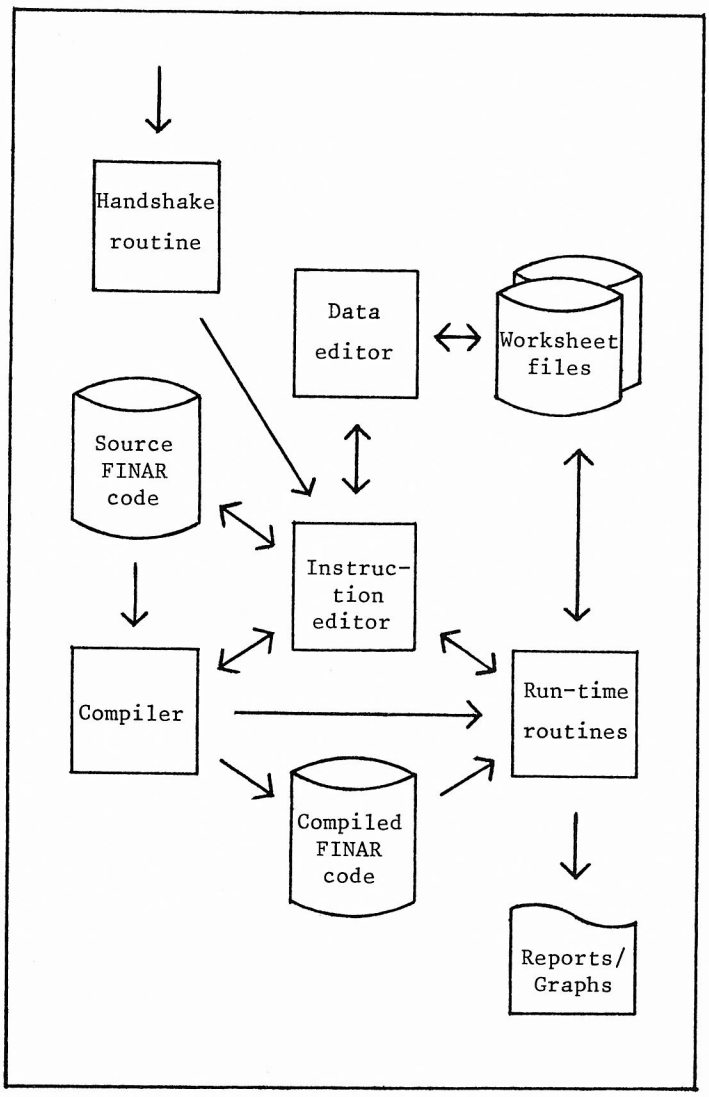


FIGURE 3. Simplified FINAR block diagram.

Benefits from BASIC-PLUS

There are a number of BASIC-PLUS features which are readily translated into advantages for the user of the FINAR system. Some of these have been achieved in a slightly unusual or non-standard way, and Figure 4 shows how a selection of BASIC-PLUS statements and facilities which are not generally found in other languages have been translated into worthwhile benefits for the package's users.

FINAR

In the following sections we look at three important aspects of financial modeling where particular attention has been paid to giving the financial analyst flexibility, while retaining clarity and control of possible errors.

These aspects are:

1) Data Access and Editing — freeing the user from consideration of file nomenclature and organization, by allowing him to specify data cells to be modified using descriptive, English names that he has chosen for his data structures.

2) A method for permitting the user access to the three dimensional matrix inherent in consolidation examples, which replaces the programmer's indexing conventions (e.g. X (I, J, K)) by a system of user-defined names.

3) The detection, interpretation and correction of syntactical and logical errors.

Feature of BASIC-PLUS	Advantage to the Programmer	Benefit to the FINAR Financial Modeler
1. XLATE, INSTR, RIGHT, LEFT, MID, etc.	Fast parsing of phrases composed from a large vocabulary.	Free-format English language input. Few rules or conventions.
2. ON ERROR/ ∧ C	Controlled abort from inadvertant program loops or unwanted output.	Orderly return to FINAR "monitor" instead of RSTS/E "Ready".
3. ON ERROR/divide by zero, illegal exponentiation, etc.	Replace erroneous result by sensible default, and inform user.	Saves user from exhaustive mathematical analysis of logic except for special cases.
4. INPUT LINE, INSTR, ON ERROR/VAL, etc.	Pinpoint data errors during terminal input.	Easy recognition of where and why error occurred. No puzzling through lists of error commands.
5. KILL, NAME . . AS	File-handling from within a program.	Users do not have to understand or learn system commands.
6. POS	Position of user's printhead can be found: (a) Type-ahead can be detected. (b) Character deletion with backslashes can be detected.	(a) Prompt can be suppressed to avoid confusion. (b) Line can be repeated back for confirmatory purposes, c.f. ∧ R.
7. Alternative line terminators.	Different actions can depend on simple keystrokes.	Shorthand method of doing commonly recurring activities.
8. ∧ Z trapped as End-of-file on terminal input.	Can be treated as a line terminator.	Use instead of ∧ U.then the prompt can be repeated, avoiding user confusion.
9. INPUT without prompting "?".	The prompt can be printed ahead of time.	Slow typers are encouraged to type ahead. (Obviously the preceding activity must not be able to generate errors).
10. Variable PRINT . . USING image strings.	Run-time formatting of output.	Choose parentheses, commas, etc. via simple option.
11. Directory lookup (SYS/FIP calls).	(a) Scan directory from within program. (b) Trap missing data file expected by user.	(a) No need to go to RSTS for catalogue of models, data files, etc. (b) Automatically prompt for required data.
12. Virtual arrays.	Rapid processing of segments of large files.	Consolidations, partial reports, etc. with acceptable response time. Worksheet size not dependent on core memory.
13. Core common.	Transfer of (limited) amount of data between chains without using files.	Speed up response.
14. $QUE available from BASIC-PLUS.	Program-controlled spooling.	Avoids learning about D.P. concepts such as spooling, on-line, etc.
15. Variable device names.	Optional distribution of files over different disks.	Freed from understanding efficiency considerations of a particular hardware configuration.

FIGURE 4. Benefits from BASIC-PLUS.

Data Access and Editing

To a BASIC-PLUS programmer, the procedure for accessing a matrix is relatively easy. The dimensions of a suitable array can be defined, and any element may be specified by giving two coordinates. A series of elements may be referenced with FOR. e.g.

A (1%,3%) = 0.0 FOR 1% = 10% TO 50% STEP 10%

and matrix arithmetic is available if required.

The FINAR user can carry out similar tasks, with a more suitable and application-oriented syntax. Calculations between rows and columns are done by using the abbreviated names that have been defined by the user.

For example:

1) Profit = Revenue — Sum of Expense1 to Expense4
   which is a calculation between rows.

2) Total = Q1 + Q2 + Q3 + Q4
   Var = Act as a % of Bud
   which carry out column arithmetic, and would be used to perform the column calculations of Figures 1 and 2.

The second example illustrates that a large number of calculations can be carried out with just one FINAR instruction: in this case, the arithmetic is performed for all the rows and for each of the column groups that were defined. In programming terms, this is equivalent to a statement involving a range of matrix elements and two FOR loops — not a procedure that is recommended for use by most financial vice-presidents!

Having defined the worksheets and the calculations, we now have to consider the fact that much of the financial analyst's work is concerned with changing data. During the simulation phase, a large number of alternatives may need to be tested, and this often requires the substitution of one series of numbers for another, either temporarily or permanently.

The need to use the system editor to change data files frequently proves an insuperable problem to the non-programmer. The difficulties are twofold:

1) How to enter the editor and re-enter the application program.

2) Using the editor itself.

Although even a complex editor, such as TECO, can be learned by anyone given time and practice, it is obviously simpler to have the alternative of a built-in editor. In FINAR, an approach similar to that adopted in some early general-purpose languages (JOSS and TELCOMP, for example) has been implemented.

When editing in FINAR, instead of each line being considered as a separate entity, two lines are considered together, with the second line modifying the line above. The system promptly changes from "?" to "??" to remind the user that editing is taking place. A small number of special characters indicate the editing to be done:

Space — leave character above unchanged.
! — delete character above.
<— insert characters following.

Otherwise the character typed underneath replaces that above.

With the use of the FINAR "CONSIDER" instruction, data can easily be changed. For example:

```
? Consider Worksheet 212
? Edit Advertising
  Advertising = 13.6 14.2 14.5 15.1
??                    3
?
```

The CONSIDER WORKSHEET instruction causes FINAR to open the correct file, whose name depends on the project name and the worksheet number. The name ADVERTISING specifies a row in the worksheet, for which FINAR locates the data which is then displayed. The user spaces along until under the character to be changed, and then a "2" is replaced by a "3". The instruction is handed over to a parsing routine which analyzes the line as edited, and puts the data in the appropriate location in file. Note that the separator between numbers is one or more spaces, which is more natural than the usual "computerese" comma.

Obviously the editor-writers of the world are not going to be put out of business. However, the availability of even this simple facility at all times means that a FINAR user can learn in five minutes to do something that could otherwise take days to comprehend. As we shall see later, the same editing system is used for instructions, which has the additional advantage of allowing even a beginner to correct mistakes from the outset.

If there are too many numbers to handle conveniently on one line, CONSIDER can do the same job at the BASIC-PLUS statement FOR, and is indeed implemented as such in the run-time routines.

For example:

```
? Consider MAR to JUN
? Edit Income
  Income = 3061 3108 3192 3244
??          94   30     <.5
?
```

In this case, the figures become 3094 3130 3192.5 3244.

The editing is achieved with a relatively simple BASIC-PLUS subroutine which checks each character of the editing string for one of the special characters. If none is found, the output string has one character of the editing string appended to it. A space causes the first line to be used, and "!" simply moves the pointer along by one character. Insertion with " < " just requires the character position to be correctly stored, and the only pitfall could be careless use of CVT$$(X$,—1%), since trailing spaces are signficant here.

The data analysis routine is somewhat more complicated. This is because various options are available to the user in addition to actual numeric input, for example:

Income = 3072 3105 grow by 10%

These are handled with a standardized parsing subroutine, which returns an argument indicating the type of "word" found next (e.g., keyword, number, symbol, don't know, etc.). The analysis can then be done from a central return point, with ON . . GOTO or ON . . GOSUB to deal with the various possibilities. The BASIC-PLUS function VAL and intelligent trapping of ON ERROR means that detection of the amazing variety of errors that a user can make on data input is not too burdensome, and their location can be accurately indicated with a diagnostic message.

We can take advantage of the BASIC-PLUS flexibility in allowing alternative line terminators to provide another facility. If a line is terminated with ESC, this has the meaning "act as though I had requested another EDIT".

For example:

```
? Edit Volume
  Volume = 52.88 51.6 50.22
??           !    <4$
  Volume = 52.8 51.46 50.22
??                    39$
  Volume = 52.8 51.46 50.39
??(CR)
?
```

The first time ESC is used (echoed as $), the line is retyped by FINAR, thus permitting characters to be changed after an insert sequence. The second time, this is a "fake" edit, and nothing is changed, but it allows the user to check the line before pressing Carriage Return and having the data stored.

An extension of the ESC idea uses the BASIC-PLUS statement WAIT. If two ESC's are typed in quick succession, this means that information is to be added to the end of the line. For example:

```
? Edit Interest
  Interest = 6.41 6.8 6.82 7.16 7.2
??$$Interest = 6.41 6.8 6.82 7.16 7.23
?
```

which simply changes the last figure to 7.23. This is implemented by allowing one second for another ESC to be typed; if none is received, the single ESC is assumed. The delay in response time has not been noticed by FINAR users since the feature's implementation.

This double ESC facility has overcome a common mistake made by new terminal users: the premature typing of Carriage Return (in addition to forgetting to type it at all!). The apparently trivial feature just described is in fact a very easy solution to a tiresome beginners' problem.

Multiple Worksheets

A frequent requirement, and a problematical one, is the manipulation of data in three dimensions. See Figure 5. It is now clearly impossible to subject the naive user to a standard programming language. Even if BASIC-PLUS were to allow it, a simple operation such as:

```
Z(I%,J%,K%) = X(I%,J%,K%-1%) + Y(I%,J%,K%-1%)
        FOR I% = 1% TO N%  FOR J% = 2% TO 6%
```

is completely outside the capabilities of a financial analyst. The problem is that the generality available from a language such as BASIC-PLUS is in reality not required for the solution of a simple consolidation exercise.

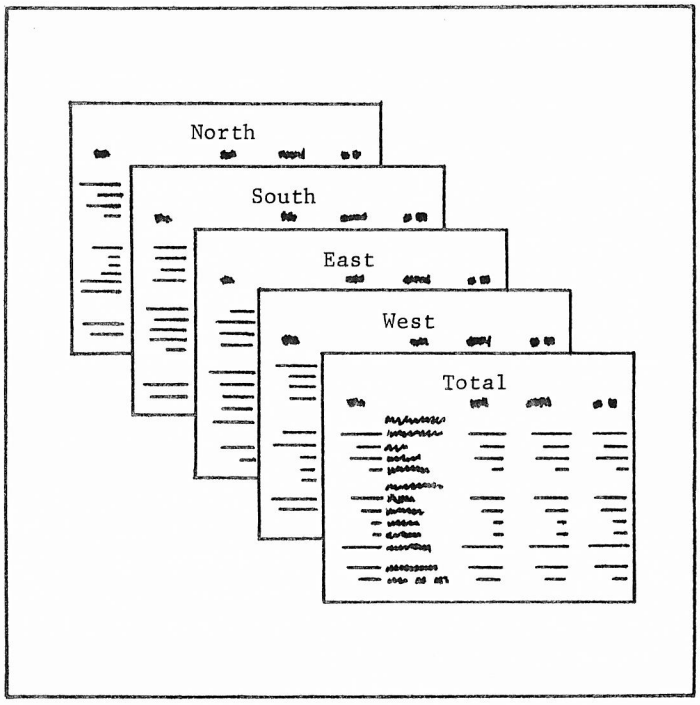


FIGURE 5. Worksheet arithmetic.

The "Worksheet Arithmetic" capabilities of FINAR allow the normal arithmetic features of the calculation section of a model to be applied to worksheets. In conjunction with CONSIDER, we have a powerful method of carrying out consolidations, eliminations, currency conversions, and in fact any of the usual inter-worksheet calculations that a financial analyst will need.

For example:

Consider Actual
Consider Expense1 to Expense4
Worksheet 8 = Sum of Worksheets 2 to 6 - Worksheet 7

We have restricted ourselves to using submatrices of cuboid shape, or at least cuboids with rectangles missing. By

doing this, the user's problem is at once reduced to the simple one of defining the edges of the cuboid, with the abbreviations previously given. In addition, by allowing CONSIDER with worksheets, the same set of calculations can be applied to a series of worksheets, either before or after the Worksheet Arithmetic itself. For example:

```
Consider Worksheets 21 to 30
Tax = Taxrate * Positive Values of Profit
Worksheet 23 = Worksheet 21 — Worksheet 22
```

where Worksheet 22 represents some special deductions to be made.

To preserve complete generality, a future version of FINAR will permit names to be given to worksheets, possibly with a grouping facility similar to that available for rows and columns.

Errors

One of the most frequently criticized aspects of software packages is the handling of user errors. In FINAR, a special effort has been made to respond in a helpful way to anything that goes wrong. Over 400 messages indicate what seems to have happened and where.

For example:

```
150 Rows Cosl (Cost)(Plan 1) Cosl (Cost)(Plan 2)
    You've already defined   ^   this name
```

A subroutine called SCAMP (Subroutine for Caret And Message Printing) locates a suitable space with XLATE and INSTR, and prints the message around the uparrow or caret in this space. This procedure saves time during model development, which is especially important when using a slow printing terminal.

The pinpointing of the exact location of the error is a valuable aid for the less obvious difficulties, for example:

```
460 Percentages are for .2 periods
 Use a whole number or  ^  a fraction like 1/6
```

A real difficulty arises in showing the correct location of an error when the problem is actually caused by something earlier in the line — an exercise which is familiar to compiler writers. It is worth making a special effort in this regard, to the extent of storing some extra flags and pointers, so that the user is not deceived. For example:

```
650 Total = Sum of Product1
     Using  ^  this function on a single item
```

Although the error is not discovered until the row Product1 is found (presumably this should be something like Product1 to Product9), the message refers to the SUM function, and a position flag is stored in case this error is discovered.

Where errors have occurred, the FINAR editor can be used for correction. Of course, some errors cannot be discovered in the compilation process, especially those concerned with the input of descriptive text, but since the FINAR editor is available at all times, corrections can be made whenever they are found. Users are encouraged to correct even minor typographical mistakes as they are discovered, since in the excitement of actually producing results, they sometimes forget all the editing that should have been done, leading to ultimate frustration.

The editor operates in the same way on instructions as it does on data. For example:

```
? Edit 300
  300 Val = Depreciation of Asst at 200%
??                                <over 6 years $
  300 Val = Depreciation of Asst over 6 years at 200%
??(CR)
?
```

Where an error has occurred, the use of the ESC key is interpreted to mean "edit the instruction that caused the error". For example:

```
  680 Pounds = Dollars / 2.2.2
  Don't understand this  ^  number
? $
  680 Pounds = Dollars / 2.2.2
                                !$
  680 Pounds = Dollars / 2.22
??(CR)
?
```

FINAR users occasionally develop sufficient interest in data processing in general to start learning more about computers. They often feel that they have been spoiled by FINAR when they start to study general-purpose programming languages: improvement could be made in most such languages in the way that errors are handled.

CONCLUDING REMARKS

The enthusiastic acceptance of FINAR by non-numerate users is directly related to:

1) Painstaking attention to initial design of the language, bearing in mind that the system will be used by managers who have no interest in computers, and who are unlikely to develop such an interest. A corollary is that the users' progress must be monitored closely, to discover where, despite our best efforts, difficulties arise in comprehension (the FINAR training course is an obvious place to to this), and updates to the system must include a way of addressing these problems in an upward-compatible way.

2) The relative ease with which a system can be created using BASIC-PLUS, and, following on from this, the speed with which new versions can be written. Obviously, a pseudo-interpretive language such as BASIC-PLUS yields an implementation of high-level language such as FINAR which is less efficient in terms of machine resources than the equivalent Assembler implementation, although the BASIC-PLUS-2 version of FINAR goes a long way towards achieving this efficiency. More importantly, the total FINAR environment is extremely efficient in its use of human resources, not only for the programmers and analysts who have written and who maintain the system, but also for the people who use FINAR in their day-to-day work. This is a tribute to the friendly and error-free nature of RSTS/E and BASIC-PLUS.

It is clear that we do not have the last word on the naive user interface, and neither does anyone else at this stage. The greater public awareness of computing matters and terminology, combined with the technical advances that are being made in hardware and software, mean that computer users can expect an even better deal from system designers in the future. It has taken about fifty years for automobiles to progress from being the prerogative of the expert who has learned how to operate the dozens of controls, to our current two-pedal vehicles that can be driven by anyone after a short period of instruction. We may have to wait just as long for computer systems that the man in the street can use, but each small step in the right direction moves the power of interactive data processing out of the hands of the elite few, and towards the grasp of the responsible manager, who demands the best decision-making tools to support his increasingly complex role in today's business world.

A RSTS/E TO VAX/VMS CONVERSION

By Jeffrey S. Jalbert and Susan Blount Duff
Denison University, Granville, Ohio

ABSTRACT

We have just completed a total conversion of our RSTS/E system to VAX/VMS employing VAX-11 BASIC as our primary language for applications programming. This paper describes the conversion process and compares the two systems and the two versions of BASIC.

At Denison, we have just completed a total conversion of our computing from RSTS/E to a VAX/VMS system. We were impelled to this decision by a complete exhaustion of the capacity of our 11/45, and a perceived need to have a system that would be able to sustain the demands of a burgeoning user community consisting of two major parts, academic and administrative.

Our academic users make up the bulk of the user population with about 2400 active accounts, and members of this group log onto our system about 1200 times per day. This load is encouraged by the stated goals of the college, one of which is to produce graduates who are literate in computing. This means that students are using computing actively in a wide range of disciplines ranging from physics to philosophy. Our social science users make heavy demands on our data-processing capacities, using SPSS, BMDP, ECPRESS, and MINITAB. Many students and faculty use the system for text preparation, and we believe that there are very few games active. Over 80% of our students use the system during their four-year stay. In addition to the above activity, students may major in computer science and we expect to graduate six to ten majors this academic year.

Our administrative users are from all offices in the college. We process student information from the time prospects make inquiries at the admissions office throughout their alumni careers. We support the maintenance of several inventories, do some billing breakdown for the telephones, do the general ledger, handle the mailing system, and produce the student payroll. We even do some work for the library in disseminating their acquisitions list. We have enumerated these application areas to give some idea as to the variety of real data-processing problems with which we have experience.

We have converted over 500 programs, the bulk of which were initially written in BASIC-PLUS, with a sprinkling of BASIC-PLUS-2, FORTRAN and PASCAL. The target language on the VAX was VAX-11 BASIC for all the BASIC-PLUS programs. FORTRAN or PASCAL were used for the others.

Our data-processing applications were supported in three different ways. First, very specialized applications such as course registration had a special set of programs. All financial systems were implemented using a purchased data management system, while the remainder of the work was implemented through a home-grown data management package.

A BENCHMARK

Before continuing, it will be necessary to describe briefly the target hardware configuration. The VAX processor was supplemented with a floating-point accelerator, which adds about 20% to its speed in complex floating point calculations. The memory is two and one-half megabytes of MOS, and the disk subsystem consists of two RP06 drives, each on a separate massbuss controller. There are two TE16 tape drives, a card reader, line printer, and 64 DZ11 lines. We chose a large memory configuration because of our experience on RSTS/E and our target of supporting 64 timesharing jobs simultaneously.

We ran a benchmark on our system in order to verify its performance at the level of 64 jobs. To do so we developed a program which could exhibit a wide variety of characteristics. A particular set of these characteristics could be stored in a file, which then became a script for a session. The script would be performed, and when finished, the program would repeat that script. Performance statistics were collected and analyzed after every ten iterations. The choice of activities for a script included the following:

terminal output
CPU bound
disk I/O by directory
disk I/O by file
scheduling

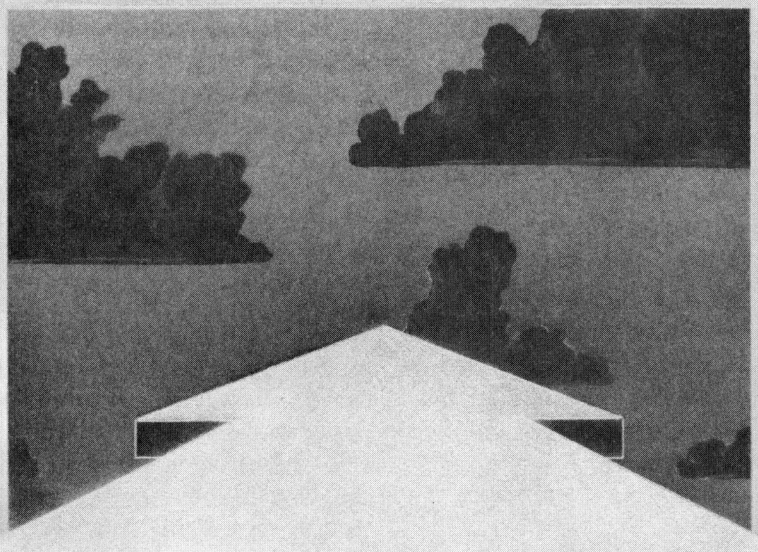
This test program was written in BASIC-PLUS-2 on our RSTS system and implemented in that language on the VAX in compatibility mode. Consequently the images were not shared between processes; each has its own copy of the code. Our expectation is that system performance would have been somewhat better if we had used VAX-11 BASIC to run the test, but only because there would have been no swapping at all.

Sixty four different scripts were developed, although there were similarities between many of them. These scripts were then divided into eight groups of eight. Each set of eight had the following composition:

4 highly interactive scripts
1 script simulating a program listing
1 script that is completely CPU bound
1 script that is completely disk I/O bound, by directories or files
1 script that mixes CPU and disk work

The object of this mix was to test the system at its limits:

Now You Have ACCESS!



Enjoy New Perspectives in Programming New Horizons in Data Management

Logical Systems announces ACCESS, an information management tool for RSTS Users

ACCESS improves productivity by generalizing your development. Programming becomes easier. Software maintenance time goes down. Your system runs and looks better, is more manageable and more modern. You feel better! And best of all–you can use ACCESS and all its features for much less than you'd think!

Imagine what these ACCESS features can do for you.

- Screen, data, and report definitions are all dictionary driven.
- Using direct cursor control, ACCESS automatically formats and protects all screen handling.
- Input fields are edit checked with the dictionary for validity and type.
- Data files are protected through a "layered" security system.
- Record retrieval is multi-key.

And there's a source program generator, a report generator . . . and more!!

Call or write:

LOGICAL SYSTEMS

Software Distribution
P.O. Box 2676 / La Jolla, CA 92038
714-455-5211

maximum CPU work, maximum disk I/O, and worst case directory (FIP) conditions. We hoped that our normal operating load would be exaggerated by the benchmark as designed.

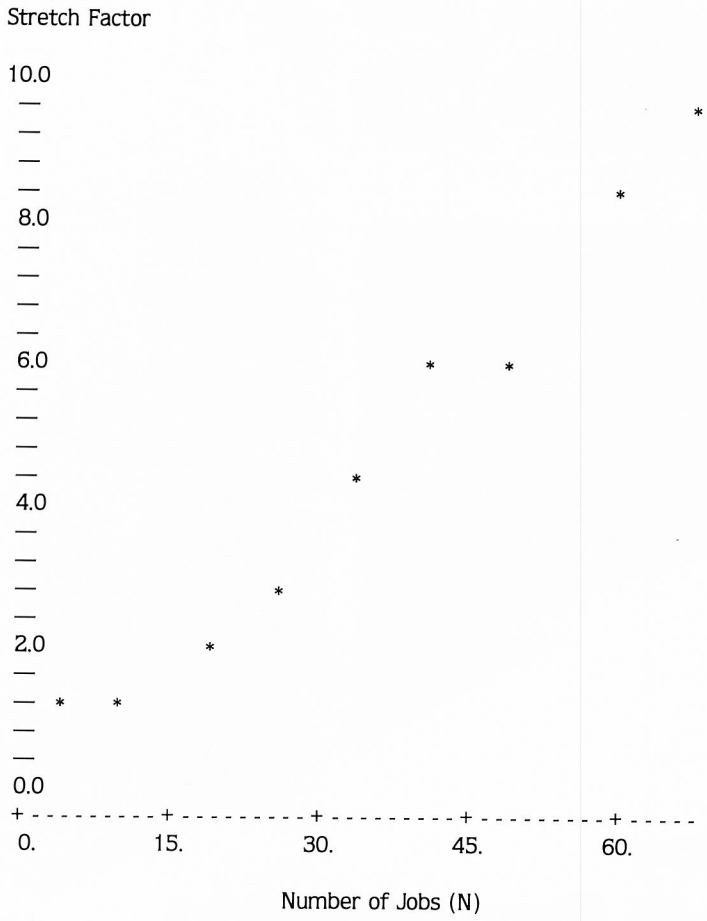


Figure 1.

We then hooked one terminal up to each of eight DZ lines in succession and started a new script. The progress of the whole benchmark was monitored by a special "test" script which we ran at the console. The performance of the console job was monitored each time 8 jobs were added, and the entire result scaled by the time it took the test job to run alone. The results of this test are presented in the diagram below.

In addition, we ran a regression on the data in order to test its linearity. The resulting regression equation was:

$$S.F. = 0.813 + 0.0712*N + 0.001*N*N$$

Our expectation was that at some point there would be a knee in the stretch curve. That knee would then be used to determine the maximum job load that a VAX could handle. It is clear from the above results that the response over the entire range tested is linear, with a slight improvement at about 45 jobs due to the release of time from the management of free and modified page lists.

In addition to these stretch factors, we determined some factors which affect the subjective feel of a system. A directory command would take between 5 and 15 seconds to initiate, but complete in a constant interval with no hesitations between file listings. Editing a one block file with TECO took 17 seconds. Purging a large directory took 4.8 seconds. Deleting a 400 block file took 4.8 seconds. A truly subjective judgement by a user who wrote and ran a small program was: "It's not too bad, certainly much better than the RSTS system with 25 users."

In performing this test we learned that some of the common wisdom for operating a VAX system was in error. The most notable problem was the suggestion that we modify the priorities of system jobs. As a result of doing this we managed to hang the console because there was enough higher priority activity so that the CPU was 100% utilized and the system jobs could never be scheduled. Our recommendation for systems with sufficient memory is that the standard priorities not be disturbed. VMS seems to be able to manage its memory resources sufficiently to keep the processor completely busy, and in these cases swapping will not be so devastating as to impede the overall system performance. At 64 jobs, there were approximately 24 swapped at any one time, and none of these was actively requesting processor time.

VAX-11 BASIC

The primary language used to implement our applications on the VAX was VAX-11 BASIC. This is an optimizing compiler which generates VAX native mode code. It is fully compatible with all other VAX native mode software, and in particular has access to all system services. Many of these system services are similar to RSTS/E system function calls. However, access to these services under VMS often involves worse bit-twiddling tricks than required by RSTS/E. The compiler supports the wider range of VAX data types but can only support in a given program either single or double-precision floating point variables but not both. Both single and double length integers can be mixed in all programs.

The compiler supports many features that aid debugging such as immediate mode. In an environment using subroutines, immediate mode can be used by "loading" these subroutines into BASIC's memory before program execution is begun from the Ready prompt. The compiler is a native mode image and exhibits considerable speed.

PERFORMANCE COMPARISION OF DIFFERENT BASIC'S

In order to get a feeling for the speed of VAX-11 BASIC we applied several tests. One was a very simple program which added in floating point the numbers from 1 to 100000. A comparison run of this program in several different dialects of BASIC and on three different machines produced the following execution time results:

LANGUAGE ENVIRONMENT	TIME
VAX BASIC	1
VAX BASIC-PLUS-2	58
VAX FORTRAN IV PLUS	.6
VAX PASCAL	.8
VAX COBOL 74	26
11/70 BASIC-PLUS	28
11/45(CACHE) BASIC-PLUS	38
TRS-80 LEVEL II BASIC	668

Perhaps a better estimate is the wall time taken to execute a complicated task. We developed two comparisons like this. The first was a program which processed a 20,000 record file of our prospects. This program looked up the alumni club area for each record as determined by a zip code table and stored the result back in the file, and did no other computation. In BASIC—PLUS-2 (compatibility mode) this took 75 minutes. In VAX-11 BASIC this took 15 minutes, and improvement by a factor of five. Similar comparisons between RSTS/E BASIC-PLUS-2 and VAX BASIC-PLUS-2 give a performance of roughly 1.5 to 1 in favor of the VAX, of course considering that the VAX jobs were running in an active environment, while the RSTS environment was otherwise quiescent. Comparisons of BASIC-PLUS on RSTS to BASIC-PLUS-2 in the past gave us about a 5 or 6 to 1 improvement, so that overall, a program compiled using the VAX-11 BASIC compiler would be approximately 35 times faster than a corresponding program using RSTS/E BASIC-PLUS.

THREE CONVERSION STRATEGIES

We identified three general strategies that would be used in converting our systems. These are:

1. Recode the programs.

2. Modify the programs to use the full functionality of RMS and all the other functionality of VAX-BASIC.

3. Use BASIC VIRTUAL organization to mimic RSTS disk structure, and otherwise minimize any other changes to the programs.

We experienced a variety of difficulties during the conversion. Following is a discussion of many of the problem areas.

SYNTAX DIFFERENCES

VAX-11 BASIC is an extremely rigorous compiler. By this we mean that it will enforce syntax requirements to a degree BASIC-PLUS and BASIC-PLUS-2 users have not yet experienced. Thus we discovered blatant syntax errors in programs that were operating satisfactorily in both of these languages. This difference is only an issue at conversion time and then primarily only a nuisance and not a real obstacle, but it did give us pause to think that these systems had been guessing at what we wanted. Because it is an optimizing compiler, constant subscripts are checked at compile time, and resolved into absolute addresses then. One result of this procedure is that some things that used to be execution time errors are now compile time errors.

In addition VAX-11 BASIC enforces the stricter syntax of BASIC-PLUS-2 rather than that of BASIC-PLUS. As a result, some programs will not compile without errors, and others, although they will compile error-free, execute in unexpected ways. Some examples of this follow.

First, in fairness we should point out that many of the instances of syntax difficulties occurred in code written in less than admirable or recommended style and which made use of undocumented features of the language. RSTS allows a casualness that approaches sloppiness that can cause real difficulty when converting. Also RSTS oldtimers will recall what contortions were necessary to squeeze a lot of program into a little space. Unfortunately this encouraged programmers to develop bad habits. However undesirable, this old-style code is prevalent in many RSTS installations and in our case, the source of more than its share of conversion headaches.

This segment will not compile:

```
FOR I%=1% TO 10%                          &
IF FNA$(I%) < >    " "                    &
  THEN NEXT I%
```

The problem here is that the NEXT is in the THEN clause. (Admittedly this is not a recommended construction.)

Nor will this line compile:

```
FIELD 3% AS A$
```

The problem here is that channel zero is not explicitly referenced. Again, this is an undocumented feature of BASIC-PLUS and therefore no loss of supported functionality.

These are relatively trivial problems compared to those caused by unexpected execution results. The treatment of strings in VAX-11 BASIC and BASIC-PLUS is substantially different. When a fielded variable is used as the source of a LET, the only thing that BASIC-PLUS does is update a pointer. VAX-11 BASIC creates a whole new string. Those programs which are coded to take advantage of this feature will compile without diagnostics, but will execute improperly. Moreover, some programs are rather subtle in their manipulation of the side-effects. An example will suffice:

```
DEF* FNGS$( . . . .
      .
      .
      .
FIELD #1%, R%*(I%-1%) AS A$, I% AS RECORD.$
FNG$=RECORD.$
FNEND
```

This routine was called by the following:

```
RECORD,$=FNG$( . . .
LSET RECORD.$= . . . . . .
      .
      .
      .
```

In BASIC-PLUS the record in the buffer is updated. In VAX-11 BASIC a new string is created for the FNG$, and a third copy of that string is created for the RECORD.$ variable. The data is inserted in that second copy, not in the record buffer. Of course, any updates made by this program will not be applied to the disk file.

Syntax errors will be flagged if variables defined in DIMENSION statements, MAP's or COM's are referenced before the occurrence of these data declarations. This means, of course, that all DIM's, MAP's, and COM's have to be moved to the first of the program.

In general, all syntactical differences experienced in converting BASIC-PLUS programs to BASIC-PLUS-2 will be likewise experienced in converting to VAX-11 BASIC.

CONVERSION FROM RECORD I/O TO RMS

VMS essentially supports only RMS files, so all BASIC programs will have to be modified to use some form of RMS file structure. This produced a whole host of problems.

RMS is very precise about the difference between PUT'ing a record and UPDATE'ing it. Therefore, one cannot use PUT for all file storage operations. As a result of this, some algorithms that were used on RSTS/E were no longer valid and had to be substantially modified. One simple solution in many cases was to prewrite every block of the file and then only use UPDATE in the course of the program.

Every single OPEN statement had to be investigated and most of them modified. OPEN statements must be very explicit about what they are going to do with a file, what kind of a file it is, etc. The OPEN statement has many clauses, most of which we had to include because the default values were not desirable in our case. For instance, one must specify file organization or it will default to sequential variable. For fixed length files, the logical record size must agree either with the explicit declaration in the OPEN or the implicit declaration in the MAP statement.

File sharing can be defined more precisely in RMS with the ACCESS and ALLOW clauses, but there are problems here as well. For instance, if a file is opened ACCESS MODIFY, but the user does not have write privilege to the file, an access violation will occur regardless of the fact that the program may never attempt to perform a write. The access violation occurs at file OPEN time, not at processing time. This problem came to our attention because this is the default access value. If in conversion this clause is omitted, as happened in our case, error messages about access violations occurred that seem unwarranted in the RSTS meaning of the phrase.

FIELD statements may be retained, but it is often wiser to convert to MAP statements. Since RMS de-blocks records, then the role of the dynamic FIELD statement is much reduced. However, there really seems no good way to define a record buffer dynamically. Both the FIELD and the MOVE are slow, and MAP's will not accept variables in their size specifications since they are used to generate absolute addresses.

Furthermore, VAX-11 BASIC is more rigid about having file organization dictate legal file activity and coding techniques. For instance, in RSTS it was possible to have both a DIMENSION and a FIELD statement for a virtual array. In VAX-11 BASIC this generates an error at execution time.

Because many of our systems were converted to RMS ISAM files the concept of a record pointer was modified. For these files, the Record File Address must be used. In order to determine the RFA of a record, a special clause of the OPEN statement must be employed, the USEROPEN clause which references a MACRO program that is executed as part of the OPEN process. This routine then can determine the address of the Record Attribute Block for the file, and this can be used to determine the RFA of a record whenever the file is read. Files of RFA's are produced by the system SORT utility. These pointers, instead of being suitable for integer virtual core arrays as were logical record numbers on RSTS, go into sequential files with six-byte records. Reference to random records in these files, for instance in order to re-start a report, becomes a loop of FIND or GET statements rather than the much more straightforward subscripting we used to be able to do.

We feel some concern in our use of RMS at this time, for it is very clear that optimal use of this technique is not yet completely understood. We don't know the best blocking factors and bucket sizes, how often to revise the files so internal links are maintained in the optimum order for retrieval, and in general we don't have a "feel" for the files. In addition many of the utilities provided on VMS for the processing of RMS files are compatibility mode images and therefore can not handle the full functionality of RMS-32.

OTHER AREAS OF DIFFERENCE

Chaining

The CHAIN statement is implemented in a limited form. No line numbers are allowed. We found that well over 90 percent of our programs that used chaining did so to lines that were not at the beginning of the program. In fact RSTS/E standards encourage this practice. To make sets of these programs work together, we turned all such main programs into subroutines called by a main program. The function of the main program is to "direct traffic", by calling the routine specified in some common area. Each program so called is responsible for checking the line number at which execution is to begin and branch to the correct line. In a sense this is the way we would have written these programs had we had the address space to play with in the first place, but it did cost us time to study each program and find out what the initial branching table should be.

BATCH Processing

Batch processing on VMS is substantially different than it is on RSTS systems. On RSTS a batch job is run at a pseudo-keyboard. On VMS, there is no such thing as a pseudo-keyboard. Instead, two separate files are used to store I/O for the program. These are the process-permanent files known as SYS$INPUT and SYS$OUTPUT. One consequence of this is that none of the data input to your program is echoed in the batch log, which is the file SYS$OUTPUT. If you wish to have this dialogue appear in the log, then the program will have to be modified to detect if it is running in batch and echo the data itself. This means that a conditional PRINT statement has to be added for every INPUT statement in your programs.

A further problem associated with batch processing is that many programs open "KB:" on some channel so as to avoid the "?" prompt when in conversation with the user. In this case, the prompt is usually performed on the same channel from which the input is taken. This works fine interactively but fails in batch because there is no keyboard at all. The file cannot be opened on SYS$INPUT because you cannot do output to that file. SYS$OUTPUT cannot be used for a similar reason. The only solution is to split the prompting away from the input file.

Debugging

Debugging a program has some new features. Since VAX-11 BASIC allows subroutine calls, these subroutines must be compiled separately. These object files may be loaded by BASIC

and the main program run. Essentially BASIC is performing some of the tasks of the linker. Subsequent to this, immediate mode statements (even things like GET with a key) may be issued. One problem with immediate mode is that being a compiled language, no new variables may be created by an immediate mode command. Very simple programs written to test an idea may fail in immediate mode because sufficient temporary variables have not been generated by the program code itself. Furthermore, a program cannot continue after being stopped if program changes were made.

Routines not written in BASIC cannot be loaded by BASIC. Programs using such subroutines must be debugged using the VAX-11 Symbolic Debugger. The Debugger allows the programmer to examine and change data, set breakpoints and trace the flow of the program. It is very similar to the debugger provided with BASIC-PLUS-2.

SYS Calls

Some SYS calls are implemented. These are:

CODE	FUNCTION
0	CANCEL CONTROL/O
2	ENABLE ECHO
3	DISABLE ECHO
5	EXIT WITH NO PROMPT
6	CALL FILE PROCESSOR
—23	FILE STRING SCAN
—13	SET PRIORITY (ONLY PRIORITY)
—10	FILE STRING SCAN
—7	CONTROL/C TRAP ENABLE
9	GET VAX BASIC ERROR MESSAGE
10	ASSIGN A DEVICE
11	DEASSIGN A DEVICE
12	DEASSIGN ALL DEVICES
13	MESSAGE SEND/RECEIVE
14	MESSAGE SEND RECEIVE (CANNOT GET JOBNUMBER, ETC)
7	GET CORE COMMON
8	PUT CORE COMMON
9	EXIT AND CLEAR PROGRAM
11	CANCEL TYPEAHEAD

Many of the things determined by other function calls can be developed using MACRO subroutine calls or direct calls to VMS system services from BASIC.

There were other problems relating to differing philosophies between the two systems, VMS uses a FORTRAN-like carriage control. The sequence is line-feed, data, carriage-return. RSTS has the sequence data, carriage-return, line-feed. Because of the VMS sequence, it is easy to overwrite data on the keyboard if printing on several keyboard channels with semicolons.

VAX BASIC defaults all terminal format files to a width of 72. If longer print lines are to be formed, the MARGIN command is necessary to override automatic wrap-around after 72 characters. Alternately, the RECORDSIZE clause, which is RSTS compatible, may be included in the OPEN statement.

VAX-11 BASIC enforces a much stricter policy regarding error processing, and entry into and out of DEF*'s. (Both DEF*'s and DEF's are present in the language.)

SUGGESTED MIGRATION PATH

If given the time, those intending to migrate to VMS should consider the following steps:

1. Convert all existing BASIC-PLUS programs to EXTEND mode. Code all existing programs in EXTEND mode using the ampersand continuation character rather than line-feed.
2. Convert BASIC-PLUS to BASIC-PLUS-2.
3. Abandon Record I/O and convert applications and files to RMS. The saving in program size alone warrants this approach. All record I/O files will have to be converted, at least to VIRTUAL organization.
4. Avoid extreme system dependent features imbedded in the SYS calls.
5. Good practices, such as using standard modules, become a great boon because the modules have to be converted only once and then re-appended.

PHYSICAL TRANSFER OF PROGRAMS AND DATA FROM THE PDP-11 TO THE VAX

We experienced only a few problems with the physical transfer of data between the systems. The most annoying of these dealt with the fact that line terminators for RSTS include the form-feed and line-feed characters. Because of this the program FLX will not transfer these files as ASCII files but requires the use of image mode to move them. This causes an extra line terminator to be inserted every 512 characters. This terminator must then be edited out later. RUNOFF produced DOC files were particularly bad in this regard, but we also experienced problems with many programs. We had no problem with programs written in extend mode using the ampersand as the continuation character. Raw data moved easily by PIP'ing the files to tape, and FLX'ing them from tape onto the VAX. The resultant file organization is virtual with 512 byte records, just what is necessary to handle virtual core arrays or unblocked records with FIELD statements. In fact, virtual core arrays of binary floating and fixed point data moved with no difficulty.

CONCLUSION

In summary, VAX-11 BASIC is a very fast compiler and run-time environment. We have found that it is quite comfortable to use, and in a sense it is the best of both worlds, since it is both semi-interactive and compilable. It is certainly miles ahead of BASIC-PLUS-2. The immediate mode editor bypasses the need for using an editor such as TECO for small corrections to programs from the Ready prompt. There is virtually no limit to program size, so programmers no longer need worry about that constraint. The result is algorithms that are more efficient of system resources and execute much faster. One is no longer limited to just BASIC in any program since subroutines written in any other native mode language may be called. Large libraries of general purpose mathematical routines thus become instantly available as well. The complete range of VMS system services is available to all users.

Our experience has been positive and we recommend the language and the system highly.

I/O from Macro -- quickly and easily!

By Bob Meyer

The following article won't be a lesson in PDP-11 assembly language, but rather a basic description of how to get started writing Macro code to run on your favorite RSTS system. I'm sure you've heard a lot of evil things about Macro; "it takes too long . . .", or "Macro code can only be maintained by long-haired weirdos with leprosy", or my favorite, "You can't do I/O from Macro; only Basic can do that!".

Well, I hope that by the end of this article, I will have changed your mind about some (if not all) of these horrors.

To get right to the meat of the matter, let's start off with some I/O. (I/O????) — Yeah! This first macro module prints to the current kb:, and will be used by other chunks of code, especially the FIRQB / XRB stuff. You may find it handy to have your copy of the System Directives Manual close by; (I keep mine under my Beer . . .).

Before we go any further, let's look at two very important items in the monitor; the FIRQB (File Request Queue Block) and the XRB (Transfer Request Block). These two beauties exist in your Virtual workspace at locations 402 (FIRQB) and 442 (XRB) [by the way, in Macro Land, we default to Octal unless otherwise noted]. These memory locations are set aside by RSTS for User/Monitor communication (and should never be used for any other purpose!). When a User program issues a Monitor Directive (an EMT), the Monitor, through the Black Magic of memory Management, gets his paws on that User's FIRQB and XRB. With the contents of one or both of these blocks, and by looking at the EMT code that was issued, RSTS is usually able to determine what it is you are trying to do. When the request is completed, the monitor returns (posts) certain information back to the user's FIRQB and/or the XRB (depending on the type of call). Pages 2-12, 2-13 of the Directives Manual show the general layout of both blocks. (I also recommend browsing through COMMON.MAC for more details.)

Now let's get back to our Print program.

Take a look at Fig. 1; the first 30 lines or so are simply Documentation (wow!) and a few common monitor definitions so we won't need to assemble with any other files (i.e., common.mac). The actual execution starts at statement 29 with the label 'PRINT:' (the .END statement selects the program's transfer address).

Basically all we're doing here is as follows:

1) Get a handy Pointer to the XRB (R0)
2) Move the Length of the message to be printed into the first two words of the XRB (XRLEN XRBC) (the length in this case was calculated at line 24)
3) Move the Address of the message into the XRB (XRLOC)
4) Set the Channel number (times two) to zero for this kb: (XRCI)
5) Set Block number to zero (XRBLK)
6) Set Wait Time to zero (only used for kb: input)
7) And Clear all modifiers (XRMOD)
   (like, "Put 0%, Record 1%" for Binary I/O)
   (or, "Get 1%, Record 32767%+16384%" for Multi TTY)

Now the XRB is loaded, we're ready to call RSTS to do the I/O. This is done at statement 38 by the '.WRITE' directive, after which we exit to the system default RTS.

When the Processor executes the .WRITE EMT, it switches to Kernel mode, and traps thru 30 (don't get scared . . .). A few microseconds later, we end up running in the EMT Phase of the monitor, where our .WRITE actually becomes a request to the Terminal Driver.

The program in this example is quite runnable (6C or 7.0), and I encourage those interested to key it in and take it for a spin. Assuming you called this critter PRINT.MAC, it can be assembled as follows:

```
MAC PRINT,PRINT=PRINT
  ;THE SECOND 'PRINT' IS THE .LST FILE
```

And task built:

```
TKB PRINT,PRINT=PRINT
  ;THE SECOND 'PRINT' IS THE .MAP FILE
```

Then just 'RUN PRINT'

Another concern I hear from non-Macro types is that of privilege problems. ("That Macro program can access anything in the system!!") Trust me, folks, Macro programs have NO special access rights to anything! The program is running in User mode (unless you get crazy and mess with the I/O page, but that's another article) and has whatever privileges the user has normally. Non-prived people can write and execute Macro code and can do no more harm to a system than an equivalent Basic-Plus program. I would advise beginners, when trying out monitor calls from privileged accounts to be very careful; it's very unlikely that you would bring down RSTS, but you might KILL someone else's job instead of changing a password by putting something in the wrong place in the FIRQB. Keep in mind, the Assembler assumes Octal numbers unless told otherwise, and the Directives Manual gives FIRQB XRB descriptions in Octal also.

For those who plan to get more involved in Macro programming, I would suggest the following documentation:

The Processor Handbook for your machine,

Macro-11 Reference Manual,

RSTS System Directives Manual

This should get you started (and this is only the beginning!)

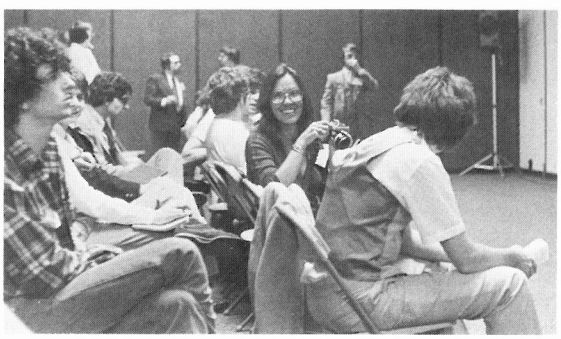
Good Luck!

```
.TITLE  PRINT   PRINT DEMO
.IDENT  /1.0/

;
;PRINT.MAC -- SIMPLE EXAMPLE OF PRINTING TO
;THE USER'S KB: IN MACRO
;
;4/25/80 BOB MEYER
;

;
;DEFINE SOME CONSTANTS
;
XRB     =       000442                  ;START OF TRANSFER REQUEST BLOCK
.WRITE  =       104004                  ;RSTS 'WRITE' DIRECTIVE
.EXIT   =       104046                  ;RSTS 'EXIT' DIRECTIVE

;
;DEFINE ASCII MESSAGE
;
.ENABL  LC
MSG:    .ASCII  /Hello from Macro land !/<15><12>
.EVEN
MSGLEN  =       .-MSG                   ;CALC LENGTH OF MESSAGE

;
;START OF MAIN CODE
;
PRINT:  MOV     #XRB,R0                 ;POINT TO THE XRB
        MOV     #MSGLEN,(R0)+           ;LOAD XRB WITH MESSAGE LENGTH
        MOV     #MSGLEN,(R0)+           ;TWICE
        MOV     #MSG,(R0)+              ;LOAD ADDRESS OF MESSAGE
        CLR     (R0)+                   ;CHANNEL # * 2 TO PRINT
        CLR     (R0)+                   ;BLOCK # TO PRINT
        CLR     (R0)+                   ;WAIT TIME FOR INPUT ROUTINES
        CLR     (R0)+                   ;OPTIONAL MODIFERS

        .WRITE                          ;CALL RSTS TO DO THE WRITE

        .EXIT                           ;EXIT TO DEFALT RTS

        .END    PRINT
```

FIGURE 1.

The 11/44 that couldn't. At Chicago 1980.

We got Joyce! Did Joyce get us?



A BASIC-PLUS-2 PROGRAMMER'S GUIDE TO RESIDENT LIBRARIES

By Al Cini, Computer Methods Corporation

While they evolve along distinctly different paths, RSTS/E and RSX occasionally intersect in some commonly implemented function or feature designed to promote compatibility between them. The RSTS/E V7.0 flavor of "Program Logical Address Space" monitor directives parallel their RSX-11 counterparts, offering the BASIC-PLUS-2 programmer expanded control of the **physical** relocation of portions of his or her application programs within computer memory. Provided monitor support for resident libraries was selected at system generation*, you can take advantage of this new feature in two ways:

1. You can use Taskbuilder commands to build and "link" to resident libraries. Only a very little bit of non-threatening assembly language coding is required — the Taskbuilder does most of the work.

2. You can execute .PLAS monitor calls to attach and map resident libraries. This approach requires some moderately demanding MACRO, but offers great flexibility of control over user memory.

This article will treat the Taskbuilder procedures required to create and use resident libraries, including all the assembly language needed to create "resident common" memory areas. While some discussion of their possible uses will be offered, a detailed discussion of .PLAS directives is deferred to your copy of the "RSTS/E System Directives Manual." Material presented in this article is intended as a supplement to standard RSTS/E documentation. Some previous familiarity with Taskbuilder procedures is assumed, and the reader is referred to the "RSTS/E Taskbuilder Reference Manual" for in-depth treatment of specific commands.

What is a resident library?

Shared memory segments — regions of physical computer memory which appear to be a part of one or more active user jobs — are not new to RSTS/E. RSTS has made good use of such structures for years: they're called run-time systems. A really lucid description of how RSTS/E relocates, or "maps," user programs, run-time systems, and resident libraries within physical computer memory (including a truly inspired drawing worth many thousands of words on the topic) can be found in Chapter 2 of the System Directives Manual. Briefly, a RSTS/E job's "logical" memory consists of its (optional, if disappearing RSX) run-time system (HISEG), its task image, and up to 5 optional resident libraries (even more, provided they're not all "attached" by a job at the same time).

In the BP2 environment, the run-time system may be RSX, BP2COM, BASIC2, or RMS11. Compiled code and data will comprise the task image, which can access either or both of the "standard" resident libraries (RMSRES/RMSSEQ and BASICS), as well as other user-built libraries.

In the BASIC-PLUS environment, a user job will consist of a flavor of BASIC-PLUS run-time system (determined when it is initially built) and a "work area" containing semi-compiled instructions and data. The total memory taken by all simultaneously mapped software components cannot exceed 32KW (this is the infamous "32K limit"). BASIC-PLUS-2 programmers are well aware that, within this limitation (it may be even more strict, depending on the local SWAP MAX), BP2 program size can be traded off against execution-time performance by selecting from among a somewhat confusing assortment of HIESEGs, ODL files, and RESLIBs in taskbuilding their jobs.

* The "disappearing RSX" option, while not a strict necessity, is a practical requirement for BASIC-PLUS-2 resident libraries.

It's important to note that, when assembling memory segments into jobs, RSTS/E considers user memory in 4KW "pages." A user job can control up to eight of these pages at a time, and the Taskbuilder deals in pages when connecting separate memory components to your task (see figure 1). The system "maps" these pages into physical memory by loading address relocation information into the job's 8 Active Page Registers (APRs) using mechanics which are totally transparent to the user. An important practical ramification of this "page" architecture is that any resident library or run-time system will occupy multiples of 4KW user memory chunks. Thus, a stripped-down 14KW BASIC-PLUS run-time system will **not** allow a program size of 18KW; 16KW is still the maximum. Likewise, a shared common area containing a one word status flag will gobble up another 4095 words of user memory when it's attached to a job. Note that **physical** memory is allocated by RSTS/E in smaller 1KW chunks, so small memory segments (such as the 3KW RSX run-time system) occupy memory regions rounded up to the next 1KW in size.

Like a run-time system, a resident library exists within a distinct, named region of physical computer memory defined when it is ADDed to the system (normally via the UT ADD LIBRARY command). Like a run-time system, a resident library may be mapped into more than one user job at a time (i.e., may be "shared"), and may contain subroutines, program sections, or other "global" symbols which may be referenced transparently by an associated task. Unlike a run-time system, a shared library contains no job control structures (such as a pseudo-vector or keyboard monitor), must be loaded at a specific physical memory address, and may be "attached" or "detached" dynamically by a user program. The RSTS/E implementation of resident libraries indeed closely parallels its run-time system mechanism. If you re-name BP2COM.RTS, as PB2COM-.LIB, it can then be ADDed and "linked" to your BASIC-PLUS-2 program (your HISEG **must** be disappearing RSX to manage this) just as though it were a resident library.

Things that go into resident libraries.

A resident library may contain **only** code (such as RMSRES and BASICS), only data (this is a "shared common" area, and can be used for high-speed inter-job communication), or some specially tailored combination of code and data.

A shared library containing code is considered to be **re-entrant** if the software within it can be executed simultaneously by more than one task (e.g., RMSRES, BASICS). BASIC-PLUS-2, like most high-level language processors, does **not** generate re-entrant code; resident libraries containing BP2 subroutines thus **can't** be shared by multiple jobs.

Whether it contains code and/or data, a resident library may either be **position-dependent** or **position-independent**. A library is position-independent if it can be accessed successfully regardless of its placement in a user's logical memory; any of user APRs 1-7 may be used to map a position-independent library. A position-dependent library, on the other hand, must always be loaded at a certain user memory address, which corresponds to a "base" address specified to the Taskbuilder when the library is constructed (position-independent libraries are taskbuilt with a base address of 0). RMSRES, for example, is a position-independent library. All run-time systems, because they are explicitly built to occupy the highest user-memory addresses, are position dependent.

In practical BP2 terms, a user program accesses a position-dependent library via global symbol (i.e., subroutine entry point) names, while a position-independent library may be entered by subroutine reference and/or referred to by COMMON or MAP name. Since BP2 compiled code is position dependent, BP2 resident subroutine libraries will be position dependent as well. A main program can communicate with such subroutines via argument lists, but not through MAP or COMMON areas. Two or more BP2 jobs can communicate with each other at super-fast core speeds through an easy-to-build "shared common" resident library, which must be position independent and, hence, can't contain any user-written BP2 subroutines (as we'll see later, elements from BP2COM.OLB can be incorporated into a shared common area to "flesh it out" to a 4KW boundary and conserve user address space).

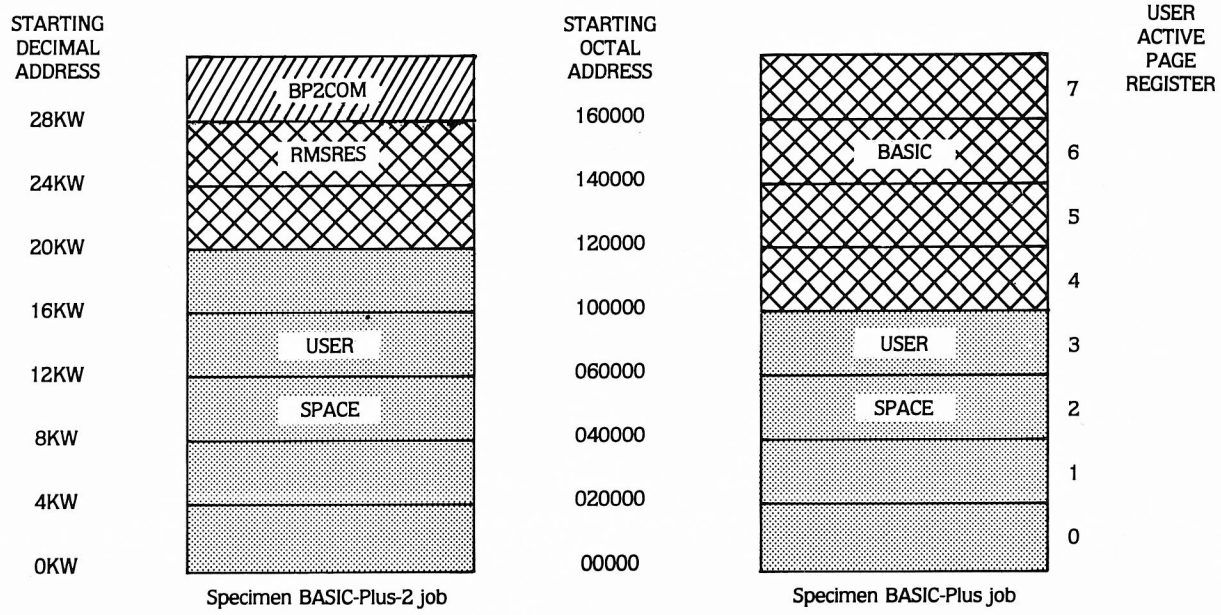


Figure 1. "Page" organization of a RSTS/E job.

Building a resident library of BP2 subroutines.

Consider the following simple BP2 program modules:

```
OLD MAIN

BASIC2

LISTNH
100       ! THIS SIMPLE ROUTINE CALLS TWO SUBROUTINES: &
          ! &
          ! STRING          RETURNS A RANDOM STRING OF 10 &
          !                 ALPHABETIC CHARACTERS.          &
          ! &
          ! STRREV          TAKES A CHARACTER STRING ARGUMENT &
          !                 AN REVERSES IT CHARACTER BY &
          !                 CHARACTER. &
          ! &
!
110       CALL STRING (X$)
120       PRINT X$
130       CALL STRREV (X$)
140       PRINT X$
150       GO TO 110
32767     END

BASIC2

OLD STRING

BASIC2

LISTNH
1         SUB STRING (X$)
100       X$=''
110       X$=X$+CHR$(RND*26+65) &
                FOR I%=1% TO 10%
32767     SUBEND

BASIC2

OLD STRREV

BASIC2

LISTNH
1         SUB STRREV (X$)
100       Y$=''
110       Y$=Y$+MID(X$,I%,1%) &
                FOR I%=LEN(X$) TO 1% STEP -1%
120       X$=Y$
32767     SUBEND

BASIC2
```

A resident library (MEMRES) containing the STRING and STRREV subroutines can be built with the following TKB command files:

(MEMRES.CMD:)

```
SY:MEMRES/-HD,MEMRES,MEMRES=MEMRES/MP
PAR=MEMRES:120000:60000
STACK=0
//
```

(MEMRES.ODL:)

```
        .ROOT USER
USER:   .FCTR STRING-STRREV-LIBR
LIBR:   .FCTR LB:BP2COM/LB
        .END
```

Once constructed, the main BP2 program can link to the subroutines in MEMRES as follows:

```
RUN $TKB
TKB>MAIN=MAIN,LB:BP2COM/LB
TKB>/
ENTER OPTIONS:
TKB>RESLIB=MEMRES/RW
TKB>UNITS=12
TKB>ASG=SY:6:7:8:9:10:11:12
TKB>EXTTSK=512
TKB>//

Ready
```

Note that a user program can be taskbuilt to a resident library which is not ADDed. if you refer to MEMRES.CMD, you will note that we have requested a task image, a load map, and a symbol table via our "MEMRES/-HD, MEMRES, MEMRES" command line. The symbol table file (MEMRES.STB) contains address information for STRING and STRREV needed to resolve the CALLs to these subroutines from the main program. The RESLIB option in the above TKB sequence directs the Taskbuilder to MEMRES.STB for this linkage data, and the Taskbuilder generates the required .PLAS directives to map our MEMRES library. RESLIB will accept a [p,pn] designation within the library name field, thus allowing users to link to resident libraries under other system accounts.

If we were to run MAIN at this point, we would get a confusing "? Can't find file or account" error message. This is caused by our Taskbuilder-generated .PLAS directives failing to attach to MEMRES, which we have yet to load into memory. Since only specially SIL-formatted files can be ADDed, we need to use the MAKSIL program to build MEMRES.LIB (MAKSIL is not built automatically during SYSLIB BUILDing — you may need to copy MAKSIL from your distribution kit):

```
Ready

RUN (1,8)MAKSIL
MAKSIL  V7.0-07
Resident Library name? MEMRES
Task-built Resident Library input file <MEMRES.TSK>?
Include symbol table (Yes/No) <Yes>?
Symbol table input file <MEMRES.STB>?
Resident Library output file <MEMRES.LIB>?
MEMRES built in 5 K-words, 247 symbols in the directory
MEMRES.TSK renamed to MEMRES.TSK<40>

Ready
```

(Refer to your "RSTS/E Programmer's Utilities Manual" for an in-depth description of MAKSIL.)

MEMRES can now be ADDed:

```
RUN $UTILTY
UTILTY  V7.0-07
#ADD LIBRARY [1,201]MEMRES<0>/ADDR:100/RW/1USER
#EXIT

Ready
```

This command ADDs MEMRES from the specified account (the default is [0,1]; if the .LIB file is under any other account — even your own — a [p,pn] **must** be specified) into main memory, beginning at physical address 100KW. Since BP2 subroutines "write" on themselves, the /RW switch is needed. Our specified protection code is 0, which allows all users read/write access to the MEMRES memory area. This code can be tailored to protect a shared library in much the same way as a disk file (see the "System Manager's Guide"). The /1USER switch will deny all access to MEMRES for any more than one user at a time. Since BP2 subroutines are not re-entrant, this avoids the probable bizarre and random consequences of simultaneous multiple-user execution. (Should this occur inadvertently, MEMRES — which may be corrupted in the process — should be REMOVED and freshly ADDed. The /REMOVE switch will cause a fresh MEMRES copy to be loaded each time a new job attaches to it; in this BP2 context, /REMOVE is a good idea.)

Refer to the previously presented MEMRES.ODL file, and notice that subroutines from BP2COM are linked into MEMRES. Memory resident libraries must be "self-contained" — they cannot call subroutines in their referencing tasks. If you were to specify map file output for MAIN.TSK, however, you would note that BP2COM routines used by MAIN that are already linked into MEMRES are **not** copied from BP2COM into MAIN.TSK. Instead, the Taskbuilder resolves references to these library routines into MEMRES, making it a "mini-BASICS" as well as a user subroutine area.

Note the "PAR=MEMRES:120000:60000" option within MEMRES.CMD. This option "names" the shared region, and specifies that it should be built beginning at user address 120000 (20KW) for a length of 60000 octal bytes (12KW). This particular specification precludes the use of any HISEG (other than disappearing RSX), but can be modified to make room for one. Note that the TKB "% Task exceeds memory limits" warning message accompanies any attempt at building 10 pound programs in 5 pound bags. You can refer to the map file to determine the necessary base address/length adjustments, then re-TKB the library.

Memory-resident overlays.

Single-user-only subroutine libraries offer few advantages to the BP2 programmer; if not for memory-resident overlays, there would be little point in discussing resident libraries of BP2 subroutines. Consider the following revision to MEMRES.ODL:

```
        .NAME X
        .ROOT X-USER
USER:   .FCTR *!(STRING-LIBR,STRREV-LIBR)
LIBR:   .FCTR LB:BP2COM/LB
        .END
```

As with disk overlays, the STRING and STRREV subroutines are assigned to the same user logical address. Unlike disk overlays, global address reference to these overlaid program segments is accomplished via memory mapping directives rather than disk READs from a task image file. This overlay approach requires the following modified MEMRES.CMD:

```
SY:MEMRES/-HD,MEMRES,MEMRES=MEMRES/MP
PAR=MEMRES:120000:60000
STACK=0
GBLREF=STRING
GBLREF=STRREV
//
```

Note that less user space is required (in this simple example, the saving is negligible), and that our subroutine names must be explicitly declared as global references. This is required because, by "overlaying" STRING and STRREV on top of an empty "dummy" root segment (X), we eliminate the automatic placement of global references to STRING and STRREV into MEMRES.STB by the taskbuilder. Note also that BP2COM.OLB modules, since they are not in the root of our overlay structures, can't be shared by the main program. While we've saved some space by overlaying, we've forced an extra 3KW of BP2COM routines into MAIN.TSK. Referring to our load map, we can include these BP2COM OTS routines into the root of our MEMRES overlay library by brute force:

```
        .NAME NLROOT
        .ROOT NLROOT-LIBR-USER
USER:   .FCTR *!(STRING,STRREV)
STRING: .FCTR SY:STRING-LB:BP2COM/LB
STRREV: .FCTR SY:STRREV-LB:BP2COM/LB
LIBR:   .FCTR 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-LIB1
LIB1:   .FCTR A1-A2-A3-A4-A5
A:      .FCTR LB:BP2COM/LB:$FADD
B:      .FCTR LB:BP2COM/LB:$STFN1
C:      .FCTR LB:BP2COM/LB:$FCON1
D:      .FCTR LB:BP2COM/LB:$STCOS
E:      .FCTR LB:BP2COM/LB:$JNCR
F:      .FCTR LB:BP2COM/LB:$ERROR
G:      .FCTR LB:BP2COM/LB:$JMOVS
H:      .FCTR LB:BP2COM/LB:$STMOS
I:      .FCTR LB:BP2COM/LB:$FMUL
J:      .FCTR LB:BP2COM/LB:$JNEXT
K:      .FCTR LB:BP2COM/LB:$FRAND
L:      .FCTR LB:BP2COM/LB:$CALLS
M:      .FCTR LB:BP2COM/LB:$STGTA
N:      .FCTR LB:BP2COM/LB:$STMSC
O:      .FCTR LB:BP2COM/LB:$IEULT
P:      .FCTR LB:BP2COM/LB:$JCONV
Q:      .FCTR LB:BP2COM/LB:$ICEOL
R:      .FCTR LB:BP2COM/LB:$BFPER
S:      .FCTR LB:BP2COM/LB:$BBTKS
T:      .FCTR LB:BP2COM/LB:$BXTRA
U:      .FCTR LB:BP2COM/LB:$STLSS
V:      .FCTR LB:BP2COM/LB:$CNTRL
W:      .FCTR LB:BP2COM/LB:$CALLR
X:      .FCTR LB:BP2COM/LB:$BINIT
Y:      .FCTR LB:BP2COM/LB:$IVOPN
Z:      .FCTR LB:BP2COM/LB:$ICULT
A1:     .FCTR LB:BP2COM/LB:$ICROP
A2:     .FCTR LB:BP2COM/LB:$ICRCL
A3:     .FCTR LB:BP2COM/LB:$ICFSS
A4:     .FCTR LB:BP2COM/LB:RQLCB
A5:     .FCTR LB:BP2COM/LB:SAVRG
        .END
```

This maneuver at once reduces the physical size of MEMRES by 3KW (BP2COM appears only once within the root segment of MEMRES, rather than once in each overlay segment), and the logical size of MAIN.TSK by 3KW (because MAIN.TSK can now use BP2COM routines contained in MEMRES). As the number of overlay segments increases, this device can become very important indeed.

Of course, these "memory-resident overlay" libraries cannot be shared, and depending on program size can demand large regions of precious physical memory. Certain "number crunching" applications involving the invocation of very large subroutines within program loops can be constructed using memory resident overlays, and will run faster than a corresponding disk-overlay implementation. (Read the resident library chapter of the Taskbuilder manual before striking off to convert your number-crunchers; there are annoying restrictions to using AUTOLOAD with memory resident overlays in a resident library which make direct translation of disk overlay specifications somewhat risky).

Shared common.

Probably the most useful application of resident libraries in a BP2 environment is the "shared common" area. Using shared common, a MAP or COM in one program can be overlaid onto MAP or COM regions of the same name in one or more other programs — thus, separate RSTS/E jobs can exchange large volumes of data at memory cycle speeds.

The following trivial assembly language program defines a four-byte program segment named LINK:

```
.TITLE TEST
.PSECT LINK,D,RW,GBL,REL,OVR
.BLKB 4.
.END
```

The .PSECT MACRO directive is analagous to the BP2 MAP statement. The .BLKB directive allocates a "block" 4 bytes long (the "." after the 4 specifies a decimal rather than octal byte count).

Assuming that PIP or an editor were used to build TEST-.MAC containing the preceding MACRO program, a shared common area by the name TEST.LIB can be built as follows:

```
RUN $MAC
MAC  > TEST = TEST
MAC  > ↑Z

Ready
```

```
RUN $TKB
TKB  > TEST/-HD/PI,TEST,TEST = TEST
TKB  > /
ENTER OPTIONS:
TKB  > PAR = TEST:0:0
TKB  > STACK = 0
TKB  > //

Ready
```

```
RUN (1,8)MAKSIL
MAKSIL  V7.0-07
Resident Library name? TEST
Task-built Resident Library input file <TEST.TSK>?
Include symbol table (Yes/No) <Yes>?
Symbol table input file <TEST.STB>?
Resident Library output file <TEST.LIB>?
TEST built in 1 K-words, 0 symbols in the directory
TEST.TSK renamed to TEST.TSK<40>

Ready
```

Two sample BP2 routines which can communicate via TEST:

```
OLD WRITER

BASIC2

LISTNH
10        ! WRITER ... THIS ROUTINE WRITES RANDOM NUMBERS &
          ! INTO THE "LINK" SHARED COMMON AREA. &
          !
100       COM (LINK) X
105       X=RND
110       X=RND &
                  UNTIL X=0
32767     END

BASIC2

OLD READER

BASIC2

LISTNH
10        ! READER ... THIS ROUTINE DISPLAYS THE CONTENTS OF THE NUMBER &
          ! STORED IN "LINK" WHENEVER IT CHANGES. &
          !
100       COM (LINK) Y
110       Z=Y
120       IF Z<>Y THEN &
                  PRINT Y &
\                 Z=Y
130       GO TO 120
32767     END

BASIC2
```

(Note: these routines must be compiled with the /NODOUBLE switch, if double precision floating point format is the compiler default.)

These programs can be linked to TEST.LIB with the following taskbuilder commands.

```
RUN $TKB
TKB>WRITER=WRITER,LB:BP2COM/LB
TKB>/
ENTER OPTIONS:
TKB>RESLIB=TEST/RW
TKB>UNITS=12
TKB>/
TKB>READER=READER,LB:BP2COM/LB
TKB>/
ENTER OPTIONS:
TKB>RESLIB=TEST/RO
TKB>UNITS=12
TKB>//

Ready
```

After TEST.LIB is added to the system, run READER and WRITER simultaneously using two terminals (you can "DETACH" WRITER, if you like). Notice that adjusting the run-burst of these jobs significantly affects the READER display rate (you may need a high-speed CRT to see a difference). This is because both jobs are compute-bound at the same priority, and will be scheduled on a "round robin" basis as their run bursts expire.

Transfers between jobs — particularly involving large data volumes — are substantially faster via shared common than by way of message send/receive. Also since transfers are **direct** from task image to task image, monitor overhead is completely by-passed and system loads are correspondingly lightened. In such "transaction processing" applications, however, the programmer assumes responsibility for coding the "handshaking" necessary to synchronize jobs communicating through a resident common. In our simple example, a change in the state of the contents of a shared MAP is used to "signal" printing within the READER job. In general, however, two or more compute-bound jobs "looking" for changes in state indicator variables will impose quite a system burden. Alternatively, the availability of data for processing in a shared MAP can be signalled by sending a small interjob message, and completion of processing can be acknowledged in the same manner.

The Taskbuilder commands we used to link WRITER and READER to the shared common area TEST include provisions for specifying the type of access required (/RW in the RESLIB command requests read-write access; /RO specifies read-only access). This specification is made independently of the protection code and /RW switch specified when the common area is ADDed, and consequently access violations are handled differently by the system depending on whether the error is detected when the accessing program is initially loaded ("?Protection violation" immediately after RUN), or while it is running ("?Memory management violation" during program execution).

As an added suggestion, a shared MAP area, associated with an RMS file opened in one job, can be interchanged with another job which doesn't use RMS. Certain RMS-intensive applications which could benefit from non-overlaid RMS (or RMSRES), but can't afford the user space, might consider an "outboard RMS" processor routine which, upon request, READS/WRITES/UPDATES RMS files using record information contained in shared MAPs.

When building a shared common area, remember that the .PSECTs you define in the shared library must be at least as large as the MAP or COMMON areas they re-define within a referencing task (TKB will warn you with an "%Incompatible reference" message if they aren't). Also, remember that whenever you make a change to the size of one or more .PSECTs within a resident common, it **and** all of its referencing tasks must be re-built.

Incorporating OTS routines within a resident common area.

Consider our TEST.LIB shared common area. Note that, because we want to reference it by COM or MAP area equivalence, we need to Taskbuild it with the /PI (position-independent) switch. Remember that, since our MEMRES BP2 resident subroutine library was position-dependent, we could pass arguments to STRING and STRREV but could not pass information to them via MAP or COMMON. Also note that, because resident libraries occupy 4KW multiples of user address space our TEST shared common area occupies 4096 words of user space, **while actually making use of only 2 of them.** In practice, few MAP areas will align perfectly to 4KW boundaries, and the resulting wasted space may often prove prohibitive.

By playing a few Taskbuilder games (borrowing from the re-build procedures for the BASICS shared library-check your BP2 Installation Guide and BP2 distribution kit), we can incorporate commonly used BP2COM.OLB routines into our shared common area to "flesh it out" and make better use of the space within it.

The procedure is simple:

1. Compile a sample BP2 program (one of the referencing tasks for your planned resident common is a good bet).

2. Taskbuild it and obtain a .MAP file. For example:
   ```
   RUN $TKB
   TKB > WRITER,WRITER=WRITER,LB:BP2COM/LB
   TKB > //
   ```

3. Get a listing of the .MAP file, and look for the heading **SECTION** on the left near the top of the first page. Note the heading **TITLE** on the right side of the same line.

4. Under the heading **SECTION**, look for
   BP2OTS:(RW,I,LCL,REL,CON).
   Entries in this column are in alphabetical order, so BP2OTS should be close to the top.

5. BP2OTS is the name of the .PSECT into which all required BP2 system subroutines are collected. In the **TITLE** column under this BP2OTS entry is a list of modules which were required by your sample routine.

6. Use Taskbuilder "BP2COM/LB:" to include these modules from BP2COM into your shared common area. Using WRITER and TEST, here is an example command file:

```
TEST/-HD/PI,TEST,TEST=TEST
LB:BP2COM/LB:$CNTRL
LB:BP2COM/LB:$ICEND
LB:BP2COM/LB:$ERROR
LB:BP2COM/LB:$FMOV
LB:BP2COM/LB:$FRAND
LB:BP2COM/LB:$TESTS
LB:BP2COM/LB:$IEULT
LB:BP2COM/LB:$IVOPN
LB:BP2COM/LB:$BINIT
LB:BP2COM/LB:$STMSC
LB:BP2COM/LB:$ICRCL
LB:BP2COM/LB:$JCONV
LB:BP2COM/LB:$ICEOL
LB:BP2COM/LB:$BFPER
LB:BP2COM/LB:$BBTKS
LB:BP2COM/LB:$BXTRA
LB:BP2COM/LB:$ICROP
LB:BP2COM/LB:$ICFSS
LB:BP2COM/LB:$STGTA
LB:BP2COM/LB:$ICULT
LB:BP2COM/LB:RQLCB
LB:BP2COM/LB:SAVRG
/
PAR=TEST:160000:20000
STACK=0
//
```

7. Obtain a program called RESSTB from your BP2 VI.6 distribution kit, and run it as follows:

```
RUN  (1,8)RESSTB
ENTER FILE SPECIFICATION FOR RESIDENT LIBRARY ? TEST
ENTER 3 CHARACTER CCL/MCR NAME ? BP2
THE PSECT FOR YOUR LIBRARY IS BASOTS.

Ready
```

(Always answer "BP2" to the "ENTER 3 CHARACTER CCL/MCR" prompt, whether or not the CCL name you specified when you built BP2 VI.6 was "BP2").

While our TEST.MAC program is position-independent, BP2COM OTS routines are not. We therefore, use **both** the /PI switch **and** non-zero base and length values in the PAR command to, in effect, build a curious hybrid resident library which has a position-independent symbol table (including PSECT addresses), but which is linked to a specific user memory address. Since .PSECT names within shared libraries (with position-independent .STBs) must not be the same as any in a referencing task (except, of course, for the shared COM/MAP), RESSTB is used to "tweak" the symbol table file entry for BP2OTS, changing it to BASOTS. **Note that the alphabetical order of PSECT names within the symbol table must be preserved** — your shared library must not use any MAP/COM names between BASOTS and BP2OTS. Also, modules included from BP2COM must be listed in the same order as they were found in the .MAP file.

8. The TEST shared library may now be ADDed and linked in the manner outlined earlier.

****Remember:** these procedures are release-dependent. They work with BP2 VI.6, but may not with previous or later releases.**

Omission of the RESSTB step will result in "LOAD ADDRESS OUT OF RANGE" errors when Taskbuilding tasks against shared common area containing OTS routines. Also, any "undefined global reference" errors accompanying the TKB for a shared common are probably the result of omitting or misspelling a required OTS module name.

.PLAS directives.

So far, we've confined our discussion of resident libraries to the Taskbuilder commands and options required to build and link to them. While a programmer in this environment is totally unaware of the dynamics involved, the Taskbuilder, in fact, uses .PLAS directives to accomplish this linkage. Armed with some knowledge of MACRO and the .PLAS monitor call formats, a systems programmer can dynamically map a resident library up to 32KW in size, mapping it 4KW at a time (use smaller segments, if you like — the mapping can take place beginning at any 32 word boundary for a minimum length of 32 words). Such a program can, in effect, directly address 60KW or more of user space to implement some exotic "virtual array" scheme for use in some scheduling or array processing application (remember, a job can "map" as many as five resident library areas at once: that's access to more than a quarter megabyte of space). All the information needed can be found in your "MACRO Programmer" and "Monitor Directives" manuals (refer to your "BP2 User's Guide" for information on calling user-written MACRO subroutines).

CONCLUSION

The Taskbuilder is slow. It takes forever to assemble a runnable core image. Now, at last, with RSTS/E V7.0, the Taskbuilder offers you something to read and think about ... while it's Taskbuilding. Have fun.

C H A N G E S ? ? ? ? ? ?

Are you changing addresses? Please let us know so you won't miss one issue of the *RSTS Professional.*

FORMER	**PRESENT/Near Future**
Name _____	Name _____
Address _____	Address _____
City/State _____	City/State _____
Zip _____	Zip _____

 . . . continued from page 25

with special instructions with respect to procedures to be followed with specific routines or documents. In fact, one can have a separate library entitled HELP. (To access the library, the library document is changed to [diskette].HELP, from the Editor Menu, which automatically will change the library document.)

SETTING UP THE LIBRARY DOCUMENTS TO DELETE THE HARD RETURN

There are two methods available to avoid hard returns following a Library Document call. (I.e., where special formatting is required the formatting must follow the library field identifier in order to be imbedded.)

The first method, of course, is to have the library information begin immediately after the field identifier (<<field>>Data xxx).

The second utilizes the soft return described in preceding sections. The following is an example:

```
L-----------------------------------------R---
<<DOCUMENT>>
LT----------------------------------------R---
                      TITLE

                      (data)
<<
```

Without modification, if DOCUMENT is referenced by the library an extra return will result, as creation of the document necessarily required a hard return after the field identifier.

However, with the insertion of a new ruler (a dummy ruler is indicated, but it should represent a required format) the hard return can be changed to a soft return (by moving to the beginning of the line immediately after the field identifier and striking the RUB CHAR OUT key) and, when referenced, the soft return will be ignored. The imbedded ruler also will appear.

USE OF LIBRARY DOCUMENTS FOR EXTRA RULER STORAGE

Quite often the ten ruler storage availability of the Word Processing System is inadequate, either because more rulers are required or because it is difficult to remember which is which. There is an alternative.

Using the same technique for removing the hard return described in the foregoing sections, rulers can be saved in a library and can be called by document type. This especially can be helpful for unusual documents, but also is helpful for general documents. The following are examples of two rulers. Expansion of the concept is quite unlimited and, obviously, up to the individual user.

```
<<LETTER>>
L---P----T------------------H------J-----------
<<

<<SCHEDULE>>
L---T----T----<------.---------.----------.--->R
<<
```

Naturally, a schedule of all of the rulers can be part of the HELP library.

SPREADING A TITLE OR HEADING

One additional note. The soft return also can be used to spread a word, or series of words, accross an entire page. Provided that the RIGHT MARGIN is set with a J (justification), everything on the soft return line will be spread across the page if the next line commences with any type of an imbedded command (e.g., ruler, print control, page marker, etc.). Example:

```
L---P-----T-----T---------------H-----J--------
(Miscellaneous text to the next line)

                      T I T L E
L---T-----T-----T---------------H-----J--------
```

By creating a soft return (i.e., using the RUB CHAR OUT at the beginning of the first line after "TITLE" in the above example, the word "TITLE" will be spread accross the page as follows:

```
T          I          T          L          E
```

In order to create the "soft" return, it only was necessary to modify some portion of the ruler, imbed it and then delete the hard return with the RUB CHAR OUT from the beginning of the line.

SOME ADVANCED FEATURES FOR INDEXES (INDEXI?)

In a preceding section of this paper, we have discussed the manner in which an index or a Table of Contents can be created, using the List Processing features of the Word Processing Systems. Here we will discuss, briefly, two enhanced features -- including page numbers and sorting.

To include page numbers it is, of course, necessary to have a "finished" copy of the document, as you will have to insert the page numbers after each word to be referenced.

Using a "finished" copy, we mark each word which is to become part of the index. Using the techniques described above, we enter the field identifier before the word, or phrase, and follow it with a space and the page number. Then, the terminating left/right arrow and dummy field identifier follow the number. For example, we are indexing on the phrase "List Processing":

```
[Text ... <><I>List Processing 22<><X> ...]
```

In a long document, we do not attempt to identify each page where the word or phrase appears, but merely repeat the process.

Depending upon how involved the index may be, we will either run the List Processor on the <I> field and pick up every instance using a "sub list" which will be SORTed, or we will use a conditional run, based upon the first letter of the field creating a semi alphabetical listing.

[The "sub-list" is quite simple. It is identified as <<I><I>.]

If we create a long index, we then SORT the "sub-list" index using the SORT package which is available through your DEC WPS representative.

What's this? You say your rep doesn't know what you are talking about? If you have any difficulty, send this author two blank diskettes, a returnable mailer AND RETURN POSTAGE, and we will send you a copy of the SORT package and its instructions. Two diskettes are required as the SORT is in DIBOL and operates on COS-310, whereas the instructions are on a WPS diskette.

Upon completion of the SORT, it is a fairly simple matter to "cut" the repetitions and allow their page numbers to flow onto the first use of the term. (Using a RIGHT ARROW ">" as the right margin -- before the R or J -- will allow these numbers to flow backwards so you can have several numbers, with commas, on the same line.)

PERMANENT CUTTING OF LONG SECTIONS OF A DOCUMENT

As you may realize, it is not possible to "cut" more than about 2-1/2 pages of a document at one time. If you want to delete several pages, and do not have to save them for any other purpose, then this can be accomplished with a single operation without concern over the actual length of the material being deleted.

The procedure requires that you proceed to the starting point of the "cut" (which can be the end or the beginning of the "cut") and press the white SEL key. Then, immediately press the red CUT key. This will remove all data from the paste buffer.

Press the white SEL key again and proceed to the point where you wish to end the cut. Press the GOLD REPLC keys (GOLD and '). This will replace all of the data between the SELect position and your current place with a single null. The cut data is not replaceable so do not use this as a "cut and paste" routine. [For long "CUT AND PASTE" routines, refer to the section above which discusses changing your document to a library document.]

TRANSPORTING RULER AND PRINT SETTINGS TO NEW SYSTEMS

Often it becomes necessary to transport your ruler and print settings from one system diskette to another. There is a fairly easy way to accomplish this.

For the ruler settings, you merely create a single document using your old system diskette and then begin to enter ruler settings separated by some meaningful code so you can identify the settings when used on the new system. E.g.:

```
0
L---P----T----->----- ... -R
1
D---P----T-----T----- ... -R

etc.
```

Placing this document diskette under control of the new system diskette (and the document may, of course, reside on the new system diskette) you merely advance below each ruler, enter GOLD RULER and then SHIFT [number] to preserve the ruler. Continue with each ruler until all 10 have been transferred over (or as many as are needed).

NOTE: The same procedure can be used in your LIBRARY to save more than ten rulers, or to call rulers by document name. Just enter the RULER NAME between the arrows (e.g. <<LETTER>>) and follow the identifier with a return. Place the desired ruler under the identifier, then enter a single RUB CHAR OUT to remove the hard return and enter your end of field marker (<<). Now, when you call the RULER from the LIBRARY, it will appear where you want it without extra returns.

For PRINTING COMMANDS, create up to ten separate documents and merely file them away. Go into the PRINT MENU for each document (either as you create them and afterward) and call up your PRINT COMMANDS from your old SYSTEM diskette. Save the commands on the document by using the GOLD MENU. Placing the diskette with your "new" SYSTEM diskette, you merely begin to print each of the documents. When in the PRINT MENU enter SS nn to save your PRINT COMMANDS as reflected on the particular document.

The procedure is also available for special printing requirements on documents where there are insufficient printer commands (i.e. -- 10) available. Just identify a document with the name of the particular type of printer commands you want to save, and you always can assign that to any of the numbers you wish for special purposes. (We find that reserving control number 9 for this purpose works out just fine.)

SUMMARY

DEC's Word Processing Systems (and even those which utilize DEC equipment) clearly are among the most powerful available on the market today. The potential -- indeed the need -- for improvements is all too obvious, if DEC intends to remain a serious contender for the Word Processing Market.

In the meantime, there are numerous routines which are available in the existing system which can make it work better and faster for you, and that is what automated word processing is supposed to be all about.

The examples provided here are but a few of the many work saving features which are available. It appears that these examples have never previously been documented by DEC, which really is a shame.

We do hope, however, that the foregoing will be of some assistance to the Word Processer user and that this Article may become part of your Word Processing Manuals.

CLASSIFIED

Send Classified Ads to: RSTS Classified, P.O. Box 361, Fort Washington, PA 19034. ($1.00 per word, first 12 words free with one year's subscription.)

CALL/370 TO RSTS CONVERSION AID—Object file on Mini-Reel with Documentation. $50. Call or write to: TCI, 6107 W. Mill Road, Flourtown, PA 19031, (215) 836-1406.

RSTS INTERNALS, TUNING, RT11/RSX MACRO under RSTS consultation & Programming — call MACRO MAN, Bob Meyer, 609-298-9127.

WATCH THIS SPACE for Software Announcements from Interactive Software Systems, Denver, Colorado (303—771-0939).

HELP start Detroit area RSTS SIG. Call Keith Kapera at (313) 559-0890.

$200 RSTS/E* WORD PROCESSOR
CBEDIT.BAS

Single Basic-Plus* program with CRT input, window edit and file save. Add, locate, change, replace, delete, block move & copy, standard paragraph append, etc. VT* series and Hazeltine terminal drivers. Others easy to add.

Fully formatted output (margins, justify, center, underscore, headers, page numbers, etc.) to terminal, disk or line printer. Bi-directional driver for Diablo. We use it daily for all our secretarial work.

9-Track $200 RK05 $260 ppd
T.F. Hudgins & Assoc., Inc.
P.O. Box 10946, Houston TX 77018
Woods Martin — 713/682-3651
*TM Digital Equipment Corporation

16K DEC 8E $4500.00. 312—584-6622.

RIMS/MPG, BASIC-PLUS/DIBOL source code generator/report writer. Call 412-373-0500.

MEN'S CLOTHING MFG. on RSTS interested in PGM System Exchange. S. Ruby, (514) 382-5000.

ARM Systems Implementers announces: Data Management Tools, CRT Program Generators and Menus. 5190 NW 167 St., Miami, 33014.

WANTED: W.I.S.E. replacement. Bill Roberts, Pasadena, CA, (213) 449-1745.

DEC PDP 11/20 for sale 4K like new $2500. Computer Timesharing Serv., Nashville, TN (615) 244-6094.

DIBOL SUBS and STRUCTURED PRE-Processor. Bob 314/725-2750, Clayton, MO

Desire contact with insurance application software users. (800/535-8796 Ray Enright)

List of Advertisers

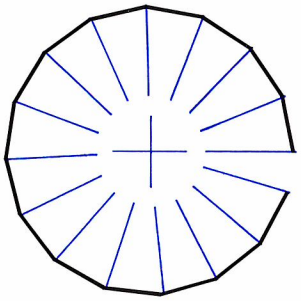
What's the difference between BASIC and Pascal?

COMPARE THESE APPROACHES TO DRAWING A CIRCLE

in BASIC

"This is easy..."

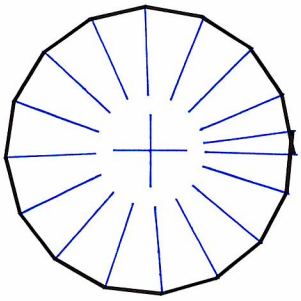
```
100 MOVE R,0
110 FOR T=0 TO 360 STEP 25
120 DRAW R*COS(T), R*SIN(T)
130 NEXT T
```



"Oops, didn't quite meet ...

... but that's easy to fix."

```
100 MOVE R,0
110 FOR T=0 TO 360+25 STEP 25
120 DRAW R*COS(T), R*SIN(T)
130 NEXT T
```



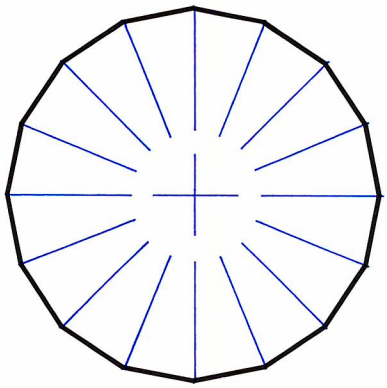
"Oh, now it closes...
in fact, it overlaps."

Programming by trial and error

in Pascal

"The simplest circle drawn with line segments is a regular polygon..."

```
procedure Circle (X, Y, Radius: real);
const Sides = 16; Pi = 3.14159265;
var   N:integer; Theta : real;
begin
  Move (X+Radius,Y);
  for N:= 1 to Sides do begin
    Theta:= 2 * Pi * (N/Sides);
    Draw (Radius * cos (Theta) + X,
          Radius * sin (Theta) + Y);
  end;
end;
```



Programming by design

GET IT RIGHT THE FIRST TIME

INTERNATIONAL DISTRIBUTORS

Australia: Sydney;
Network Computer Services
290-3677

Canada: Vancouver;
Valley Software
(604) 291-0651

England: Stafford;
Hourds Computing Ltd.
0785-44221

Japan: Tokyo;
Rikei Corporation
03-345-1411

If you like the feel of precision tools, give us a call or return this coupon.

Oregon Software

2340 SW Canyon Road • Portland, Oregon 97201
(503) 226-7760 • TWX 910-464-4779

Name _____

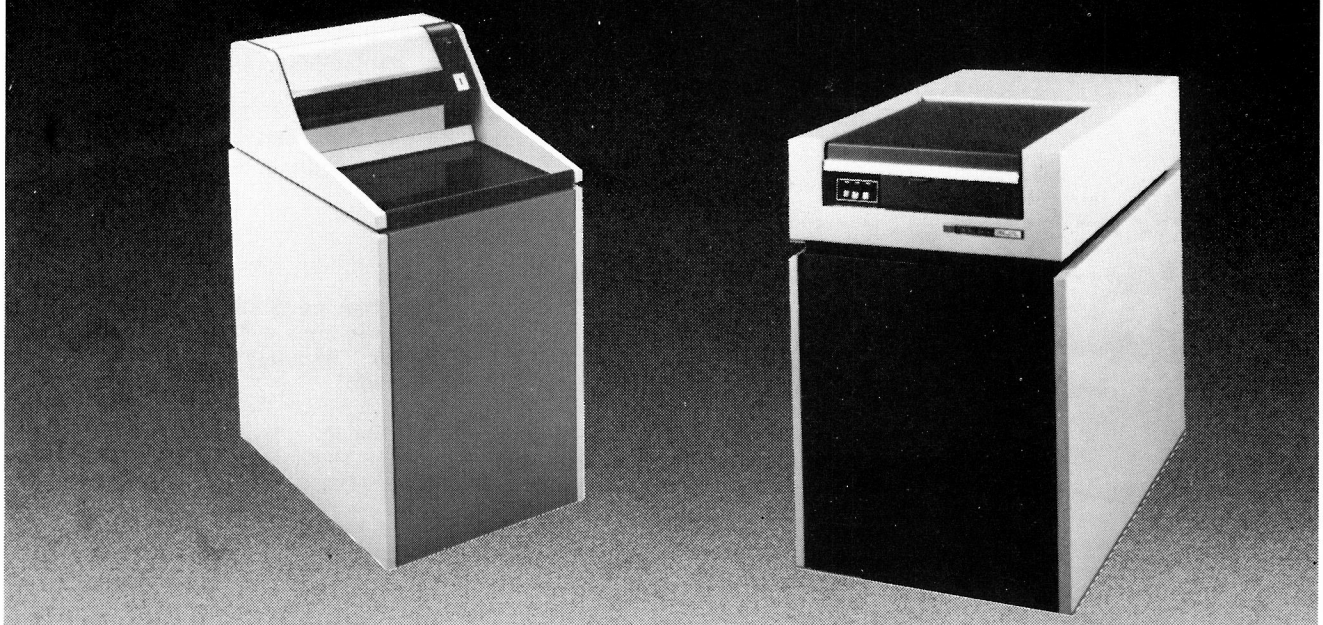
Firm _____

Address _____

City _____

State _____ Zip _____

Double Your Disk Capacity For Half DEC's* Price



The New Standard For PDP-11 Disk System Technology

Only System Designed Just For PDP-11 Family

Designed exclusively for DEC's UNIBUS or MASSBUS CPU's. On the UNIBUS, it's just one card that plugs into any spare SPC slot. On the MASSBUSS, four cards plug into any spare existing RH70 standard back plane.

Same Disk Drive as DEC RM02-03, RP06 and RM04-05

We use the same disk manufacturers as does DEC. The RP06 is the Memorex 677 200MB disk drive, the RM02-03 is the 9762 CDC 80MB and the new RM04-05 300MB is the CDC 9766. Only the LOGO is different.

Transparent to All DEC Software, Diagnostics and Drivers

You bet! Use your existing Software . . . no change needed. Runs all DEC's Diagnostics plus has its own. Fully emulates DEC disk Drivers.

Worldwide Installation and Maintenance

Through Data Systems Services, maintenance and installation is provided via Memorex or CDC for both Drive and Controller. We also offer full PDP-11 system support.

Full Media Compatible?

That's right! You can read or write on our drives. Put it on DEC's and it will play or vice versa. TRULY MEDIA COMPATIBLE.

Compare These Prices

DRP06	200MB Slave	$14,995
DRJP06	200MB + Controller	$21,995
DRWP06	200MB + Controller	$23,995
DRM02-3	80MB Slave	$ 8,995
DRJM02	80MB + Controller	$15,995
DRWM03	80MB + Controller	$17,995
DRJM04-5	300MB Slave	$16,995
DRJM04	300MB + Controller	$23,995
DRWM05	300MB + Controller	$25,995

Call us for all your DEC needs

DATA SYSTEMS SERVICES

22642 Lambert
Suite 408
El Toro, CA 92630
(714) 770-8024

*PDP and DEC are registered trademarks of Digital Equipment Corporation

Now you can make your PDP-11 run faster than ever. It's in the cards.

If you're looking for a way to supercharge your present PDP-11 instead of upgrading to the next computer, we can give you the boost you need. It's in the cards. It's in the cards we are about to put in your hand. It's in the single-board, cache-buffer and add-in memory cards we've already put on the table for people like you.

The biggest bargain in the industry is our SCAT/45, an add-in memory which installs up to a full 128K of high-speed memory (in 32K word increments) on the FASTBUS of the 11/45, 11/50 and 11/55 while using only ¼ the power and ¼ the space at about ¼ the cost of the DEC equivalent FASTBUS bipolar memory. SCAT/45 can increase your 11/45 processing speed up to 300%!

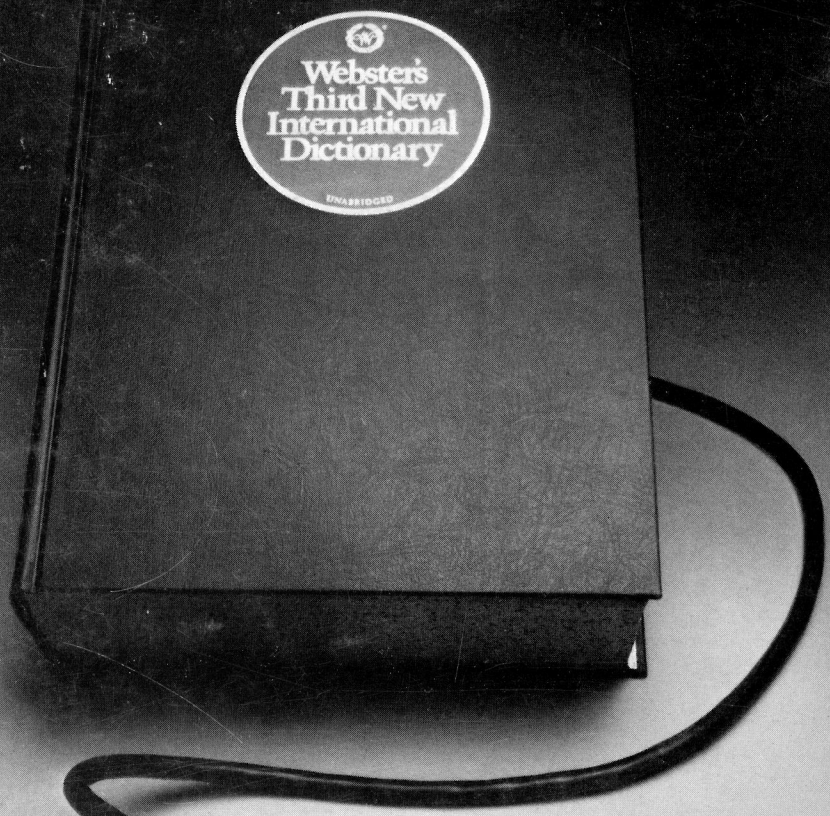
You'll be just as impressed with our incredible 8K cache. Available as CACHE/434 and CACHE/440 for the 11/34, 11/34A, 11/35 and 11/40, it is the 8K cache with everything. It has byte and address parity, upper/lower limit switches, on-line/off-line manual switch control, activity indicator lights, 8K bytes of memory (4K words) providing a capacity four times that of competitive units, and a backplane interconnect module replacement design which requires *no extra space* in your computer.

The incredible cache and companion buffer, CACHE/45, increase processing speeds as much as 100% in the 11/34, 11/34A, 11/35, 11/40 and 11/45. Find out how to get more out of your present PDP-11. Write for details on our complete line of memory, communications and general-purpose products. Able Computer. 1751 Langley Avenue, Irvine, California 92714. (714) 979-7030. TWX 910-595-1729.

Able, the computer experts

PDP, FASTBUS and DEC are registered trademarks of Digital Equipment Corporation.

Responsive Word Processing. Take Our Word For It.



• PDP-11

• Multi-terminal

• Concurrent data processing

• Low cost

WORD-11 is proven word processing power. Power responding to your needs. Designed to run on Digital's family of PDP-11 minicomputers, WORD-11 supports up to 50 inexpensive VT52 or VT100 terminals and uses a wide range of high speed and letter quality printers.

WORD-11 is productivity. And efficiency. By running concurrently with data processing, WORD-11 enhances the overall effectiveness of your system.

And WORD-11 is a variety of useful and unique features. Such as the multiple dictionary capability that detects and highlights spelling errors.

WORD-11 is also inexpensive. The per terminal cost of WORD-11 is much lower than similar systems. Whether as an addition to your current system or as a dedicated word processing system, the cost of WORD-11 is agreeably low. DPD can also provide accounting and utility software for your RSTS/E System. Call or write for information on our software or for details on turnkey systems. Ask for our free brochure, today.

WORD-11

Data Processing Design, Inc., 181 W. Orangethorpe Ave., Suite F, Placentia, CA 92670, (714) 993-4160.

Data Processing Design, Inc.
Specialists in Digital Equipment sales and software applications.

PDP-11, and RSTS/E are trademarks of Digital Equipment Corp., Maynard, MA.