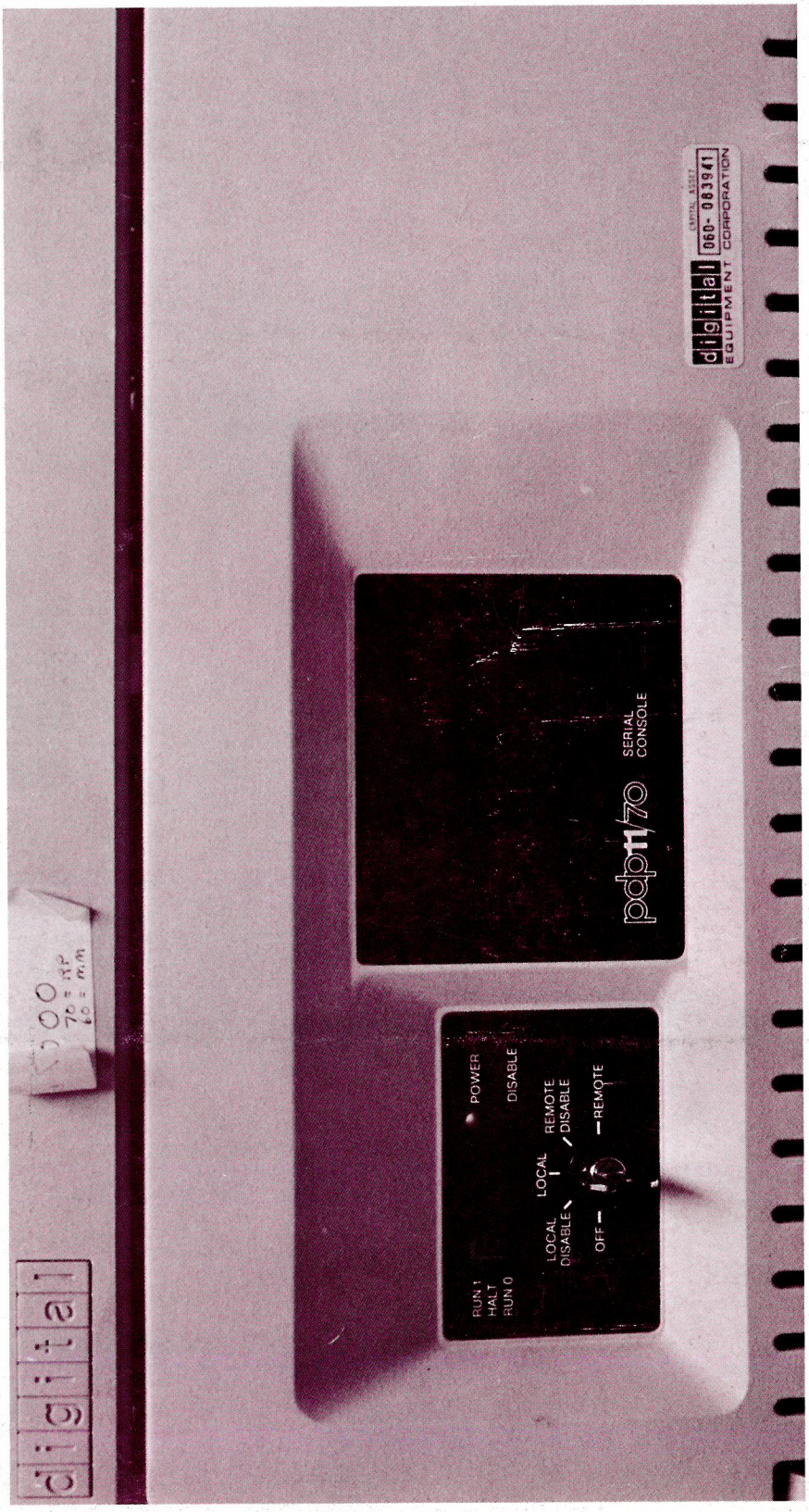
RSTS PROFESSIONAL

Volume 1, Number 1

November/December 1979

$7.50/issue, $20.00/year



INSIDE:

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

EDITORIAL POLICY OF THE RSTS PROFESSIONAL

To provide an independent forum for the RSTS community. This will be accomplished by:

1. Regular special interest sections. Specifically, we are planning, TECO, performance, systems programming (hacking), RPG, RT11, user profiles, COBOL, FORTRAN and others as requested (UNIX, PASCAL, etc.).

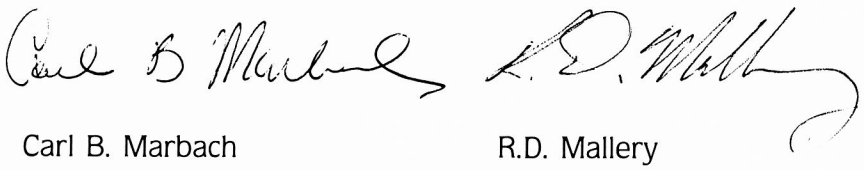
2. Articles of general interest. A regular (quarterly for now), reliable (we'll be here!) journal free from censorship by one group.

3. Product reviews and announcements. We will bring you an analysis of hardware and software products for your systems. The variety of these products, from many manufacturers, is both staggering and gratifying; at least the RSTS users aren't the only ones who believe in RSTS.

4. An editorial position and exchange of views. If we don't like something, we'll let you know — you do the same.

5. Through advertising, we'll provide a "marketplace" for RSTS products, all of them.

6. ??? We're flexible, and we believe it is really our (RSTS) journal, so let us know what you want and we'll try to provide it.



Carl B. Marbach R.D. Mallery

THE EDITORS

RSTS PROFESSIONAL*

Published quarterly by M. Systems, Inc., P.O. Box 361, Fort Washington, PA 19034.

Second Printing June 1980

Editors
R.D. Mallery
Carl B. Marbach

Editorial Assistant
Bonnie Staubersand

Contributors
Ronald Arenson, M.D.
Scott Banks
Peter Clark
Al Cini
Jack W. London, Ph.D.
Dave Mallery
Carl B. Marbach
Richard A. Marino
Martin Pring, D.Phil.
Kenneth Ross
Joel Schwartz, M.D.

Photographic Consultant
Arthur Rosenberg

Design & Typography
Grossman Graphics

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.)

*THIS PUBLICATION IS NEITHER PROMOTED, AUTHORIZED, NOR IN ANY WAY AFFILIATED WITH DIGITAL EQUIPMENT CORP.

Contents

In Next Issue...

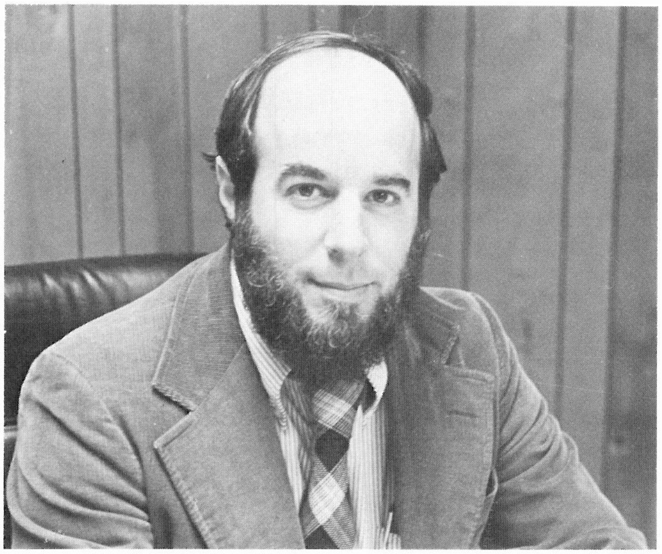
February/March 1980

- Everything You Always Wanted To Know About Modems and Multiplexers
- How to Install Your Own Peripherals
- Add-on Disks and Memory
- V7.0 Initial User Reactions and Reports
- Programming Standards
- Disk Directories
- How TECO?
- RPG Conversion from a System/3
- User Site Profile
- More . . .

From the editors . . .

ME AND YOU

By Carl B. Marbach



ME an editor! My English Comp 101 professor just woke up in a cold sweat. He doesn't know why, but we do: I'm an editor. When I was publishing articles, THEY were the editors, now it's ME.

I've got to confess that it's more than I thought it was. It's a heavy responsibility to YOU. Yes sir, I'm responsible and accountable to YOU, the RSTS professionals of the world. ME and YOU.

This journal will live or die by what YOU do with it. I'll help, but it's really for YOU and by YOU. RSTS needs YOU. I need YOU, to help ME.

I guess what I really want to say is that TOGETHER, WE can help each other, all of US. WE need to know more. WE need to know how. RSTS has more of US than WE know, and THEY need more, better, faster, and newer information.

This is an invitation to ALL OF YOU to participate in the exchange of information and knowledge in

The RSTS PROFESSIONAL.

Write! Articles, letters to the editors, love letters (I love my 11/70), hate letters (I hate to wait 9 months for an RP06), ideas, questions, or whatever. This exchange of information will benefit all of US.

Like I said, ME and YOU!

RSTS — A BEGINNING

By Dave Mallery

Great operating systems take on a life of their own in time. An exceedingly complex and interconnected organism, it does the unpredictable, organizes its own environment, and becomes charged with a vitality all its own. The earliest sign of RSTS's viability was its emergence as the prime commercial operating system for the PDP11 despite DEC's insistence that it was only for education. DEC soon found itself with a product they never really planned, setting sales records that were never forecast. A vocal and organized user community demanded and got release after release. The organism evolved new capacities by demanding them from its creator. Originally, RSTS meant BASIC+ and that was that. The BASIC interpreter was imbedded in the monitor. Constant pressure from the growing user community transmitted through DECUS finally caused the splitting-off of run-time systems from the monitor.

As usual, the organism refused to dance to the designers tune. Free at last from the constraints of BASIC, many expected RSTS to shed the interpreter like an old skin. But again, the user community resisted the change. Today's BASIC+ interpreter is better than ever. There is even serious talk of a BASIC+ for the VAX. BASIC+2 is a language on its own, but will never replace BASIC+.

Run-time systems (HISEG'S), once the domain of the select few, are now being written up and down the street. Since they offer real security for proprietary software, they are encouraging more entries into that market. They also provide the best possible environment for user and system security. We look forward to many exciting products in this area.

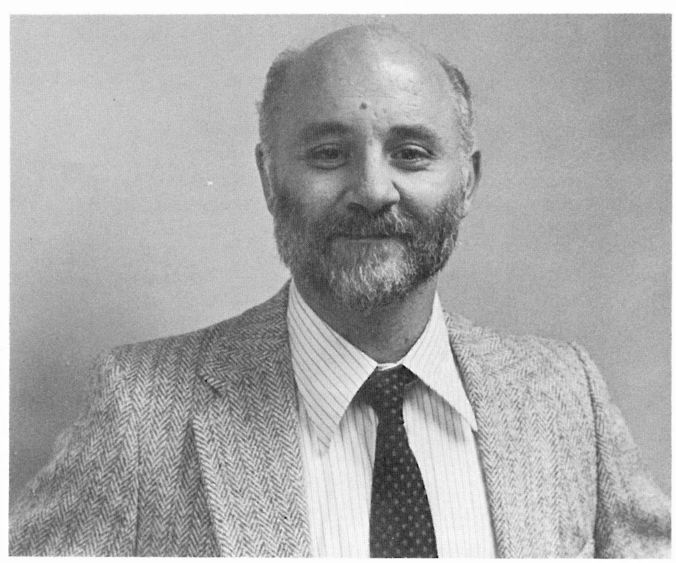
Full support of MACRO is a real milestone. Many of the best system software packages have their own HISEGS and are at least partially coded in MACRO, others use RT11 or RSX HISEGS.

Macro based cusps (PIP.SAV) are coming. The net performance enhancements provided by such maneuvers can greatly extend the usable life of a system.

I remember solemn pronouncements that 5C was the 'END of RSTS. Each succeeding release has been accompanied by the same dire tidings. Right now, for all to hear, I'd like to solemnly announce the

'BEGINNING OF RSTS'

as a mature, vibrant, viable operating system. We offer you the RSTS PROFESSIONAL as a forum. A forum is a meeting place, a place for politics and speeches, a place for intellegence gathering and a place to do business; a place to learn and a place to teach. A forum that is not crowded with people and full of noise and activity is a useless empty space. The RSTS PROFESSIONAL will be a forum only if you, the professionals, flock to it, use it, trade in it, teach in it and learn from it.



RSTS/E SOFTWARE ENGINEERING NOTES
— Structured Methods for BASIC-PLUS (-2) Programmers

STRUCTURED PROGRAMMING in BASIC-PLUS and BASIC-PLUS-2

By Al Cini, Nationwide Data Dialog

THE INSPIRATION FOR STRUCTURED PROGRAMS

Of each dollar spent on software engineering in the Data Processing industry today, more than 50 cents is channeled into the repair or adaptation of existing systems, and much less than the remaining half is spent on new development. If even a fraction of this cost could be reclaimed by the use of techniques which reduce the maintenance overhead of programs, the industry-wide savings would be enormous.

Theoretical papers in the late 60's by Dijkstra and others began an analysis of the fundamental nature of programs and programming, which eventually converged on "Structured Programming." This early treatment of elementary program architecture began as a series of essentially descriptive inventories of the **minimum** number of control structures needed to express a program. Eventually, these descriptive analyses were used to prescribe specific strategies for building "better" software, and, as such a set of normative rules, structured programming as it is practiced today gradually settled into the framework of the industry.

Of course, rules in Data Processing are rarely accepted without some fuss. Even now that most of the dust has settled, many a coder will bristle at the mention of "structured programmming," refusing even to read past the phrase in a book or journal. But as so-called "structured" computer languages (such as the widely adored PASCAL) are introduced and accepted, and as the discipline's vocabulary becomes a part of the industry's jargon, an eventual encounter with structured techniques is becoming an inevitable event in a coder's career.

THE MEANING OF STRUCTURE

The early work in structured programming centered around a hypothetical system of "correctness proofs" which, it was hoped, would provide a basis for providing programs correct before they get to the computer. The goal of such systematic "desk checking" wasn't so much the development of such a proof procedure (the application of which could be quite tedious for a large system) as it was the definition and understanding of "correct" software. Everyone observed that some programs, apparently more organized than others, would more readily lend themselves to this sort of process, so the preliminary focus of study shifted to collecting commonalities among organized programs. Largely by induction (though later these ideas would be refined and verified more rigorously), the following "earmarks" of organized programs emerged:

1. They appeared to comprise well-defined segments, which were connected at discrete points and did not randomly reference each other's components.
2. They tended to avoid obscure language constructs.
3. They were relatively easy to read.
4. They avoided unnecessary extremes of involution, such as excessive parenthetical nesting or multi-level IF-THEN.

Over time, these observations were refined and reduced to a few definitions which are the basic principles of structured programming:

A **program** is a representation of a procedure suitably defined to allow its automatic execution by a computer. All programs can be written using combinations of the following elementary structures:

Sequence. In a program sequence, control is transferred serially from the first statement (the top) to the last (the bottom).

Selection (IF-THEN-ELSE). In a program selection, the alternative execution of two distinct and separate program segments is based on the evaluation of a logical condition.

Iteration (DO-WHILE). In a program iteration, a program segment is executed repeatedly while an associated logical condition is true.

As a new era of respect for "common sense" dawns in our industry, we observe that English is a superset of FORTRAN (and all other programming languages), and not vice versa. It is therefore possible to render a plain English translation of any program, and often it is handy to do so in advance of actual programming. Structured programmers often discuss something called "pseudo-code," which is really an ad-hoc form of "pigeon-English" they use to express thoughts in a systematic way. Free of the idiosyncracies of particular programming languages, pseudo-code programs allow easy exchange of ideas among programmers and, eventually, can be converted to code in the target computer language by following a few rules. Pseudo-code representations of the elementary structures follow:

Sequence.

```
Statement 1
Statement 2
Statement n
```

Selection.

```
IF <condition—> THEN
        Statement A1
        Statement A2
        Statement An
ELSE
        Statement B1
        Statement B2
        Statement Bn
ENDIF.
```

Note: when the ELSE in a selection has no statements, it is sometimes parenthesized:

```
IF < condition > THEN
        Statements
(ELSE)
ENDIF.
```

Iteration.

```
DO WHILE <condition>
        Statement 1
        Statement 2
        Statement n
END DO.
```

Flowcharts, in a structured context, assume a somewhat new meaning. Each of the elementary structures has only one entry and exit, so flowcharts of the structures typically sport one 'in' arrow (at the top) and one 'out' arrow (at the bottom). Combinations of the structures, therefore, are easily partitioned into their basic components — a property not often found in traditional flowcharts. This "plug-to-plug" compatibility of program elements lead to the development of an alternative flowcharting technique (called Nassi-Schneiderman diagrams, or Chapin charts), offered as a contrast to traditional flowcharts below:

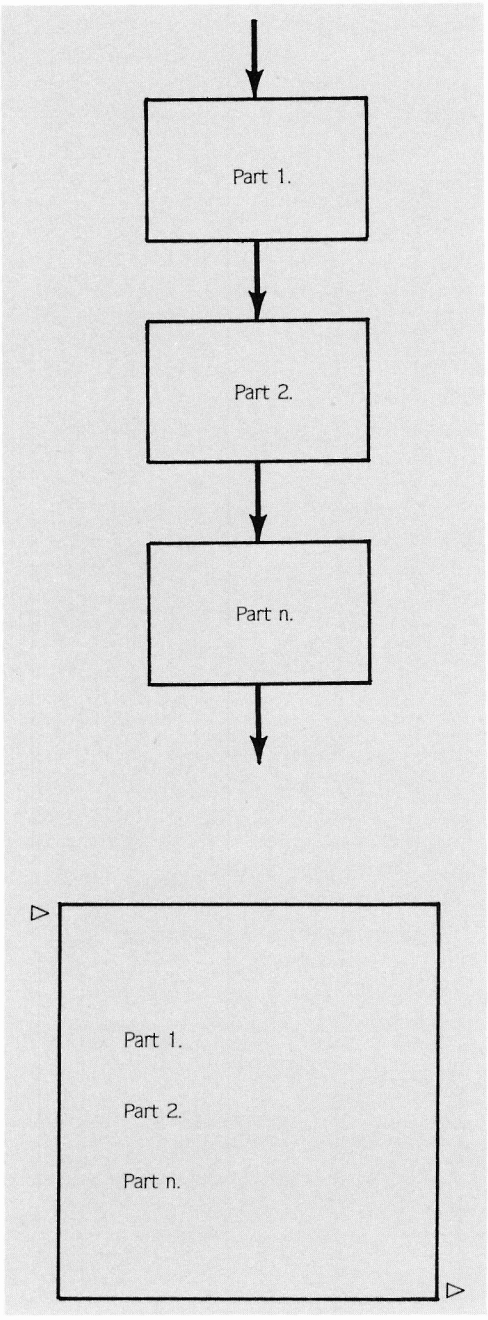


FIGURE 1. Sequence.

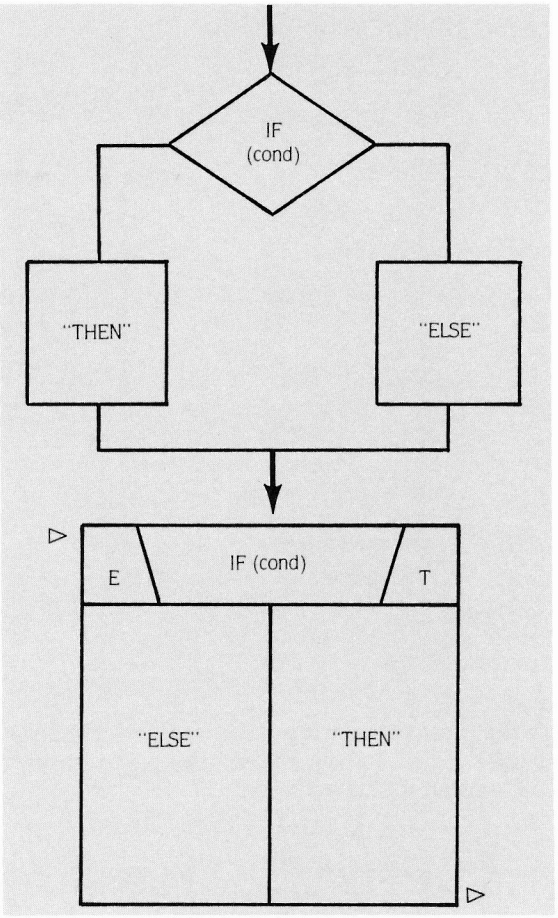


FIGURE 2. Selection.

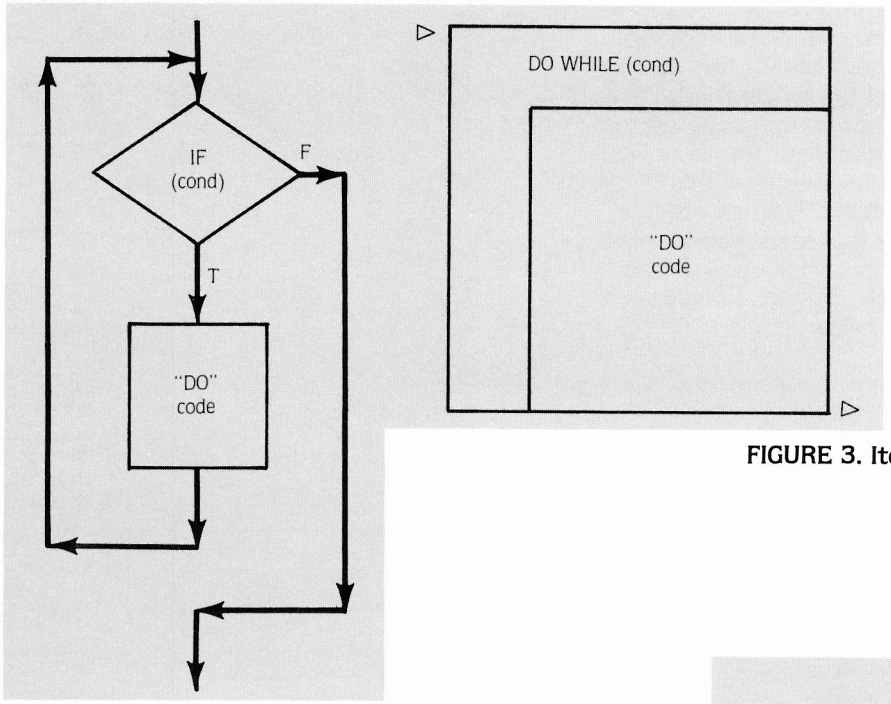


FIGURE 3. Iteration (DO-WHILE).

While the three elementary program forms presented so far have been proven to be sufficient, two additional structures are introduced for convenience:

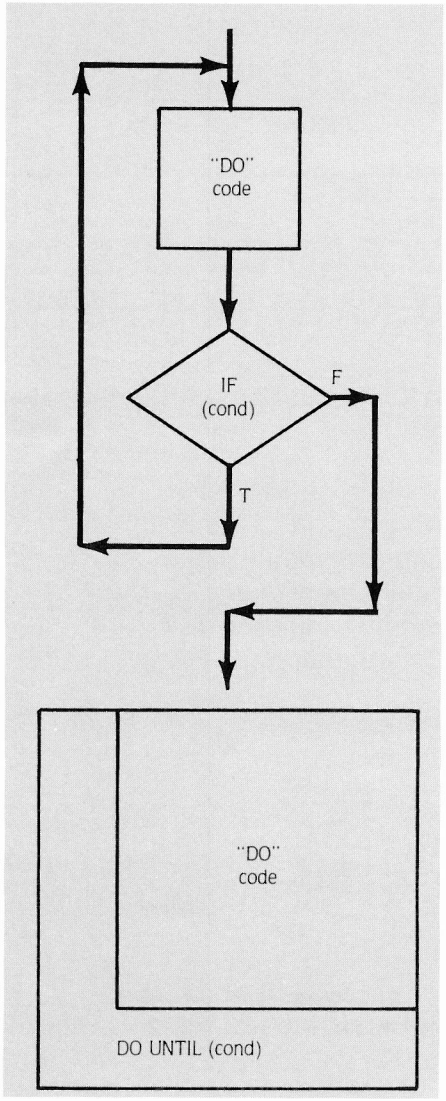


FIGURE 4. Iteration (DO-UNTIL).

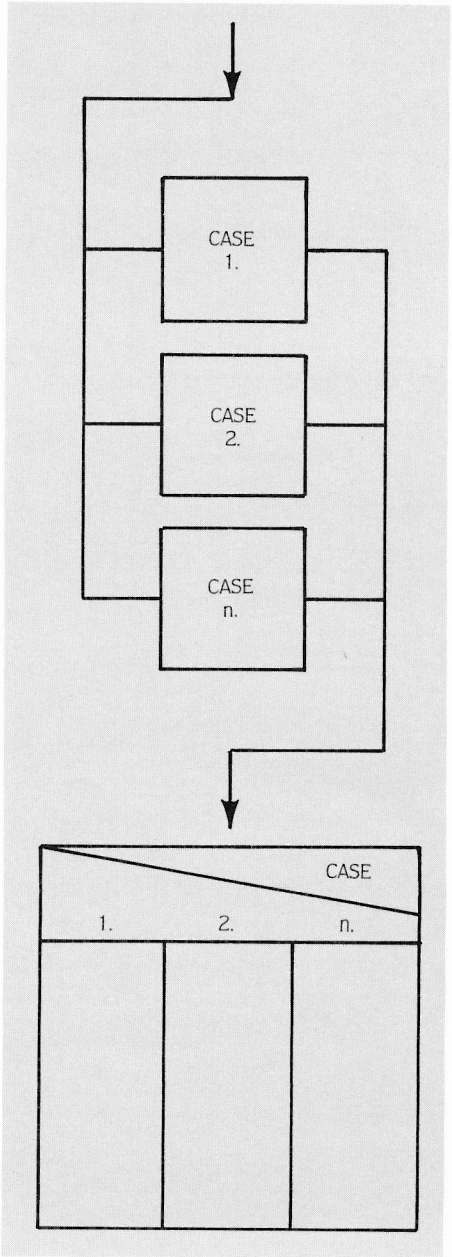


FIGURE 5. Case Selection.

Most people associate structured programming with notorious campaigns against the use of "GO TO" in programs and, since none of the idealized "elementary structures" includes a provision for unconditional branching, such an association is, in theory, accurate. In practice, however, people aren't programming in pseudo-code on Turing machines, and most commonly employed programming languages do not offer direct representations of the elementary structures (this is very true in FORTRAN, for example, and to a lesser extent in BASIC-PLUS). Current state of the art, alas, demands an occasional unconditional branch.

As a practical matter, building structured programs in "real" computer languages is accomplished through the canny use of traditional language elements to synthesize the elementary structures — now and then, these devices will demand a GO TO.

BASIC-PLUS Language Restrictions.

BASIC-PLUS is an easy-to-use, highly interactive language which offers rapid program development and fairly good run-time economy (for an interpreter). A wealth of software has been and is being written effectively in BASIC-PLUS every day. (Its compiled counterpart, BASIC-PLUS 2, isn't as interactive a language, but it offers a broadened syntax and extra features.) Nonetheless, there are two major restrictions inherent in this language which complicate the construction of structured software. Specifically, 1) BASIC-PLUS (and PLUS-2) statements are identified by numeric rather than symbolic labels (PASCAL offers both, numeric labels required only for the unnecessary GO TO), and 2) the BASIC-PLUS "IF-THEN" construct allows only one statement within the scope of THEN and ELSE (except for the last one on the line), and offers no "END IF" device. For the most part, these restrictions arise because, at the moment, neither BASIC-PLUS nor PLUS-2 is an entirely "block structured" language.

Nonetheless, structured programs can be written in these languages, but in doing so, we must be careful not to accept BASIC-PLUS language constructs which appear structured at their face value.

Structuring BASIC-PLUS Programs.

As we've already suggested, "block-structured" languages are easier to structure than non-block-structured languages. Simply stated, block-structured languages allow programmers to treat a series of program statements as though they were a single statement, and these 'blocks' of software can be substituted in-line for any single statement. PASCAL's 'IF' device, like BASIC-PLUS, allows only one statement in its 'THEN' and 'ELSE' clauses. Unlike BASIC-PLUS, however, a series of statements can be substituted for a single statement by bracketing them with 'BEGIN' and 'END' statements. Thus PASCAL allows

```
IF <condition> THEN
        BEGIN
                Statement;
                Statement
        END
ELSE
        BEGIN
                Statement;
                Statement
        END
```

BASIC-PLUS, of course, does not and, in fact, neither does PLUS-2. Even though BASIC-PLUS-2 allows multiple program statements in 'THEN/ELSE' clauses, extreme care must be taken when one of these statements is another IF. For example, the following pseudo-code can't be represented directly in BASIC-PLUS-2:

```
IF <condition—1> THEN
        A=0
        B=0
        IF <condition—2> THEN
                C=C+1
        ELSE
                C=C - 1
        ENDIF.
        Q=0             ! <—This poses a problem
ELSE
        A=1
ENDIF.
```

In this example, the statement 'Q=0' is handled in line with its predecessors 'A=0' and 'B=0.' Directly converting this statement to BASIC-PLUS-2 places 'Q=0' in line with the alternative statement for the innermost 'IF', viz. 'C=C - 1'. While this "dangling-ELSE" problem is easily side-stepped in a trivial case such as this (we'll consider some alternative approaches in the next examples), more complex expressions involving even shallowly nested 'IF' statements can be quite misleading.

In an earlier definition of block-structuring, we listed two criteria for truly block-structured languages: 1) That multiple program statements can be treated as a single statement, and 2) That such multiple-statement blocks can be substituted in-line for single statements or other multi-statement blocks. BASIC-PLUS and PLUS-2 meet the first criterion, but not the second. It is possible for a group of program statements in a BASIC-PLUS program to be treated as a single statement by bracketing them with GOSUB/RETURN statements, by including them in function definitions, or by compiling them separately as subroutines (PLUS-2 only, of course). An invocation of such a "block," via GOSUB, CALL, or function reference, is syntactically equivalent to a single statement; thus, we meet criterion 1 for block-structuring. Unfortunately, we can't take these groups and substitute them in place for the statements which invoke them. As a result, these 'paragraphs' (in this respect, they are similar to COBOL paragraphs) will end up in geographically separate places in the listing. Our BASIC-PLUS rendering of the previous pseudo-code problem, using out-of-line blocks, becomes:

```
1000    IF <condition—1> THEN &
                GOSUB 2000      &
        ELSE                    &
                A=1             &
        ! ENDIF.
2000    A=0                     &
        / B=0                   &
        IF <condition—2> THEN &
                C=C - 1         &
        ELSE                    &
                C=C - 1         &
        ! ENDIF.
2010    Q=0                     &
        RETURN                  &
```

With a little rearrangement, it is often possible to express such constructs using in-line code. To further pursue the previous example:

```
1000   IF NOT  <condition—1>  THEN  &
               A=1                  &
       ELSE                         &
               A=0                  &
               B=0                  &
               IF <Condition—2> THEN &
                     C=C+           &
               ELSE                 &
                     C=C - 1        &
!              ENDIF.
1010   Q=0     UNLESS  <CONDITION—1>
```

It is possible, using out-of-line code blocks, to convert selective pseudo-code directly to BASIC-PLUS and BASIC-PLUS-2. Without a pre-processor of some kind (and one wonders about pre-processors for interpreters), however, the resulting listing can be unwieldy. Rearrangements of program logic, particularly involving the use of statement modifiers, can preserve the structure of a program without sacrificing its clarity. With practice, these "work-arounds" evolve as matters of unique style which, nonetheless, are maintainable across programmers.

The sequence and iteration structures are easily represented in BASIC-PLUS and PLUS-2 syntax. Unless an error is trapped (we'll discuss this later), any collection of executable statements which do not contain branching instructions will constitute a sequence, and any collection of statements bracketed (or modified) by "WHILE" or "UNTIL" will constitute a "DO-WHILE" loop. The "case selection" structure, which is really a syntactically convenient nested "IF-THEN" device, has as its most direct BASIC-PLUS analog the "ON <scalar> GOSUB" statement, which executes one of a vector of out-of-line coding blocks based on a given scalar value. ("ON <scalar> GOTO", inasmuch as it fails to offer automatic return to its following statement via "RETURN" from its out-of-line blocks, is a less suitable substitute.)

Note, however, that although BASIC-PLUS provides an "UNTIL/NEXT" loop control mechanism, it does not behave at all like the "DO-UNTIL" structure. Instead, both "WHILE" and "UNTIL" loops in BASIC-PLUS evaluate their specified conditions **before** the code within them is executed (this is often called a "leading" decision). The DO-UNTIL structure, on the other hand, tests its associated condition **after** its code is executed (a "trailing" decision). Thus, in a DO-UNTIL, the scope of the loop will always run at least once when the loop is activated, but in BASIC-PLUS "UNTIL" and "WHILE" loops the enclosed code may never run at all. Indeed, "WHILE <condition>" and "UNTIL NOT <condition>" are equivalent in BASIC-PLUS and BASIC-PLUS-2.

"DO-UNTIL" can be expressed in BASIC-PLUS by using combinations of "IF-THEN" and "GOTO", or by using "UNTIL (or WHILE)/NEXT." The following program segments will search through DATA statements in a program until "*FILES" is found.

```
1000   RESTORE
1010   READ X$                      &
       IF X$ <> "*FILES" THEN 1010
```

— or —

```
1000   RESTORE                      &
/      READ X$                      &
/      UNTIL X$="*FILES"            &
/            READ X$                &
/      NEXT                         &
```

While the second implementation of this simple example may seem unusual (for one thing, it uses two identical READ statements), it is the most commonly applied and most maintainable representation of "DO-UNTIL." Structured programs in other languages lacking a generic "DO-UNTIL," most notably COBOL, will use such a "pump priming" READ on an input data set before entering a loop to process subsequent records. While the statements themselves are syntactically identical, reading the first record in a file is considered functionally different from reading each subsequent record; hence, we are justified in including the (apparently) same statement in two different places.

Alternatives for Formatting Programs.

It should be clear by now that a goal of structured discipline is more easily maintainable software. We need to understand, however, that a maintainable program is more a matter of clarity than mere uniformity. Structured programming will, of course, leave the selection and representation of algorithms up to the programmer, who may then choose from among available language devices to represent the required program structures. This process leaves considerable room for the development of individual style as it stresses the value of straightforward expression. At the same time, a certain measure of uniformity in program appearance will be a natural by-product.

Our examples so far have made use of indentation and spacing to emphasize the scope of effect of various control statements, and thus to improve the readability of the code. Other devices in BASIC-PLUS and BASIC-PLUS-2 can be employed to further clarify a program's intent.

User-Defined Functions.

We've already mentioned 1) that BASIC-PLUS statements are identified by numeric labels rather than alphanumeric names, and 2) "blocks" of BASIC-PLUS code must be organized into separate packets and invoked out-of-line. Using function definitions, a BASIC-PLUS programmer can partially defuse these problems — particularly in this age of "EXTEND" mode — and greatly improved code readability. Major sections of BASIC-PLUS code can be collected into appropriately named functions, and invoked by name rather than number from the main body of the program. If these functions carry no argument lists (for the most part, the limited capabilities of these argument lists offer no advantage in this context), they become mechanically interchangeable with GOSUB/RETURN, but offer significant internal documentation advantages.

The FOR Statement.

The BASIC-PLUS and PLUS-2 "FOR" statement hasn't been mentioned so far because, strictly speaking, it isn't needed to write structured programs. Just the same, FOR/NEXT loops are procedurally similr to (but not identical with) WHILE and UNTIL loops, and are an obvious choice in looping situations with concommitant indexing.

Statement Modifiers.

Special forms of the selection and iteration elements we've mentioned so far (and one we haven't mentioned: "UNLESS") can serve as modifiers of individual program

statements, and, as we have seen, these constructions can be used to circumvent compiler restrictions (particular with IF-THEN) which can hamper the implementation of the elementary structures. When the modified statement is a GOSUB, CALL, or function reference the scope of the statement modifier can be very significant; though given minor play in the manuals, statement modifiers are powerful language tools.

Logical Variables and EXTEND Mode.

All programs consist of instruction and data components, and while the chief realm of structured programming is "the procedure division," maintainable software is a function of clarity of expression within the "data division" as well. BASIC-PLUS and BASIC-PLUS-2 offer, at minimal cost (one word of job space for every three characters in BASIC-PLUS), up to 30 characters in a variable or function name. Descriptive (not necessarily lengthy) nomenclature for such items (PURCHASE.ORDER.NO$ or even PO.NO$ rather than Q3$) will go a long way toward simplifying a system's maintenance. Also, the TRUE/FALSE value of a given condition can be set or evaluated and stored in a named integer variable. Using such a variable name in place of the condition's less clear arithmetic expression — particularly in the "main" section of a program — can more clearly express the "functional reason" for a loop or "IF."

Program Comments.

Comments in program listings, flowcharts, block diagrams, and all forms of ancillary program documentation suffer the same phenomenon of program maintenance: **documentation lag.** Documentation lag occurs when, for whatever reason, a change to a program is made, but not to its accompanying documentation. This can happen not only during the maintenance of an old system, but during the development cycle of new programs as well. Under these curcumstances, ancillary documentation not only loses its descriptive value, but may even do serious damage by misleading further development or engineering efforts.

Of course, comments should not be "banned." Certain comments such as copyright notices, titles, modification histories, variable and procedure name inventories, etc. are helpful and justifiably included in locally adopted programming standards. But comments can't necessarily be relied upon. The listings of structured programs should, for the most part, offer their "logic comments" intrinsically within the code itself. Changes to the code will then necessarily change the "comments."

Some Notes on Error Handling.

BASIC-PLUS and BASIC-PLUS-2 run-time errors are handled by direct out-of-line transfers into designated error routines, which then typically "RESUME" into the main stream of processing after the proper action is taken. It has become common practice to trap all such errors to some common location (usually line 19000), where processing proceeds through a series of tests (sometimes called a "skip chain") to determine where the error actually happened, and what should be done about it.

Obviously, this is a workable device employed perhaps in every RSTS/E shop. Nonetheless, it poses some disadvantages:

1) All program segments which can cause run-time errors are bound together by the common error routine. A change to the error code can, therefore, inadvertently change the behavior of an unrelated module. This restriction is minimized when the common error routine is kept to a skeletal, minimum size.

2) The error recovery process of a program segment will not be localized to that segment. This may necessitate "page-flipping" of program listings during maintenance.

3) The "skip chain" in the common error routine is likely to contain tests on the ERL pseudo-variable (it almost certainly will if the same RSTS/E error is possible at different points in the program). Editing or resequencing line numbers in the body of the program, without reflecting these changes in the error area, will cause problems.

4) Connected through the common error routine, individual program segments can't be fully isolated from others for unit testing.

5) When converting BASIC-PLUS to BASIC-PLUS-2, program modules which are coupled to a common error routine — particularly if they are functions — will branch out of their defined boundaries to process their errors. This may defeat features in future releases of the product, such as global optimization, and such modules do not submit readily to subroutine conversion for overlaying.

The alternative to a common error area — error trapping within the segment causing the error — circumvents these disadvantages. More importantly, the general notion of functionally localized software, which is a central structured programming theme — implies local error handling.

Remember, however, that "ON ERROR GO TO 19000" cannot be labelled "bad" and forbidden, any more than "GO TO" can. Standards for such specific language mechanisms are adopted on an installation-dependent basis, in response to the unique pressures of a particular project or programming shop. Although the widely published "BASIC-PLUS Standards" document used by DEC to write RSTS/E CUSPS mandates an ON ERROR GO TO 19000, it nonetheless allows for the establishment of local error trapping where needed. In any event, where common error handling is employed, the size of the "common" area should be restricted, with the bulk of error recovery occuring local to the segment which caused it.

Performance and Efficiency.

A wealth of essentially sloppy BASIC-PLUS and BASIC-PLUS-2 code is excused on the grounds of "performance" and "efficiency." Until now, we've made no mention of the relative performance advantages of particular techniques, because **the performance characteristics of a program or system is by far more a function of its basic design than of its individual statements.** Indeed, most of us will remember that, when we've isolated performance problems in our systems, really significant improvements in efficiency statistics demanded re-design rather than re-arrangement of programs.

Just the same, for your information, bench tests on the use of GOSUB vs. function references (without arguments, since the presence and size of an argument list detracts greatly from the run-time performance of function invocation) suggest that executing functions takes about twice as

long as executing statement subroutines. By the same token, paired GOTOs in and out of code segments takes about half the time as GOSUB/RETURN. In BASIC-PLUS-2, corresponding language devices require about half the run-time of their BASIC-PLUS counterparts, and the ratios among their performance characteristics remain essentially constant. Interestingly, the BASIC-PLUS-2 "CALL" is the slowest of all these constructs. Remember, though, that in comparing these implementation-dependent program control devices, we are dealing in milliseconds (an 11/70 takes about 2.2 seconds to invoke a function **10000** times; 10000 iterations through the corresponding GOSUB construction took about 1 second).

As we mentioned earlier, the size of variable names has no effect on the size of a BASIC-PLUS-2 program, while, for each name, BASIC-PLUS requires 1 word of job space for evary three characters.

People are always debating the size/performance issues of structured programming, but so far the results of comparisons between structured and unstructured code has offered nothing conclusive. In BASIC-PLUS and PLUS-2, structured programs will need fewer line numbers and line number references than unstructured programs; perhaps this economy can be used offhandedly to "pay" for larger variable names and function definitions.

We can probably put this issue to rest by stating that performance and efficiency are **not** goals of structured programming, but the preliminary planning and attention that structured programming demands tend to lead to software which is designed rather than simply "tooled" to be efficient.

Program Design.

An extensive discussion of the design techniques which normally precede the development of structured programs, often referred to as "top-down" design, is beyond the scope of this article. Traditionally, however, structured systems are designed from the "top" of an application (the part the user sees) to the "bottom" (the part the programmer deals with) by successively refining nested functional modules until the lowest modules can be implemented in the target programming language. This process of "stepwise refinement" offers relatively quick availability of the "user interface" portion of a system, and on-going consistency checks which assure the ultimate correct interplay of all software components.

Top-down system design is similar to outlining the contents and component topics of a book before it is written. The major topics are selected and ordered first, then each major topic is "refined" in terms of its sub-topics, until the lowest level topics are specified enough to allow orderly research and composition. In the final analysis, sticking to the original outline guarantees a cohesive final product.

In a multi-person project, the top-down design strategy allows (to an extent) the parallel design and implementation of individual components, and offers a framework for the partitioning of "people tasks" which allows engineers to work independently and yet converge on an integral product.

While the elementary structures we've itemized and discussed so far are sufficient for building structured programs, any such resultant program is not guaranteed to be structured simply because it consists of organized pieces. The preliminary act of functional analysis, however informal, is essential to the development of structured software.

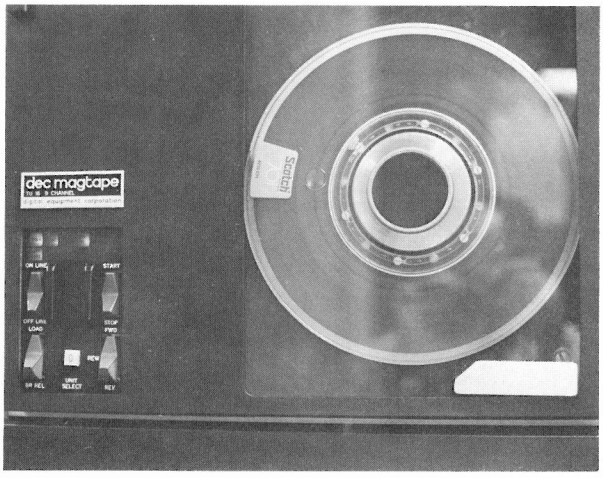
Examples.

These examples programs are based on simple specifications, selected to permit easy reading and to demonstrate most of the constructions presented in this article. In a few cases, particularly within example 2, limited design approaches were employed in order to "force" the occurrence of specimen constructs.

STRUCTURED PROGRAMMING SAMPLE 1

This sample program will read a line of characters from an input terminal, reverse it, and display the reversed string. Execution continues through successive input records until and end of file (CTRL/Z) is sensed.

```
10      EXTEND
20      YES%,TRUE%=-1% &
\       NO%,FALSE%=0%
1000 &
!MAIN PROGRAM TO READ, REVERSE, AND DISPLAY STRINGS OF CHARACTERS &
        X%=FNREAD.STRING% &
\       UNTIL NO.MORE.STRINGS% &
\               REVERSE$='' &
\               REVERSE$=REVERSE$+MID(ASCII.STRING$,I%,1%) &
                        FOR I%=LEN(ASCII.STRING$) TO 1% STEP -1% &
\               PRINT 'REVERSED: ';REVERSE$ &
\               X%=FNREAD.STRING% &
\       NEXT &
!END OF READ/REVERSE PROGRAM. &
 &
 &

2000 &
DEF FNREAD.STRING% &
\       ON ERROR GO TO 2090 &
\       PRINT "SERIES OF CHARACTERS, PLEASE"; &
\       INPUT LINE ASCII.STRING$ &
\       ASCII.STRING$=CVT$$(ASCII.STRING$,4%) &
\       NO.MORE.STRINGS%=NO% &
\       GO TO 2099
2090    IF ERR=11% THEN &
                NO.MORE.STRINGS%=YES% &
\               RESUME 2099
2099    ON ERROR GO TO 0 &
\FNEND &
!END OF ASCII STRING INPUT FUNCTION &
 &
 &

32767   END &
 &
 &
```



STRUCTURED PROGRAMMING SAMPLE 2

This sample program reads successive lines of characters from the keyboard, keeping count of the occurrence of vowels within them. A sub-total report is printed after each line, and a grand total after end of file (CTRL/Z).

```
10      EXTEND
20      YES%,TRUE%=-1% &
\       NO%,FALSE%=0%
1000 &
!PROGRAM TO READ ASCII STRINGS FROM A TERMINAL, AND COUNT THE OCCURENCE  &
!OF EACH VOWEL. SUB-TOTALS BY LINE AND GRAND TOTALS ARE PRINTED. &
        A.TOT%, &
        E.TOT%, &
        I.TOT%, &
        O.TOT%, &
        U.TOT%=0% &
\       X%=FNREAD.STRING% &
\       UNTIL NO.MORE.STRINGS% &
\               A.SUBTOT%, &
                E.SUBTOT%, &
                I.SUBTOT%, &
                O.SUBTOT%, &
                U.SUBTOT%=0% &
\               X%=FNEXTRACT.FIRST.CHARACTER% &
\               UNTIL NO.MORE.CHARACTERS% &
\                       INDEX%=INSTR(1%,"AEIOU",CHAR$) &
\                       ON INDEX% GOSUB &
                                1100, ! "A" FOUND  &
                                1200, ! "E" FOUND  &
                                1300, ! "I" FOUND  &
                                1400, ! "O" FOUND  &
                                1500  ! "U" FOUND  &
                                        UNLESS INDEX%=0% &
\                       X%=FNEXTRACT.FIRST.CHARACTER% &
\               NEXT &
\               X%=FNDISPLAY.DETAIL.TOTALS% &
\               X%=FNREAD.STRING% &
\       NEXT &
\       X%=FNDISPLAY.GRAND.TOTALS% &
\GO TO 32767 &
!END OF VOWEL COUNTING PROGRAM. &
! &
! NOTE: WE BRANCH TO 32767 IN THIS SAMPLE TO GET AROUND THE &
! GOSUB-TYPE SUBROUTINES AT LINES 1100-1500. IN SAMPLE 1, WHICH &
! USED ONLY FUNCTION-TYPE SUBROUTINES, EXECUTION NATURALLY &
! SKIPPED AROUND THE SUBROUTINE CODE. &
 &
 &

1100    ! INCREMENT "A" COUNTERS &
        A.TOT%   =A.TOT%   +1% &
\       A.SUBTOT%=A.SUBTOT%+1% &
\       RETURN &
 &
 &

1200    ! INCREMENT "E" COUNTERS &
        E.TOT%   =E.TOT%   +1% &
\       E.SUBTOT%=E.SUBTOT%+1% &
\       RETURN &
 &
 &
```

```
1300    ! INCREMENT "I" COUNTERS &
        I.TOT%    =I.TOT%    +1% &
\       I.SUBTOT%=I.SUBTOT%+1% &
\       RETURN &
 &
 &

1400    ! INCREMENT "O" COUNTERS &
        O.TOT%    =O.TOT%    +1% &
\       O.SUBTOT%=O.SUBTOT%+1% &
\       RETURN &
 &
 &

1500    ! INCREMENT "U" COUNTERS &
        U.TOT%    =U.TOT%    +1% &
\       U.SUBTOT%=U.SUBTOT%+1% &
\       RETURN &
 &
 &
2000 &
DEF FNREAD.STRING% &
\       ON ERROR GO TO 2090 &
\       PRINT "SERIES OF CHARACTERS, PLEASE"; &
\       INPUT LINE ASCII.STRING$ &
\       ASCII.STRING$=CVT$$(ASCII.STRING$,4%) &
\       NO.MORE.STRINGS%=NO% &
\       GO TO 2099
2090    IF ERR=11% THEN &
                NO.MORE.STRINGS%=YES% &
\               RESUME 2099
2099    ON ERROR GO TO 0 &
\FNEND &
!END OF ASCII STRING INPUT FUNCTION &
 &
 &
2100 &
DEF FNEXTRACT.FIRST.CHARACTER% &
\       NO.MORE.CHARACTERS%=(LEN(ASCII.STRING$)=0%) &
\       IF NOT (NO.MORE.CHARACTERS%) THEN &
                CHAR$=LEFT(ASCII.STRING$,1%) &
\               ASCII.STRING$=RIGHT(ASCII.STRING$,2%) &
!       (ELSE) &
!       ENDIF.
2190 &
FNEND &
 &
 &

2200 &
DEF FNDISPLAY.DETAIL.TOTALS% &
\       PRINT "FOR THIS STATEMENT:" &
\       PRINT "  A COUNT =";A.SUBTOT% &
\       PRINT "  E COUNT =";E.SUBTOT% &
\       PRINT "  I COUNT =";I.SUBTOT% &
\       PRINT "  O COUNT =";O.SUBTOT% &
\       PRINT "  U COUNT =";U.SUBTOT% &
\FNEND &
 &
 &
```

```
2300 &
DEF FNDISPLAY.GRAND.TOTALS% &
\       PRINT "GRAND TOTALS:" &
\       PRINT "    A TOTAL =";A.TOT% &
\       PRINT "    E TOTAL =";E.TOT% &
\       PRINT "    I TOTAL =";I.TOT% &
\       PRINT "    O TOTAL =";O.TOT% &
\       PRINT "    U TOTAL =";U.TOT% &
\FNEND &                                                          32767  END &
 &                                                                 &
 &                                                                 &
```

STRUCTURED PROGRAMMING SAMPLE 3

This example is a RSTS/E system program designed to provide SYSTAT-like reporting features along with UTILTY-like job-killing capabilities. A user of this "KILJOB" program can display and kill user jobs running under specific accounts (project and programmer numbers can be provided as ranges, or as specific values). This example includes generalized monitor look-up routines which are self-documenting and may be removed for use in some other context.

(Some of the functions designed in this program accept and return arguments and results in the traditional function-definition sense. Remember that, when these specific features are used, function definitions lose their interchangeability with GOSUB/RETURN. As we've said, under normal circumstances the performance advantages of GOSUB are negligible. In a compute-bound routine, however, where out-of-line coding segments are invoked within high-traffic loops, significant performance advantages [the term 'significant' is very relative] can be gained by using GOSUB/RETURN in lieu of function definitions. Of course, executing the code in-line, if possible, offers still greater run-time economy.)

```
10      !EXTEND!
20      PNAME$="KILJOB" &
\       VERSN$="V01" &
\       EDITION$ ="A"
100     PRINT IF CCPOS(0) &
\       EXEC% = FNMONITOR.TABLES% &
\       PRINT PNAME$+"  "+VERSN$+"/"+EDITION$+"  "; &
\       INSTALLATION.NAME$=CVT$$(RIGHT(SYS(CHR$(6%)+CHR$(9%)),3%),4%) &
\       PRINT "   SYSTEM ID = "; INSTALLATION.NAME$ &
\       PRINT "TODAY = ";DATE$(0%);"  NOW = ";TIME$(0%); &
\       JOB.NO% = FNJOB.NO% &
\       P.PN%   = FNP.PN%(JOB.NO%) &
\       PROJ%   = SWAP%(P.PN%) AND 255% &
\       PROG%   = P.PN% AND 255% &
\       KB%     = FNCONSOLE.KB%(JOB.NO%) &
\       PRINT "JOB";JOB.NO%;"  KB";NUM1$(KB%);":  "; &
              "["+NUM1$(PROJ%)+","+NUM1$(PROG%)+"]" &
\       PRINT &
!       DERIVE AND PRINT OUR FULL IDENTIFICATION. &
 &
 &

1000    EXEC%=FNGET.OPTION% &
\       UNTIL OPTION$='E' &
\               IF OPTION$='H' THEN &
                        EXEC%=FNHELP.OPTION% &
                ELSE &
                        IF OPTION$='L' THEN &
                                EXEC%=FNLIST.OPTION%  &
                        ELSE &
                                IF OPTION$='K' THEN &
                                        EXEC%=FNKILL.OPTION% &
                                ELSE &
                                        PRINT 'SORRY, BUT ';ENTERED.OPTION$; &
                                           ' IS INVALID.' &
\                                       PRINT 'TYPE "HELP" FOR HELP.' &
\                                       PRINT &
```

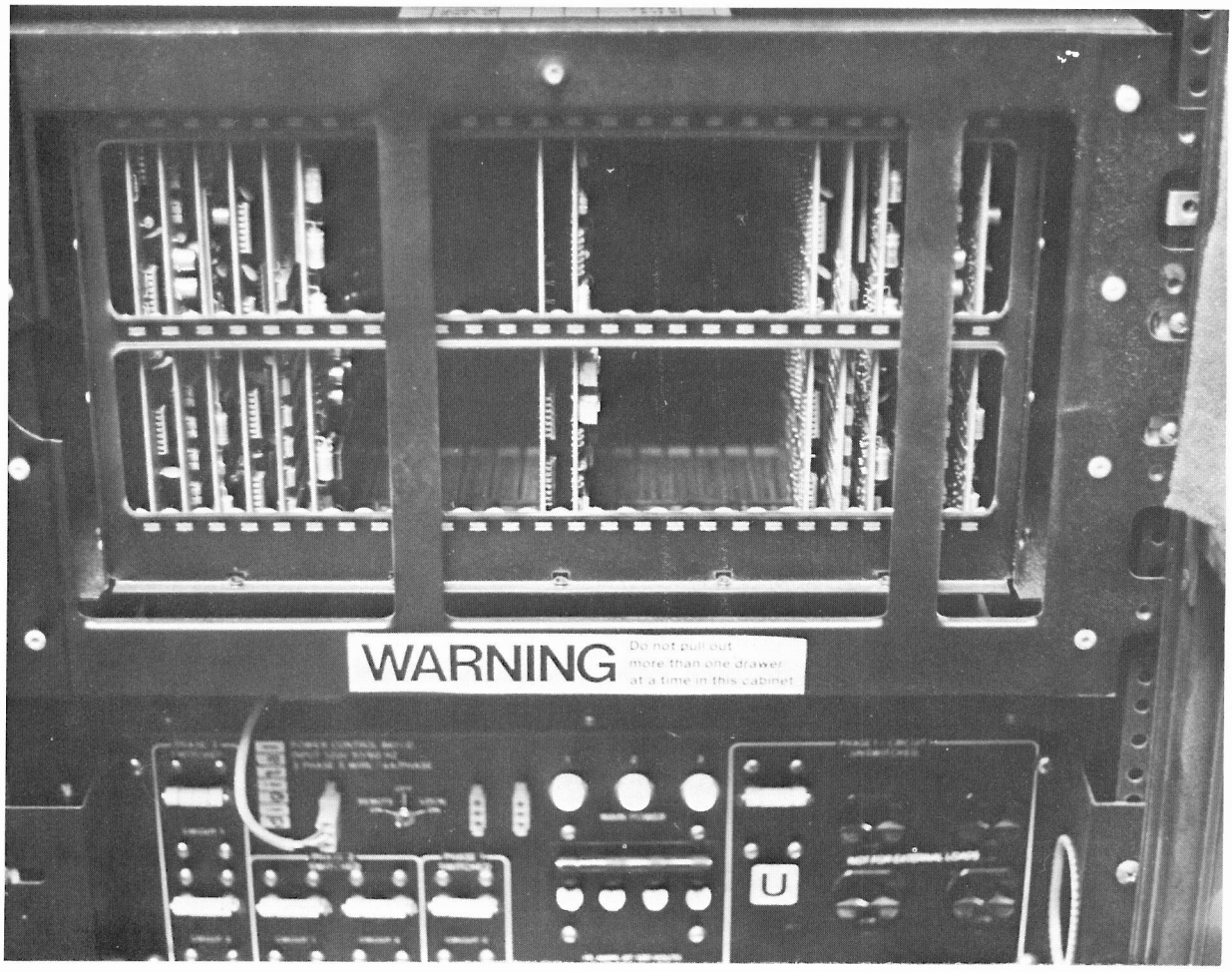
```
1010            EXEC%=FNGET.OPTION% &
\       NEXT &
\       PRINT &
\       PRINT 'END OF JOB' &
\       PRINT &
\       GO TO 32767 &
 &
 &

2000    &
DEF*    FNHELP.OPTION% &
\       PRINT &
\       PRINT 'OPTIONS ARE:' &
\       PRINT TAB(4%);'L[IST]';TAB(20%);'LIST THE JOB TABLE' &
\       PRINT TAB(4%);'K[ILL]';TAB(20%);'KILL JOB(S)' &
\       PRINT TAB(4%);'E[XIT]';TAB(20%);'EXIT FROM ';PNAME$ &
\       PRINT TAB(4%);'H[ELP]';TAB(20%);'PRINT THIS OPTIONS LIST' &
\       PRINT &
\       FNEND &
!END    FNHELP.OPTION% &
 &
 &

2010 &
DEF*    FNLIST.OPTION% &
\       EXEC%=FNGET.P.PN.SPECS% &
\       PRINT &
\       PRINT INSTALLATION.NAME$;'  JOBMAX:';MAX.JOB%;'  KBMAX:';MAX.KB% &
\       PRINT &
\       PRINT 'JOB(S) SELECTED:' &
\       PRINT '  PROJ:';LOW.PROJ%;'-';HI.PROJ% &
\       PRINT '  PROG:';LOW.PROG%;'-';HI.PROG% &
\       PRINT &
\       JOB.COUNT%=0% &
\       FOR JOB%=1% TO MAX.JOB% &
\               IF FNLOGGED.IN%(JOB%)=-1% THEN &
                        IF FNQUALIFY%(FNP.PN%(JOB%))=-1% THEN &
                                PRINT FNJOB.LINE$(JOB%) &
\                               JOB.COUNT%=JOB.COUNT%+1%
\
2015    NEXT JOB% &
\       PRINT &
\       PRINT 'TOTAL JOBS FOUND:'; JOB.COUNT% &
\       PRINT &
\       FNEND &
!END    FNLIST.OPTION% &
 &
 &

2020 &
DEF*    FNKILL.OPTION% &
\       OPEN 'KB:' AS FILE 1% &
\       EXEC%=FNGET.P.PN.SPECS% &
\       PRINT 'SELECTIVE? (Y OR N) <Y>'; &
\       INPUT LINE  1%, RESP$ &
\       SEL.FLAG%=-1% &
\       SEL.FLAG%=0% &
\               IF ASCII(CVT$$(RESP$,-1%))=ASCII('N') &
\       FOR JOB%=1% TO MAX.JOB% &
\               IF FNLOGGED.IN%(JOB%)=-1% THEN &
                        IF FNQUALIFY%(FNP.PN%(JOB%))=-1% THEN &
                                EXEC%=FNKILL.JOB%(JOB%)
```

```
2025    NEXT JOB% &
\       FNEND &
!END    FNKILL.OPTION% &
 &
 &
3000 &
DEF*    FNJOB.LINE$(JOB%) &
\       X.3000$ = '' &
\       X.3000$ = 'JOB  '+NUM1$(JOB%) &
\       IF FNLOGGED.IN%(JOB%)=0% THEN &
                X.3000$ = X.3000$+CHR$(9%)+' ****   NOT LOGGED IN   ****' &
        ELSE &
                X.3000$ = X.3000$+CHR$(9%)+"UNDER ["+ &
                        NUM1$(SWAP%(FNP.PN%(JOB%)) AND 255%)+','+ &
                        NUM1$(      FNP.PN%(JOB%)  AND 255%)+']'  &
\               X.3000$ = X.3000$+CHR$(9%)+"RUNNING "+FNJOB.NAME$(JOB%) &
\               IF FNATTACHED%(JOB%) = -1% THEN &
                        X.3000$ = X.3000$+CHR$(9%)+"AT
 KB"+NUM1$(FNCONSOLE.KB%(JOB%))         &
                                +":" &
                ELSE &
                        X.3000$ = X.3000$+CHR$(9%)+"DETACHED" &
3010    FNJOB.LINE$=X.3000$ &
\       FNEND &
!END    FNJOB.LINE$ &
 &
 &
4000 &
DEF*    FNGET.OPTION% &
\       OPEN 'KB:' AS FILE 1% &
\       PRINT  1%, 'OPTION:'; &
\       INPUT LINE  1%, ENTERED.OPTION$ &
\       ENTERED.OPTION$=CVT$$(ENTERED.OPTION$,4%) &
\       OPTION$=LEFT(CVT$$(ENTERED.OPTION$,-1%),1%) &
\       OPTION$='H' &
                IF OPTION$='' &
\       FNEND &
!END    FNGET.OPTION% &
 &
 &
4100    &
DEF*    FNGET.P.PN.SPECS% &
\       OPEN 'KB:' AS FILE 1% &
\       PRINT  1%, "  PROJ: <1-255> "; &
\       INPUT LINE  1%, PROJ.RANGE$ &
\       EXEC%=FNPARSE%(PROJ.RANGE$) &
\       LOW.PROJ%=LOW.VAL% &
\       HI.PROJ%=HI.VAL% &
\       PRINT  1%,"  PROG: <1-255> "; &
\       INPUT LINE  1%, PROG.RANGE$ &
\       EXEC%=FNPARSE%(PROG.RANGE$) &
\       LOW.PROG%=LOW.VAL% &
\       HI.PROG%=HI.VAL% &
\       PRINT &
\       PRINT "PROJ:";LOW.PROJ%;'-';HI.PROJ% &
\       PRINT "PROG:";LOW.PROG%;'-';HI.PROG% &
\       PRINT &
\       FNEND &
!END    FNGET.P.PN.SPECS% &
 &
 &
```

```
5000 &
DEF*    FNPARSE%(RANGE$) &
\       ON ERROR GO TO 5020 &
\       RANGE$=CVT$$(RANGE$,-1%) &
\       RANGE$='1-255' &
                IF LEN(RANGE$)=0% &
\       D.5000%=INSTR(1%,RANGE$,'-') &
\       IF D.5000%=0% THEN &
                D.5000%=LEN(RANGE$)+1% &
\               RANGE$=RANGE$+'-'+RANGE$
5005    RANGE$='1'+RANGE$ &
                IF D.5000%=1% &
\       RANGE$=RANGE$+'255' &
                IF D.5000%=LEN(RANGE$) &
\       LOW.VAL%=VAL(LEFT(RANGE$,D.5000%-1%)) &
\       HI.VAL% =VAL(RIGHT(RANGE$,D.5000%+1%))
5007    IF LOW.VAL%>HI.VAL% THEN &
                LOW.VAL%=VAL('X') &
!       IF LOWER IS GREATER THAN HIGHER, THEN FORCE AN ERROR. &

5010    FNEND
5020    LOW.VAL%=1% &
\       HI.VAL%=255% &
\       PRINT  1%, "ILLEGAL RANGE: 1-255 ASSUMED." &
\       RESUME 5010 &
!END    FNPARSE% &
 &
 &

5100 &
DEF*    FNKILL.JOB%(JOB%) &
\       IF SEL.FLAG%=0% THEN &
                EXEC%=FNCLEAR.JOB%(JOB%) &
        ELSE &
                PRINT  1%, 'KILL ';FNJOB.LINE$(JOB%);'? <N>'; &
\               INPUT LINE  1%, RESP$ &
\               IF ASCII(CVT$$(RESP$,-1%))=ASCII('Y') THEN &
                        EXEC%=FNCLEAR.JOB%(JOB%)
5110    FNEND &
!END    FNKILL.JOB% &
 &
 &

5200 &
DEF*    FNQUALIFY%(P.PN%) &
\       IF (SWAP%(P.PN%) AND 255%)>=LOW.PROJ% &
                AND (SWAP%(P.PN%) AND 255%)<=HI.PROJ% &
                AND (P.PN% AND 255%)>=LOW.PROG% &
                AND (P.PN% AND 255%)<=HI.PROG% THEN &
                        FNQUALIFY%=-1% &
            ELSE &
                        FNQUALIFY%=0% &
 &
 &

5210    FNEND &
!END    FNQUALIFY% &
 &
 &
```

```
5300 &
DEF*    FNCLEAR.JOB%(JOB%) &
\       IF JOB%=JOB.NO% THEN &
                PRINT CHR$(7%)+ "  HEY, I'M JOB";JOB%;"- I CAN'T DO THAT!" &
        ELSE &
                FNCLEAR.JOB%=ASCII(SYS(CHR$(6%)+CHR$(8%)+CHR$(JOB%) &
                   +STRING$(24%,0%)+CHR$(255%))) &
\               PRINT '  -- JOB';JOB%;'KILLED.' &

5310    FNEND &
!END    FNCLEAR.JOB% &
 &
 &

28000 &
DEF*    FNMONITOR.TABLES% &
\       DIM T.3%(30%) &
\       CHANGE SYS(CHR$(6%)+CHR$(-3%)) TO T.3% &
\       MAX.KB% = T.3%(3%) &
\       MAX.JOB%= T.3%(4%) &
\       DEVCNT% = T.3%( 5%)+SWAP%(T.3%( 6%)) &
\       DEVPTR% = T.3%( 7%)+SWAP%(T.3%( 8%)) &
\       MEMLST% = T.3%( 9%)+SWAP%(T.3%(10%)) &
\       JOBTBL% = T.3%(11%)+SWAP%(T.3%(12%)) &
\       JBSTAT% = T.3%(13%)+SWAP%(T.3%(14%)) &
\       JBWAIT% = T.3%(15%)+SWAP%(T.3%(16%)) &
\       UNTCLU% = T.3%(17%)+SWAP%(T.3%(18%)) &
\       UNTCNT% = T.3%(19%)+SWAP%(T.3%(20%)) &
\       SATCTL% = T.3%(21%)+SWAP%(T.3%(22%)) &
\       JSBTBL% = T.3%(23%)+SWAP%(T.3%(24%)) &
\       SATCTM% = T.3%(25%)+SWAP%(T.3%(26%)) &
\       CHANGE SYS(CHR$(6%)+CHR$(-12%)) TO T.3% &
\       FREES%  = T.3%( 3%)+SWAP%(T.3%( 4%)) &
\       DEVNAM% = T.3%( 5%)+SWAP%(T.3%( 6%)) &
\       CSRTBL% = T.3%( 7%)+SWAP%(T.3%( 8%)) &
\       DEVOKB% = T.3%( 9%)+SWAP%(T.3%(10%)) &
\       TTYHCT% = T.3%(11%)+SWAP%(T.3%(12%)) &
\       JOBCNT% = T.3%(13%)+SWAP%(T.3%(14%)) &
\       RTSLST% = T.3%(15%)+SWAP%(T.3%(16%))    &
\       ERLCTL% = T.3%(17%)+SWAP%(T.3%(18%))    &
\       SNDLST% = T.3%(19%)+SWAP%(T.3%(20%)) &
\       LOGNAM% = T.3%(21%)+SWAP%(T.3%(22%)) &
\       DEVSYN% = T.3%(23%)+SWAP%(T.3%(24%)) &
\       MEMSIZ% = T.3%(25%)+SWAP%(T.3%(26%)) &
\       CCLLST% = T.3%(27%)+SWAP%(T.3%(28%)) &
\       FNEND &
!END    FNMONITOR.TABLES% &
                        &
\DEF*   FNJOB.NO%       = (PEEK(518%) AND 255%)/2% &
 &
 &
\DEF*   FNJDB%(JOB.NO%) = PEEK(JOBTBL%+2%*JOB.NO%) &
 &
 &
\DEF*   FNJDB2%(JOB.NO%)= PEEK(FNJDB%(JOB.NO%)+8%) &
 &
 &
\DEF*   FNP.PN%(JOB.NO%)= PEEK(FNJDB2%(JOB.NO%)+24%) &
 &
 &
```

```
\DEF*   FNJOB.NAME$(JOB.NO%) = &
                RAD$(PEEK(FNJDB2%(JOB.NO%)+12%)) &
            +   RAD$(PEEK(FNJDB2%(JOB.NO%)+14%)) &
 &
 &
\DEF*   FNLOGGED.IN%(JOB.NO%) = &
                [PEEK(JOBTBL%+2%*JOB.NO%) <> 0%] &
 &
 &
\DEF*   FNIOB%(JOB.NO%) = PEEK(FNJDB%(JOB.NO%)) &
 &
 &
\DEF*   FNCONSOLE.KB%(JOB.NO%) = &
                (SWAP%(PEEK(PEEK(FNIOB%(JOB.NO%))+2%)) AND 255%) &
 &
 &
\DEF*   FNATTACHED%(JOB.NO%) = &
                [((PEEK(PEEK(PEEK(FNJDB%(JOB.NO%)))+2%) AND 255%) &
                        = JOB.NO%*2% &
                   AND &
                  (PEEK(PEEK(PEEK(FNJDB%(JOB.NO%)))+6%) AND 8192%)=8192%) &
                   AND &
                  FNLOGGED.IN%(JOB.NO%)<>0% ] &
! &
! &
!       *** END OF MONITOR INFORMATION RETRIEVAL FUNCTIONS &
!32767          END &
```



RSTS/E — THE IN-HOUSE TIMESHARING ALTERNATIVE

By Kenneth Ross, President, Ross Systems, Inc.

SUMMARY

The last 5 years have seen the introduction of high-speed, low cost mini-computers with interactive timesharing capabilities. With the recent advent of user oriented, problem solving languages such as financial modeling and interactive data base management that run on these computers and the availability of public, nationwide networks, it has now become feasible to set up an in-house timesharing facility. A PDP-11 with the RSTS/E operating system presents one of the best alternatives for establishing such a facility.

THE MINI-COMPUTER MARKETPLACE

Over the years, the market for business uses of mini-computers can be visualized as divided into three broad segments. In summary, these are:

1. General purpose business machines that represent the main data processing applications of a company such as accounting systems, inventory control, etc.
2. Machines dedicated to one or a few specialized applications in a company which has additional computers for other data-processing tasks.
3. Machines installed to provide general, interactive data processing to end users.

The next few years will see a dramatic increase in the purchase of computers for the last of these.

TIMESHARING DEFINITION

Timesharing can be thought of as the use of the computer to provide general, interactive computing services directly to end users. It differs from traditional computing in that the users of the service develop many of the applications themselves rather than relying on computer professionals to do it for them. Additionally, the applications that are done on timesharing are likely to be supplementary to the company's core systems requirements and oriented more towards decision-making rather than data manipulation. Financial models, budgets, financial consolidations, data retrieval systems, graphics and other data analysis systems are examples of timesharing systems.

The users of timesharing benefit from these services by being able to develop, operate and change systems at their own discretion.

The company's data processing organization benefits by being freed from the inherent problems of dealing with numerous, smaller systems so that they can concentrate upon the major operational systems that are critical to the company's business.

The market for general purpose timesharing up to now has been totally dominated by the major, nationwide companies with large networks and offices throughout the U.S. (and overseas), General Electric, Service Bureau Corp., and TYMSHARE are just some of these major companies.

These timesharing firms have traditionally been effective in providing services to firms that already have large EDP organizations because they have offered 4 major components to their services that end users need:

1. Problem oriented languages that allow users to set-up, operate and change aplications directly without help from EDP people.
2. A great deal of support via local 'tech reps'.
3. National (international) networks.
4. Interactive computers that are easier to use than their in-house computers and could be used for simple BASIC programming.

A company that intends to provide these services internally must provide these four elements to its users if it hopes to compete with outside firms.

PROBLEM ORIENTED LANGUAGES

It is important that general timesharing users not feel the frustration that they sometimes associate with in-house EDP projects. In larger companies, even the smallest projects can take months or even years to complete. In-house EDP staffs are not oriented toward the smaller, decision support systems that are normally done on timesharing services. Traditional procedural computer languages such as FORTRAN and COBOL are programmer oriented. Timesharing firms have developed special problem oriented languages which users can use directly to develop, change and operate certain types of systems. Over the years, the two biggest applications in this marketplace have been:

1. Financial Modeling and Reporting
2. Interactive Data-Base Management

A company that wants a successful in-house timesharing facility must typically offer both of these products to its users.

FINANCIAL MODELING AND REPORTING

The first widely used problem oriented languages on general timesharing systems were used for financial modeling and reporting. These languages are used to develop and operate a wide range of "matrix" oriented systems such as models, budgets, consolidations and cash flow forecasts. They are used by financial personnel — analysts, controllers, and accounts, to develop forecasting systems and reporting systems. Since there is a heavy amount of user involvement in the manipulation of data and reports, financial modeling languages tend to be used as a supplement to general ledger systems. In fact, a major source of input data is a summary taken from general ledgers.

Are you sure this is a DEC computer?

In a typical application, the user defines the rows of the report, its columns, the report titles and its calculation logic. Logic, for example, might involve summing a series of rows to get a subtotal or multiplying a row by a constant to get another row. While simple in concept, modeling languages have grown in capability to handle such features as large-scale financial data-bases, simultaneous equations (used in modeling), backwards iteration and complex logical calculations. In our experience with MAPS, our proprietary financial modeling language, we have used it to construct financial consolidation and reporting systems for fortune 500 companies as well as simple cash flow forecasting models for small electronics manufacturing firms. The people that use MAPS include accounting clerks, controllers, programmer-analysts and vice-presidents.

Financial modeling languages, in order to be effective in a timesharing environment, should be easy-to-use yet should contain enough features so that "real world" problems can be solved by them. Some of these features should be — flexible logic and calculations, powerful report formatting capabilities, extensive data storage and retreival methods and large capacity.

INTERACTIVE DATA-BASE MANAGEMENT

Interactive data-base management systems are used to manage records of data — data maintenance, sorting, reporting and "ad hoc" inquires. While financial modeling languages deal with matrices, data-base management systems deal with records or transactions. The timesharing, interactive requirements for data-base systems differ from traditional data base systems in 2 important aspects.

1. A person with no programming background should be able to use an interactive data-base management system from start to finish to set-up and operate an application — all without writing a computer program. This includes defining the data-base, adding, changing and deleting information from the data-base, changing the data-base definition, performing ad hoc inquiries and developing complex reports.
2. The need for sophisticated data structures (rings, networks, etc.) is less, yet the need for extensive, easy to use reporting features is greater.

Our firm originally attempted to use RMS-11 and DATATRIEVE in this environment. We were unsuccessful because of the limited reporting facilities that were available in DATATRIEVE and the difficulty of programming in BP II/RMS-11. We then developed, from the ground up, a completely interactive data-base management system that met all of our objectives. ORBIT has now been in use for a number of months as a general purpose data-base management system.

In order to be effective in a timesharing environment, an interactive data-base management system should have features such as easy, question and answer format for creating data files, automatic editing of data, good inquery facilities and a way to easily generate more complex reports. Today, data base management systems are the single largest used products on timesharing systems.

USER SUPPORT

The second requirement for effective in-house timesharing is good user support. This is the one that the major timesharing firms have been most effective at, and the one that is the most underestimated in terms of effort, for companies that are bringing timesharing in-house. Many users need a great deal of "hand holding". There is a wide ranging of capabilities of users in any firm and in order to keep all users satisfied, some of them will require much more support than others. The potential installer of an in-house timesharing service must be prepared to hire a staff of people that can handle the problems, requests and even new system development that is part of timesharing.

NETWORKS

Traditionally, the large timesharing firms were the only ones to offer easy network access, both domestic and overseas. Over the last few years, this is no longer the case and the availability of at least 2 public, packet switched networks makes it easy to hook a RSTS system to a network for only a few hundred dollars.

EASY TO USE COMPUTERS

The final item that the major timesharing firms offered their cutomers was a computer that was easy to use relative to the firms in-house computer. Easy to use refers both to the operating system features (directories, editor, login/logout etc.) as well as the programming facilities such as BASIC and FORTRAN. The advent of RSTS/E and machines with the capability of an 11/70 offer facilities even easier to use than timesharing firms offer. Combined with the BASIC PLUS interpreter, RSTS offers the best interactive timesharing environment available today.

CONCLUSION

Today the RSTS systems offer the best facility for in-house timesharing. Of the 4 basic requirements, problem oriented languages for financial modeling and interactive data base management are now available from vendors like Ross Systems, networks are readily available from either Tymnet or Telenet and DEC supplies the easy-to-use computer via the RSTS operating system. The computer must supply only the user support.

BIOGRAPHY

Ross Systems is a management consulting and computer timesharing firm located in Palo Alto, California. We operate 3 PDP-11/70 RSTS systems and offer for sale proprietary software including MAPS, a financial modeling and reporting language and ORBIT, an interactive data base management system.

? Why TECO ?

By Carl B. Marbach

The main reason to use TECO and the major premise of this article is —

! BECAUSE IT IS THE EASIEST EDITOR TO USE !

The fact that it is the fastest, most versatile, and most powerful editor is secondary to the main premise. It is always interesting to note that editors are almost always used by experienced programmers and users, and we are afraid it's too complicated for these talented people! For any of you who have stayed away from TECO because it was too complicated, hard, or secretive (it is not a DEC supported product, just everyone at DEC uses it), read on and I'll show you the easiest editor around.

The first thing one wants to do to edit is invoke the editor and tell it what you want to edit. Thus,

```
TECO FOO.BAS
```

starts TECO, identifies that you want to edit FOO.BAS and as a bonus, create FOO.BAK in case you destroy or modify something you didn't want to. Clearly, there is no shorter, clearer or easier way to do this.

TECO now responds with an * indicating its readiness to accept commands. A subset of TECO commands necessary to edit files are listed here.

V	View the line you are currently on.
L	Move to the next line.
D	Delete the next (following) character.
K	Delete from where you are to to the next CR LF, inclusive.
Ixxx<ESC>	Insert text XXX.
Sxxx<ESC>	Search for XXX. Searches and Inserts are delimited by ESC (altmode).
<ESC> <ESC>	Go and do it.
EX<ESC><ESC>	All finished, exit to monitor.

Thus, to delete the 10th line of a Basic program called FOO:

```
TECO FOO.BAS        !Call in the editor
*                   !TECO prompt
9L                  !Move 9 lines forward
K                   !Kill it
< ESC><ESC>         !Execute the command
*                   !TECO prompt for next
                     command
```

TECO commands may be given one at a time. It is usually more convenient to type, in a single command string, several commands that form a logical group. <ESC> are only needed between things that require a delimiter (i.e., searches). Typically, this looks like

```
*9LK$$              !$ is an <ESC> echo
```

To delete the 10th character in the 10th line:

```
*9L9CD$$
```

To delete CARL from the text:

```
*SCARL$             !Delimit the search with one
                     <ESC>
*-4D                !Delete the four characters
                     (- means before)
*V$$                !View it and do it
```

To insert RSTS:

```
*IRSTS PRO<CR>      !Note that carriage return places
$$                   you in the next line and is re-
                     quired to insert a line.
```

Note that <RETURN> is not used very much in TECO. <ESC delimits searches and tells it when to go. The reason is that you may someday want to operate on <RETURN> (delete it to move two lines into one) and using <ESC> divorces TECO delimiters from the standard monitor delimiter, <RETURN>. Notice that now you can invoke the editor, search, add, and delete either lines or characters, and exit saving the file and creating a backup.

You have now notices that commands can take numeric arguments (-4C) and they can operate forward and backward. An important TECO concept is the pointer (where you are in the file). Movements C and -C are relative to the current pointer position while L (or -L) puts you at the beginning of the next (or previous) line. After a little practice you will find keeping track of the pointer second nature, and movements becoming automatic. The single letter commands are somewhat cryptic but they make movements in the text and commands easy to type in and perform.

Why edit at all? I know RSTS sites where they don't use any editor except for patching cusps, and then it is a cookbook approach. I have implemented TECO as an editor here because, simply, it increases programmer productivity. We can change all the "GET #File.5%" in one swell foop, or realign an output format quickly and easily. We almost never retype a whole line at the monitor level unless we are debugging interactively. In short, editors allow a single change on a long line to be made quickly and easily and they allow multiple changes to be made (in loops) with one command.

This is not meant to be a TECO text, but simply an argument that you can use TECO easily. A programmer with interactive experience can read the beginners TECO section in the manual, and begin editing in less than half a day; and be proficient in less than one week of editing. TECO offers not only editing, but a powerful string manipulation language that has great power and features.

? How TECO ?

Martin Pring, D. Phil., Director, Medical School Computer Facility
University of Pennsylvania, 37th and Hamilton Walk, Philadelphia, PA 19104

The previous article has convincingly argued the case for 'Why TECO?'. One cogent reason is that, despite the richness of the command set and the consequent power of TECO as an editing language, the beginner can perform any straightforward editing task very concisely with only a small subset of its commands, such as C, D, K, T, L, I, N, FN and the basic loop structure. There are, however, a few pitfalls for the unwary; specifically, a certain class of operations, when performed in the most direct way, can be very inefficient. Furthermore, certain more intricate problems can be solved very simply if one knows, or at least knows of the existence of and can look up or enquire about, a much more extended set of commands. The purpose of this article is to warn of some of the pitfalls and, in pointing out how they may be circumvented, to engender an appreciation of the power and utility of some of the more advanced commands.

In order to understand how it is possible to be inefficient in TECO it is necessary to have some understanding of how it functions internally. Editing is performed on a core-resident text buffer that is filled from and emptied to the input-output buffers. In default of direct commands TECO will manage the flow into and out of the text buffer itself as needed and so this schema will usually be transparent to the beginning user. The limits of the text buffer are only appreciated when occasional attempts are made to type or delete text beyond them, and the commands whereby its size can be manipulated by demanding input to or output from it are, for the most part, unknown. The importance of the text buffer to the consideration of efficiency arises from the fact that, each time any element of a command string causes it to grow or shrink by the addition or deletion of characters, it must be completely repacked. Most inefficiencies are characterized by approaches that repeatedly and unnecessarily insert/erase small numbers of characters into/from the text buffer. It is this class of inefficiency that I shall address here.

Before passing to some specific examples, I should briefly mentioned another factor that might be perceived as a potential source of inefficiency. This is that most TECOs are interpreters. They take each element of a command string, decode it to determine the required action, then perform that action. If the element is in a loop it is decoded in each iteration; there is no storage of compiled (decoded) instructions, as for example is the case for a FORTRAN or BASIC plus-2 program. Although it is always desirable to use as concise an equivalent command string as possible, especially within loops, this factor is seldom a significant one, particularly for the beginning user. The reason is the very conciseness of TECO commands: the overhead of interpreting them is in general a small percentage of the time consumed in executing them.

Let me illustrate the problems associated with repacking TECO's text buffer with a real-life example. I have a FORTRAN program originally designed to output the numerical tables it calculates to a terminal as efficiently as possible, in terms of type-out time. Each column is six characters wide and columns are separated by a single blank. To retain as much precision as possible output formats are constructed internally, and for simplicity each element of the table is output separately, suppressing all carriage control. For example, if X is determined to obey $100.0 \leqslant X < 1000.0$, it is output by:

```
WRITE (KB,FMT2)X
```

where the array FMT2 contains ('+',F6.2,$)(equivalent to print #KB, using "###.##", X;). As so often happens, this program was later modified to fulfill a purpose other than that for which it was originally designed. Its output became too voluminous to direct to a terminal and was therefore sent to a disk file for later printing. Unfortunately, in our FORTRAN the character $ at the end of the variable format only suppresses carriage control when output is to a physical terminal. In the disk file each element of a row, such as X above, occupied a separate line beginning with +. When the file was output to the printer with the interpretation of the first character of each line as FORTRAN carriage control, all of the columns were overprinted on top of one another.

The most immediate solution to this problem was plainly to search the file for and delete each of the very frequent occurrences of the character string <carriage return> <line feed> +. The simplest way to do this is to use the A command repeatedly to read the whole file into the text buffer, for reasons explained below, and then search-replace the string with an FS command. Since it is being replaced by nothing that is deleted, a delimited search-replace is the most elegant construct:

```
*<@FS/
+//:>EX$$
```

Execution of this procedure with a small test file of only 20,000 characters and 2000 occurrences of the string to be deleted took 32.6 seconds of CPU time (KI10 processor). The reason for this excessive time is plain: for each string deleted all subsequent characters in the text buffer had to be repacked equivalent in this case to unpacking and repacking the whole 20,000 character buffer 1000 times. Since the time taken by this procedure grows as the square of the size of the file, it is plainly inapplicable to cases of realistic size.

This approach can be improved by working with a text buffer of normal size, without appending to it, and using the FN command to search through the file, in place of the FS in the above example. A problem with this is that TECO will, if it can, on filling the text buffer terminate it at the end of a line. Therefore, a small proportion of the <carriage return> <line feed> + strings are split across consecutive text buffers and therefore not found in the first pass through the file. Thus, the file must

be repeatedly searched, reporting each time a string is found in sweeps after the first, until none are found (this is the reason that reading the whole file into the text buffer in the first solution so much simplifies it). Thus, after the first pass through the file, we do:

```
*<@FN/
+//:↑ AX↑ A>EX$$
```

For the test file described above it was on the fourth sweep of the file that no Xs were typed out. That is, no example of the string was found and most probably, but not certainly, all occurrences of it has been deleted. This took a total of 16.3 seconds. Although this time only grows proportionately to a power of the size of the file of about 1.3, it is still unacceptably large for routine practical use.

To obtain a really efficient solution to this problem one must control the emptying and filling of the text buffer oneself. The following example shows how this can be done:

```
*<OUA< S
+$: QA,.-3P .UA >B,QAK ZUZ A (QZ-Z)"E-1; ' >EX$$
```

The outer loop initializes the inner loop by storing zero in Q-register A, OUA. The inner loop then searches the current text buffer for occurrences of the string with the S command. Each time one is found, the text from the end of the previous occurrence to the beginning of the current one is copied to the output buffer without modification, QA,.-3P. The pointer position at the end of the current occurrence is stored in Q-register A, .UA. When no more are found, the inner loop exits, ; and the outer loop continues by deleting all of the text from which that to be retained has been copied, B,QAK. The number of characters left in the text buffer is recorded in Q-register Z, ZUZ, and more text, if available, is read into it from the input buffer, A. A test is made to determine whether the number of characters in the text buffer has changed, (QZ-Z)"E, and if it has not, the task is over and the loop is exited, 1;. Otherwise, ', the outer loop is reiterated.

This operation took 5.0 seconds of CPU time on the test file. It only requires a single pass of the file, and the time required depends linearly on the file length.

Very similar procedures can be applied to cases where repeated insertion, rather than deletion, is required. As an example, a large, profusely commented FORTRAN program had all of its comments neatly enclosed in boxes of asterisks:

```
C**********************************************
C*                                            *
C*          . . . . . comment . . . . .       *
C*          . . . . . comment . . . . .       *
. . . . . etc. . . . . .
```

In the course of extensive improvements to the program the comments were further expanded and modified beyond recognition. This was done in free format and the net effect was to shift the right-hand margin of asterisks to the left and make it extremely ragged. It became plainly desirable to restore the readability of the program and comments. Since the parent program came from cards, the original line length was 80 characters.

To realign the right margin, a line of 78 blanks was inserted at the beginning of the first text buffer and then:

```
*<80UA<S
C$: .UB L 3R .UE (QB-QE+78)UN QN"N QA,.P B,QNP .UA '>
80,QAK ZUZ A (QZ-Z)"E 1; ' >B,80K EX$$
```

The outer loop is very similar to that of the previous example, except that the first 80 characters of the text buffer contain the line of blanks (including carriage return and line feed) and are ignored. In the inner loop comment lines are sought, identified by <carriage return><line feed>C. If found, the pointer position after the C, .UB, and that before the terminal asterisk, L 3R .UE, are stored in Q-registers B and E respectively. The number of missing characters is calculated in Q-register N, (QB-QE+78)UN, and if non-zero, QN"N, the new text prior to the terminal asterisk, QA,.P, and sufficient extra blanks, B,QNP, are copied to the output buffer. Q-register A is then updated with the current pointer position, .UA. Note how much more efficient the insertion of the blanks is made than with the simpler equivalent QN <I $> , in which for each iteration the text buffer must be repacked.

It should be plain that the techniques illustrated in the two examples discussed above for deletion and insertion are equally applicable to replacement or in fact any mix of the fundamental processes. My third and final example addresses replacement of strings of variable length. It concerns a problem that we had in bygone days reading 7-track even parity BCD tapes generated on *BM machines. (Since the growth in personal computer use is likely to make this journal the subject of family reading, I have eschewed obscenity. In this case the asterisk replaces the operative vowel to protect those of tender years or sensibilities.) Because a zero frame at even parity has no bits set and therefore cannot be used for synchronization, their character code was modified so that all blanks on them were replaced by % characters. The problem of course was to restore the blanks.

We procede much as in the previous example, placing at the beginning of the first text buffer a line of blanks greater than the longest contiguous string of % in the file, say 148 in number. Then:

```
*<150UA<S%S:QA,.-1P .UB ZUA :S↑ N%$"S(.-1)UA '(QA-QB+1)UN
QAJ B,QNP >150,QAK ZUZ A (QZ-Z)"E 1;' >B,150K EX$$
```

This command string follows principles very similar to those discussed in the first two examples, and I leave it as an exercise for the reader to work out how it performs the required task. The command :S N%$"S searches for any character that is not % and returns a truth value that is tested for success in the search.

I hope that the above examples have given some idea of the efficiencies that can be achieved with TECO when many repetitive modifications must be made to one file. In general, in such cases the most efficient procedure is not the simplest and involves more advanced commands. The choice whether to devise and use it will depend on the length of the file, the frequency of the required modifications, and how often in the future the same or a very similar procedure will need to be employed. However, a knowledge of these more advanced commands will always be found to be useful. In the second example the calculation of the number of blanks to be inserted to justify the right margin could not have been made without them, and the corrections either could not have been made, or would have required time-consuming manual or trial-and-error counting.

INTRODUCTION TO RSTS DIRECTORIES

By Scott Banks, Nationwide Data Dialog

1.1 To begin with . . .

This is the first installment in a series of articles whose purpose is to explain the RSTS directory structure. Our goal here is to begin modestly, procede with diligence, and conclude with a really good understanding of directories.

Generally, documents on this subject assume the reader already knows a great deal and is simply looking to clarify some particular facet of directory structure. By contrast, we will cover as much basic theory as possible as well as delving into the details. Follow along with us, and you will be able to put our notes and examples to actual, practical use. Understanding the directory structure and how RSTS plays with it gives system programmers an edge in reaping machine performance. As an added bonus, you can enhance your crash recovery arsenal with some tried-and-true (albeit last resort) repair and verification techniques. Although directory tinkering is always considered the domain of the very brave or the very stupid (depending on the outcome), it's nice to own the confidence that you can try something, heroic yet reasonable, before resorting to back-up tapes.

1.2 UFDs and the MFD

Let's start with some fundamental definitions. Each account (on each disk) has its own directory structure. This is called a UFD, for User File Directory. To illustrate, the UFD for [1,2] contains a list of the files belonging to that account, information including file size, protection code, and other items that appear on a DIRECT or PIP listing, are contained in the UFD. One of the critical pieces of information in the UFD, and one not currently displayed by VO6C directory listing programs, is the physical location of disk files. This is what RSTS needs to know when actual reading and writing of data files is to take place.

In order to find any given data file, the monitor must search the proper UFD for a specified file name. Before this can happen, RSTS must know where on disk the UFD itself is. A non-obvious fact about UFDs is that they themselves are files. This is an important consideration in dealing with them. The Master File Directory, or MFD, is a list of UFDs, their location on disk, and other account related information. There is exactly one MFD per disk and RSTS always knows where it resides on the disk. (The MFD begins on device cluster 1, one of the very few items whose disk location is constant.) The structure of the MFD is identical to that of the UFDs, although the meaning of some data elements is altered. This feature allows most of the same routines, whether in the monitor or in a user program, to access either the MFD or any UFD on any disk.

Account [1,1] is the MFD. It doubles as the UFD for files stored in account [1,1] as well. This is possible because, of all the files in [1,1], the ones that are really UFD entries (as opposed to user data files) are flagged as such. This is why we never see anything except data files in [1,1]. When searching the MFD for a specific UFD, RSTS ignores data files. In the MFD (or [1,1] account if you prefer), the first entry is a UFD entry for [1,1] and it points to itself. Looking for a file in [1,1] still requires a search of the MFD for the [1,1] UFD. This is a result of the fact that only the MFD can be found by directly reading a known spot on the disk.

1.3 The Great and Mighty FIP

There is a section of the RSTS monitor called FIP, the File Processor. Mostly, FIP worries about directories. When creating, extending, or deleting a file, you are relying upon this nimble code to keep your directory in good shape. FIP knows how to find the MFD and each UFD, and it knows how to retrieve, insert, remove, and modify both data file and UFD entries.

FIP is considered serial in that it performs a task for only one user job (or the RSTS monitor) at a time. It doesn't go on to another task until it brings things to a logical conclusion. Since a directory is a linked-list, this gives RSTS the security it needs for multi-user file additions and deletions. Rule number one about directories is . . . "Look, but don't touch." The confusion that results when you cross wires with FIP can cost you an account or worse — a disk. However, there is nothing at all harmful in the act of reading a UFD or MFD and seeing what's there.

Clearly, RSTS spends much of it's time doing FIP operations. At SYSGEN time, 'FIP Buffering' may be selected. This option permits FIP to use a special section of main memory, called

XBUF, to cache directory information. FIP attempts to keep recent directory blocks in memory, thus avoiding a disk transfer each time a section of some directory is required. When a directory must be updated, the data is rewritten immediately to keep the disk current — essential in the event of a crash.

1.4 Device Clusters

Allowable RSTS clustersizes are positive integer powers of 2 up to 256. Each disk has it's own pack clustersize. The minimum pack clustersize for a given disk is directly related to the number of physical disk blocks on the device. RSTS demands that it be able to specify any cluster on the pack by using an unsigned 16-bit integer. This sets a limit of 65536 device clusters, 0 through 65535. For example, an RK05 (2.5MB) has a minimum pack clustersize of 1, while an RP06 (176MB) requires 8 blocks.

The 16-bit integer used to describe a physical cluster is known as a device cluster number, or DCN. All disk references are based upon the DCN, data files and UFDs alike. To reference a single block, one must know the device cluster in which it resides. Knowing which of the (possibly) several blocks it is within that device cluster completes the deal. User programs related to data files by block number, a far easier scheme for programmers. The directory holds the device cluster information.

Each data file has it's own cluster factor and list of device clusters that compose it's physical existence. The clustersize for any particular file is minimally the pack clustersize. If the cluster factor for a file is greater than the pack clustersize, every entry in the list of DCNs is really the first DCN for the larger clustersize. As clusters (whatever their size) are always contiguous blocks on disk, it's then easy to identify any DCN within a large cluser. One of the reasons that clustersizes are defined as powers of 2 is to guarantee that some even number of small clusters will fit into a larger cluster.

UFDs also have a cluster factory and list of device clusters. The MFD, as cited earlier, is responsible for retaining said items. Like data files, the UFD clustersize can never be less than that of the pack. The maximum, however, is 16. This limitation is based upon the nature of the directory structure. Another limitation of UFDs is that they may have no more than 7 clusters. We'll see why this is so a little further on. For now, concentrate on that fact that a UFD is built just like a data file — a cluster factor and a list of device cluster numbers.

Coming next issue . . . **Inside the UFD**

DEC TIMESHARING (1965)

By Peter Clark

The majority of DEC timesharing customers are relative newcomers as far as timesharing is concerned. Many may also be newcomers to Digital as well. The University of Pennsylvania Medical School Computer Facility (MSCF) has been a DEC customer since 1965. In fact, we purchased the first timesharing system available from DEC, the PDP-6.

Before 1965, the MSCF owned a Control Data 160-A. The 160-A would probably be classified as a moderate sized mini today. It was a single user machine with no batch processing capability. Programmers and/or users would go to the computer room, sit at the console, load the paper type bootstrap and start up a program. This was our environment when we first started to hear about "timesharing". Of course, my first thought was, "It sounds OK but it can't possibly be as good as having the entire computer to yourself. Besides, who the hell is Digital Equipment Corporation and what is a PDP."

We soon found out that DEC was a computer manufacturer in Maynard, Mass. and PDP was a "Programmed Data Processor". At the time, there were not many places to go to see a PDP-6, (I think DEC made 24 of them). We were lucky. Applied Logic Corporation in Princeton, NJ already had one.

My first session on a timesharing system was at Applied Logic to try to convert a large FORTRAN system. The conversion from CDC FORTRAN turned out to be easy. It would be with FORTRAN II.

At first exposure to the terminal (Teletype Model 33), I was a bit apprehensive about screwing up other users. I was assured that I could type anything I pleased and the system would not crash. This turned out to be very nearly true. I was sold on timesharing when I realized that I could do everything I could do on the 160-A and then some.

When our own PDP-6 came a few months later, we became "true" customers. We soon found out what the deficiencies of the system were. The first problem turned out to be with DEC's delivery system. The driver of the truck that picked up our system from Maynard drove his 14 foot truck under a 12 foot bridge. Most of the computer survived. Our delivery date was moved back a few weeks, and the printer (128 columns, 600 LPM from Analex) ran for two months naked: No case.

The configuration we ordered had 64K words of memory (36 bit words), four dual dectape drives, 2 556 BPI magtapes (7 track), a 600 LPM printer and a 300 CPM card reader. You will notice the absence of any disk. This meant that any system access required reading a magnetic tape device. In those days, the dectapes were used for that. The dectape was used rather like a random access device. One could read or write in fixed blocks on the tape in sequential or random fashion. The access time was measured in seconds or possibly minutes if other drives were in use.

There was no such thing as swapping jobs which meant we had to sign up for the amount of core required. You reserved a terminal, XK of core and a time slot (max one hour). This assumed we knew how much core was needed which frequently was not true. There was no such thing as shareable programs in 1965. Every user that was compiling had to have his own version of the compiler in his core area. Of course, in those days, the FORTRAN compiler was 11 or 12K words and the operating system was about 6K words. Today our TOPS-10 operating system is about 60-70K words. We could effectively support about 4-6 simultaneous users with much bickering about who had more core than he should have, etc.

Cache, or semi-conductor memory was not in use at this time, but the "six" did have some fast memory. The first 20 (octal) locations of core were mapped into "fast memory"; a special set of locations that were accessed in the [nano-second] range (we never knew how). These locations were used as fast accumulators. The real first 20 locations of core were inaccessible to the programmer and were known as "shadow memory". This shadow memory had a very important use; there was no bootstrap module in this machine, so the bootstrap was kept in the shadow memory. The procedure was to use the shadow memory to read a paper tape, which read the monitor off a dectape. When hardware failures occurred, it was likely that the shadow memory would be wiped out, necessitating that it be toggled in through the console switches, all 16 instructions (36 bits = 36 switches). The then system manager (still is!) had a success/failure ration of about 10-1 favoring failure, so it was up to the resident CE to toggle it in. (He had it all memorized, which should give you an indication of how often it was used.) All of our CE's were good, but the best was something to see... Arthur Rubenstein (he plays the piano) never looked this good! His hands moved so fast, it was impossible to see each individual movement. Although he now heads up the Cherry Hill, NJ, Field Service office, he still has the shadow memory bootstrap for the PDP-6 memorized.

The sequence to compile, load and execute a FORTRAN program was as follows (The (".") dot, as RT11 people know, is the monitor prompt):

```
                        .R F4
; get FORTRAN compiler in 12K of core

                *DTA1:PROG=DTA2:PROG1,PROG2,.....,PROGN
; wait another minute or so to compile
; sources from DTA2 and put the relocateable
; files on DTA1.  (Pray for no errors.)

                        *AL
; back to monitor

                    .R LOADER 30
; get loader in 30K

                    *DTA1:PROG
; load in program

                    *SYS:/S
; search system
; dectape for FORTRAN library routines
; wait a couple of minutes

                .SAVE DTA1:PROG
; save loaded program

                    .START
; begin execution.
```

```
?DEVICE LPT NOT AVAILABLE
; whoops, someone else is using the printer now. let's go
  see how long he will be.
```

The above sequence no doubt seems very archaic, and it was. But, at that time, disks were still a very new thing and anyway, DEC didn't sell or support one with the PDP-6. Shareable code couldn't really be supported with PDP-6 hardware even if it had been thought of. Queueing of input and output really needs a disk to be done effectively. We could have had printer output go out to magtape or dectape but there was no software available to make that process easy or manageable. i.e., schemes for labelling output on tape, etc. There were no command files supported. No "CCL" (concise command language). Today on the TOPS-10 system, to do the above compile, load and go sequence, you type:

```
.EX PROG1,PROG2,...PROGN
```

or

```
.EX @PROG.CMD ; command file
```

This capability, which came from a DEC customer originally, was not available until the very early 70's.

Two editors were popular then, LINED and TECO. LINED was a line number oriented editor that required line numbers on data that was to be edited. PIP provided switches to add and delete line numbers. It operated like the BASIC Plus monitor editor; i.e., add or delete whole lines. There was no way to change a part of a line. TECO (Text Editor and Corrector) was used by more sophisticated users (has this changed?) and ran not unlike today's version (yes, TECO is on all DEC computers) notwithstanding 15 years of refinements.

Multiplexing (we called it scanning) was handled by a PDP-8 in half duplex mode. Full duplex came with our first disk (RP02) and monitor. We had one dial in line and our ubiquitous CE rigged a red light to tell us someone was on. The "8" could not sense the carrier and had no way of knowing when someone hung up or was cut off, so it was not unusual for a person dialing in to find a job already attached to that keyboard; KB to you, TTY to us.

The next real breakthrough in timesharing for us was the purchase of an RP02 disk pack drive and disk monitor. Then we had real time-sharing. Jobs could now be swapped to the disk and we could easily handle 20-30 users. Terminal data was handled in full duplex mode and a new innovation was /xxx/ to show what had been deleted when rubout was typed.

In the 60's DEC field service was much different that today. We had a resident service engineer (CE). The card reader required almost constant adjustment and it seemed he was always busy fixing something. We used to estimate MTTR (Meantime to Repair) by the number of open cabinet doors, and there were a lot of doors. it seems he was always fixing our TTY Model 33's. When we moved to Model 35's, we thought we really had it made. 10 CPS was great if it was reliable. Now we see our service technician once a week for an hour or so. I wonder why our rates haven't gone down???

Current DEC timesharing systems have learned a lot from the old PDP-6 days. I hope you RSTS users appreciate the efforts of us old PDP-6 users.

A DATABASE SYSTEM FOR A HOSPITAL RADIOLOGY DEPARTMENT

Jack W. London, Ph.D. and Ronald Arenson, M.D., Department of Radiology,
Hospital of the University of Pennsylvania, Philadelphia, PA 19104

The Radiology Department of the Hospital of the University of Pennsylvania has developed an administrative database system using BASIC-PLUS under RSTS. The CGR Medical Corporation, an x-ray equipment manufacturer, has collaborated in the development of the software. This system is unique; no other Radiology database system handles in an integrated fashion all the various functions of a hospital Radiology department. These functions are: patient registration, radiological examination scheduling, tracking of radiographs (x-ray films), patient tracking, reporting, statistics, teaching files, and billing. In addition, this database system is also used by the hospital Medical Records Department for accessing patient records and assigning Medical Record numbers.

The hardware for this system consists of a PDP-11/70 with 160 KWords, two RP04 disk drives, one TU16 tape drive, two DH11 multiplexers, 20 CRT's (mostly VT52's), three LA36's (one being the console), four Diablo 1620 printers (300 baud), 24 Identacon pen readers, a Hewlett-Packard Optical Mark reader, and one line printer (300 lpm). An additional line printer and tape drive, as well as more memory, are on order.

USER-LEVEL SYSTEM DESCRIPTION

The software may be described in terms of functional modules. The patient registration module enables the Radiology Department receptionists to determine by name, number, or phonetic (IBM Soundex) searches whether a patient is already registered. For new patients, their name, sex, birthdate, address, telephone number, primary physician, and (if an inpatient) hospital location are entered via CRT.

The radiological exam scheduling module takes many facts into account. In addition to finding sufficient available time for a given exam in the proper exam room, consideration is also given as to whether the patient is ambulatory, the source of the patient (e.g., from the Emergency Room), and any special techniques to be used. This module also warns of any similar or conflicting exams already completed or scheduled for a patient. Once an exam is scheduled this module prints the appropriate exam documents on a Diablo printer, including a bar code which points to a given exam order record.

The x-ray film tracking module uses bar code labels printed by a Diablo printer and Identacon light pen readers to locate the jackets containing patients' radiographs. These film jackets are tracked to various locations within the department as well as external long-term storage lcations. Furthermore, the film tracking module records loans of film subfolders to physicians. Bar coded labels pointing to a film loan record are prepared upon receiving a request for films. This label is then scanned when the films are sent out and upon their return.

The patient tracking module makes certain that patients do not become "lost and forgotten" after arrival within the department. The system is notified of the arrival of a patient by a receptionist using a CRT. A "clock" is then started, and if the patient has not had his exams completed within the expected time, his name is automatically entered on a "delayed patient" list. This list is displayed at regular intervals on a CRT for review by the head radiographer, who can investigate the cause of the delay. It should be noted here, that the system is notified of the completion of an exam by the quality control radiographer scanning the bar code printed on the exam document.

The reporting module enables transcriptionists to enter the radiologist's findings from dictaphone to the system using CRT's. Word processing ability is supplied by software (the terminals are "dumb"). An automatic mode of reporting, in which the radiologist scans a bar'coded standard report and then the exam card bar code, is also available for the many "normal" reports. This mode of reporting bypasses the transcription pool, and is therefore very efficient. With either mode of reporting the computer generates the final hardcopy, ready for distribution.

The statistical module generates reports on film usage, patient delays, radiation exposure, and repeated exams. These reports can be generated with many combinations of sub-categories, such as staff radiographer, exam room, patient source, and exam type. The data for these reports are obtained from the exam order data base file along with information read from mark/sense cards, which are filled out by the radiographers when the exam is performed (for example, they record the radiation factors on the mark/sense cards). The film usage reports serve as inventory control.

The teaching file module enables interesting cases to be readily referenced by standard anatomical and pathological codes (as well as by patient). Coding and diagnostic information is entered via mark/sense cards.

The billing module prepares a magnetic tape with daily charges for the hospital data processing department, which does all inpatient billing. Income to the department has increased since the computer system went on-stream, since the department now has an organized means of accounting. The system also prepares charge documents and bills for outpatients, who are billed by the Radiology Department billing office. Currently, an accounts-receivable module is under development, along with programs to prepare third-party (e.g., Blue Cross) forms.

SOFTWARE DESCRIPTION

We currently are operating under RSTS version 6C. All programs are written in BASIC-PLUS. Programs are designed to occupy 8K or less (this design goal has been met in almost all instances). About 30 users are timesharing during prime shift.

The database resides on the two RP04 disk packs. All database disk accesses are performed by a single database manager progrm (DBMGR), which runs with priority zero. A user-level program wishing to modify the database communicates with DBMGR using the RSTS message Send/Receive

system calls. The system calls are packaged in a function (function D), and a function call within the body of the program is all that is needed to access the database. For example, to obtain a patient's basic data record (i.e., name, sex, birthdate, etc.) a program would execute the following statement:

```
IF FND% ("GD", "PREC", 3%,PATN.KEY$) <> 0, GOTO 32700
```

This database function call requests a specific block and record from disk file "PREC" (the "GD" specifies a "get direct" operation). The block and record pointers are contained in the string PATN.KEY$. The data is to be deposited into the null device I/O record defined on channel 3. The key PATN.KEY$ was most likely obtained by a previous database call to a key file such as "NAME", using the first six letters of the patients last name as a key. (Other key files correspond to the patient's hospital or department number, and the soundex code for the patient's last name.) A nonzero value for FND% indicates that a database error has occurred (e.g., bad key, empty record, etc.). In the above example the error handling code begins at line 32700.

Output of exam documents and other printed forms which require bar-code printing by one of the Diablo printers is spooled by a program called SUPERQ. This program, which communicates via Send/Receive system calls with the user-level programs, determines the proper Diablo to use for printing, regardless of the source terminal of the request.

The light pens used for reading the bar-codes are multiplexed (16 pens may be multiplexed per multiplexer). Each of the two pen multiplexers appear as a terminal to the system, and are interfaced through one of the DH11's. The bar-codes are Identacon 2/5 code.

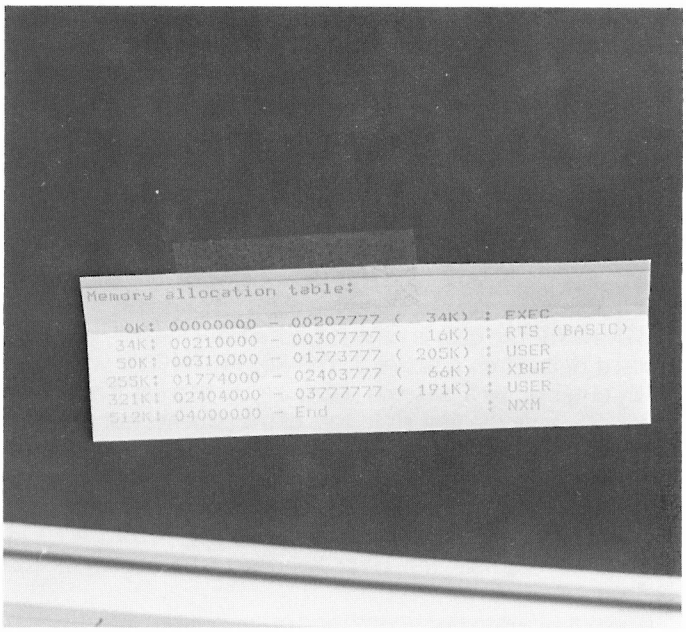
All user-level programs operate from a non-privileged account. To use the system, a person must enter their personal password, in addition to the account password. Confidential patient information (e.g., reports) requires entry of a social security number.

Each evening the disk database is backed up onto magnetic tape. Three 3200 foot reels are required. The 1 hour and 20 minutes necessary for the backup is the only regularly scheduled downtime for the system. (A disk-to-tape backup program written by CGR is used, rather than DEC's backup. A weekly backup of non-database disk files does use DEC's backup, however.) Except for when the database is down for backup, a journal tape logs every modification to a database file after the disk I/O has been completed. Thus, should a crash occur, the database can be restored to its pre-crash state by reading the 3 backup tapes, followed by the journal tape. "Father" and "grandfather" backup tape sets are kept (one set in a safe in another room).

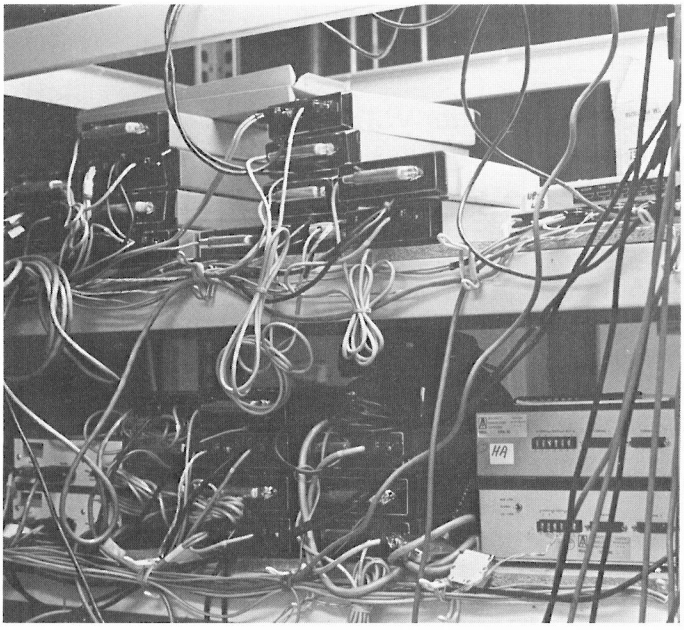
The Radiology Department database system has been in full-time operation since September 1, 1977. The use of RSTS and BASIC-PLUS has been successful in this application. The system is fairly close to being fully developed, with primarily the accounts-receivable being the only module awaiting development. It is our hope that this and similar systems be commonplace in Radiology departments in the future.

NAME THE TERMINAL CONTEST:

We'll give a prize to the first individual to identify this logoless terminal.



You say you're running three partitions?



Who is talking with which and to whom?

The Great Imposter

DOUBLE PRECISION INTEGERS

By Dave Mallery

WHAT: A double precision integer is a signed, 32-bit binary number stored in four bytes. It has a range of precision of -(2**31) to 2**31-1.

WHY: Commercial software running under RSTS must use double precision floating point. These require 8 bytes of disc storage for each one when using CVTF$ format. Double precision integers will store almost as large a number in only four bytes as there are no wasted bits for floating point characteristics. Massive disc space savings are certain in any commercial application.

HOW: Two user-defined functions are utilized. FNDI$(z) will convert a floating point operator to a four-byte string. EG:

```
LSET TOT.DOL$ = FNDI$ (TOTAL) ! CONVERT TO DOUBLE
```

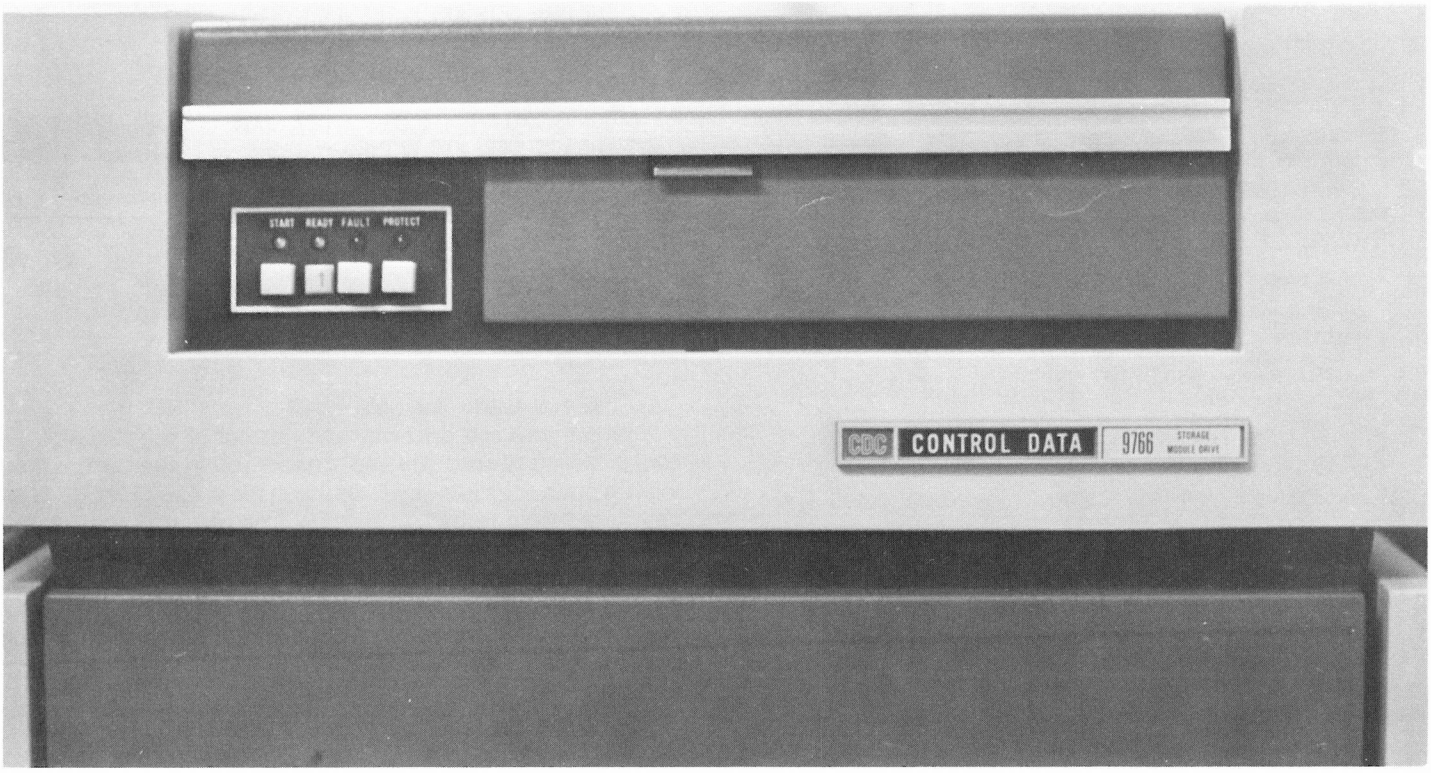
FNDI(Z$) will convert such a string back to a floating point operator:

```
TOTAL=FNDI(TOT.DOL$) ! MAKE FLOATING
```

LAST WORDS: This code will compile in BASIC+ or BASIC+2. The integers produced are not compatible with COBOL computational or the 'long' integers of the FPP. The code could be restated using SWAP% to produce a fully compatible integer.

CREDITS: The code presented consists of a 'final' re-write by Scott Banks of NDD of a very old function coded by the author in 1975 and presented in the SIG newsletter.

```
1         EXTEND

20010     !    FNDI, FNDI$ DOUBLE INTEGER CONVERSION FUNCTIONS &
          !                                                    &
            DEF FNDI$(Z)                                       &
          / Z1%=FIX ((Z+2147483648.) /65536.)-32768.           &
          / Z=Z-65536.*Z1%                                     &
          / Z=Z-65536. IF Z>32767.                             &
          / DI$='....' UNLESS LEN(DI$)=4%                      &
          / LSET DI$=CVT%$(Z1%) + CVT%$(Z)                     &
          / FNDI$=DI$                                          &
          / FNEND                                              &
          !

20020       DEF FNDI(Z$)                                       &
          / Z=CVT$%(RIGHT(Z$,3%))                              &
          / Z=Z+65536. IF Z<0                                  &
          / FNDI=65536.*CVT$%(Z$) + Z                          &
          / FNEND                                              &
          !

32767     END

Ready
```



Three-hundred megabytes on an 11/70.

GAMES

By Joel Schwartz, M.D.

You're stuck on the Expressway traffic again and it's a toss up — Who will boil over first, you or your car? Your mind drifts for a moment.

```
RUN BOMBER
YOU ARE NOW A PILOT IN A WORLD WAR II BOMBER
WHICH SIDE -- ITALY(1), ALLIES (2), JAPANESE(3), GERMANY(4)? 2
AIRCRAFT - LIBERATOR(1), B-29(2), B-17(3), LANCASTER(4)?

YOU'RE BUSTING A GERMAN HEAVY WATER PLANT IN THE ROHR.

HOW MANY MISSIONS HAVE YOU FLOWN?
```

15, I'm a real ace.

You're back. The traffic seems to be easing. You turn on the radio to get some news. "The market was down 15 points in reaction to the raising of the prime rate to 12.75%. The biggest losers were IBM, RCA . . . " You can't bear to hear any more so you switch to music.

```
RUN STOCK
THIS PROGRAM PLAYS THE STOCK MARKET. YOU WILL BE GIVEN
$10,000 AND MAY BUY OR SELL STOCKS. THE STOCK PRICES WILL
BE GENERATED RANDOMLY AND THEREFORE THIS MODEL DOES
NOT REPRESENT EXACTLY WHAT HAPPENS ON THE EXCHANGE.
```

(I'm not so sure it doesn't happen that way! . . .

STOCK	INITIALS	PRICE/ SHARE
INT. BALLISTIC MISSILES	IBM	109.25
RED CROSS OF AMERICA	RCA	79.5
LICHTENSTEIN, BUMRAP & JOKE	LBJ	170.
AMERICAN BANKRUPT CO.	ABC	151.25
CENSURED BOOKS STORE	CBS	115.25

NEW YORK STOCK EXCHANGE AVERAGE: 125.05

WHAT IS YOUR TRANSACTION IN
IBM? +85

★★★★★ END OF DAY 1'S TRADING ★★★★★

STOCK	PRICE/ SHARE	HOLDINGS	VALUE	NET PRICE CHANGE
IBM	114.	85	9690	4.75

★★★★★ END OF DAY 2'S TRADING ★★★★★

STOCK	PRICE/ SHARE	HOLDINGS	VALUE	NET PRICE CHANGE
IBM	132.5	85	11262.5	18.5

Terrific! I'm rich!

You're almost home. You could kill your boss for giving you that extra work. That S.O.B.!!! But you dare not say anything.

```
RUN BOXING
OLYMPIC BOXING -- 3 ROUNDS

INPUT YOUR OPPONENTS NAME
? THE BOSS
INPUT YOUR MAN'S NAME
? JOEL

DIFFERENT PUNCHES ARE, 1 FULL SWING, 2 HOOK, 3 UPPERCUT,
4 JAB
WHAT IS YOUR MAN'S BEST? 3
AND WHAT IS HIS VULNERABILITY? 4

THE BOSS' ADVANTAGE IS 4 AND VULNERABILITY IS SECRET.

ROUND 1 BEGINS . . .

JOEL'S PUNCH? 1
JOEL SWINGS AND HE MISSES

JOEL IS ATTACKED BY AN UPPERCUT (OH, OH) . . .
AND THE BOSS CONNECTS . . .
JOEL IS ATTACKED BY AN UPPERCUT (OH, OH) . . .
AND THE BOSS CONNECTS . . .
JOEL'S PUNCH? 2
JOEL GIVES THE HOOK . . . CONNECTS . . .
JOEL'S PUNCH? 2
JOEL GIVES THE HOOK . . . CONNECTS . . .
THE BOSS GETS JOEL IN THE JAW (OUCH!)
. . . . AND AGAIN!

JOEL IS ATTACKED BY AN UPPERCUT (OH, OH) . . .
BLOCKS AND HITS THE BOSS WITH A HOOK.
THE BOSS GETS JOEL IN THE JAW (OUCH!)
. . . . AND AGAIN!
JOEL IS KNOCKED COLD AND THE BOSS IS THE WINNER AND
CHAMP.

AND NOW GOODBYE FROM OLYMPIC ARENA.
```

Just like at work. The boss wins!

You pull into the driveway and get out of the car. A frizby comes sailing toward you and smacks you in the temple. You look up but no one is in sight. You open the door and yell "Hi dear, I'm home," only to be told that because you were 30 minutes late, no one could wait, dinner is on the table and she'll be down to sit with you after she gets off the phone. You eat alone, in silence. After dinner you go to your study where you hang out a "Do Not Disturb" sign and lock the door. You dial 789-7146 and put the phone down on the coupler. You log in and type

```
RUN ADVENTURE
```

Yesterday you were at a complex junction . . .

GO WEST

A LITTLE DWARF JUST WALKED AROUND A CORNER, SAW YOU, THREW A LITTLE AXE AT YOU WHICH MISSED, CURSED, AND RAN AWAY.

YOU ARE IN BEDQUILT, A LONG EAST/WEST PASSAGE WITH HOLES EVERYWHERE. TO EXPLORE AT RANDOM SELECT NORTH, SOUTH, UP, OR DOWN.

THERE IS A LITTLE AXE HERE.

TAKE AXE.

OK.

There is a knock at your door. "Dear, how was your day?" "Fine," you say to your wife. "Listen, I'm on an important call. I'll be with you in about half an hour." She leaves.

WEST

YOU ARE IN SWISS CHEESE ROOM.

WEST

YOU ARE AT THE EAST END OF THE TWOPIT ROOM.

WEST

YOU ARE AT THE WEST END OF THE TWOPIT ROOM.

WEST

YOU ARE IN THE SLAB ROOM. THERE IS A THREATENING LITTLE DWARF IN THE ROOM WITH YOU. ONE SHARP, NASTY KNIFE IS THROWN AT YOU. MISSES.

Another knock. "Daddy, can you help me with my math?" "I'll be with you in half an hour. I'm involved in something important," you say and you hear the footsteps going away from the door.

THROW AXE.

YOU KILLED A LITTLE DWARF. THE BODY VANISHES IN A CLOUD OF GREASY BLACK SMOKE. THERE IS A LITTLE AXE HERE.

TAKE AXE.

OK

UP

YOU ARE IN A SECRET NORTH/SOUTH CANYON.

SOUTH.

YOU ARE IN A SECRET CANYON WHICH EXITS TO THE NORTH AND EAST.

A HUGE GREEN FIERCE DRAGON BARS THE WAY!

THE DRAGON IS SPRAWLED OUT ON A PERSIAN RUG!!

Somehow the hastle of the traffic, the news of the stock market, your boss and your family don't seem as important as they did a little while ago. You look over the keyboard and a gleem comes to your eye. You raise your hand and slowly, but with some authority, type in

KILL.

PROGRAMMING STANDARDS

By Scott Banks, Nationwide Data Dialog

1. Goals of Programming Standards

1.1 Introduction

The purpose of this document is to introduce and define programming standards for Nationwide Data Dialog. With programming standards, every aspect of the programming experience will benefit.

This can occur only if the standards are good and if they are followed. The ideas presented here are not known, nor intended, to be inherently correct. Rather, they are an attempt to provide an organized, repeatable set of rules. These ideas should be expected to evolve, yielding better ideas as the old ones are applied more severely. If the ideas presented here are applied and modified in an orderly fashion, better ideas will certainly result. Perhaps in the meantime, all of us will better be able to read and debug each other's code, as well as our own.

Program structure is an important issue. Wherever possible, an effort has been made to achieve a bond between the code standards and good program structure. In addition, programming design and implementation guidelines receive some separate attention.

1.2 Standard Routines

The Programming Standards are heavily oriented towards the concept of standard routines. These are functions and subroutines that have been designed to general and tested to be reliable. Every standard routine has its own variables and line numbers. Special provisions are taken to ensure that variable usage and line number allocation can't conflict with application code.

A program is divided into two sections. The applications programming area is from line numbers 1 through 19999, with restrictions placed on the specific use of certain areas. The lines 20000 up to 32767 are devoted to standard routines. No application program is permitted to code above 20000.

Standard routines may be appended if they are needed. Entire functions of subroutines may be deleted during the course of the program design, but no standard routine may be modified. Those standard routines that do require custom tailoring use parameters defined before line 20000. This may include variables, arrays, DIM, DATA, etc.

A given standard routine will work the same way in every program in which it appears. Multiple versions may be offered to meet space or other design constraints. For multiple versions, the decision to use the same line numbers versus an independent set is based upon the needs of the particular routine.

The design and maintenance of standard code is the responsibility of those who really care. Ensuring simultaneously the many opposing qualities needed by standard routines is a non-trivial task. To name a few — unique variables and line numbers, generality, efficiency, ease of use, reliability, expandability, and upwards compatibility (for new versions).

1.3 Applications Programming

Many restrictions are placed upon applications programming. They are worth it. The common initial reaction of "This is too much trouble" is readily offset.

By defining line number allocation, a program becomes predictable as to just where to look for existing code as well as providing a place for adding code. The rules for variables provide, at the least, a grouping scheme. More than that, they provide a basis for determining which variables 'belong' to which modules. This is advantageous in design, invaluable during debug and maintenance.

The opportunity for programmers to write creatively is not only enchanced, it is encouraged. Looking for the best structures and techniques is still the most fascinating aspect of coding. One should never take the view that the standards are constraints. It's quite the opposite.

2. Variable Usage

2.1 Standard Variables

Standard routines have a set of variable names that can never conflict with those used by application code.

Standard variables are always two or more characters long, and never contain any single dots. If dots are used they must appear in pairs. This applies to integer, floating point, string, and array names. Standard functions will always be named by this convention.

A given standard function may leave useful data in a standard variable, making it available for the main program. Also, when specifically defined, a standard variable may be given a value by application code in preparation for a subsequent event.

In no case may application code use these variables for any other purpose, thus hampering the action of any existing or future standard routine. No code will rely upon the value of a variable that is not actually defined as the output of a standard routine.

In the design of standard routines, careful attention must be paid to wise selection of variable names. Conserving space while providing mnemonic symbols represents the major trade-off. Also, variables that interact with the system, or a planned extension, must be avoided. Standard Variables with no dots are expected to be those most commonly defined. When the two-dot approach is used, the characters preceeding the dots can be used as a grouping technique. The application variable section details this concept.

2.2 Application Programming Variables

The set of variable names to be used by application programs is that of all multi-character names with single dots between alphanumerics. No standard routine will ever modify any variable with single dots. it is expected that a single dot will normally be used, the multiple (but not contiguous) dot format is for those cases that require it.

This rule applies to all types of variables, arrays, and function names that are defined in application code.

2.2.1 Naming Convention

Application variables are subject to a naming standard, which exists for ease of expansion and transportation of code. As all application variables must contain at least one dot, that dot will be put to good use.

The basis of the naming convention is the concept that a variable 'belongs' to a specific routine or set of related routines. Normally, a variable will be defined as three characters, a dot, and three more characters. The first three characters will represent the group to which the variable belongs, while the second three will describe its specific use.

For example:

CUS.NAM$	Refers to the name field in the customer file.
CUS.BLK%	Refers to a disk block in the customer file.
SRT.TMP$	This string is used by a sort routine to hold a temporary value.

Note that the group specifier, CUS for instance, will usually refer to a data file. It might, however, refer to some section of code that performs a well defined task. In-core sorting, keyboard input, and field validation are common examples. By looking at a variable one can get an idea of not only its information content, but also where it has been assigned a value.

In most cases, this XXX.YYY approach will work well. Sometimes, many data items are fielded in a buffer, and the sensible combinations of three letter groups is limited. Simply go to an XXX.YYYY format, preserving the group name while expanding the specific usage identifier. This should be the exception, but it is available. If this path (or even three-dot-five) is taken, it should be logically applied. It should be used for all variables that appear together within the group, but not for all variables. The additional expansion path available is that of multiple, non-adjacent dots. Using three-dot-three-dot-three lends a nice structure to programs that need it (and can afford the space).

Another limitation is the bulk of the variable names themselves, as they do take up space and do require repeated typing. Shortening the group name to one or two charcters for highly used groups is allowable. If a great deal of fielding and/or reference is made to the CUS group of variables, shorten it up like this:

C.KEY$	Key to the customer file
C.PTR%	Pointer to the customer file

Be careful however, as this use may conflict with that described as 'Special Application Variables'. Check this section first.

2.2.2 Special Application Variables

Some variable types have a tendency to appear very often. File channels and loop indices are good examples. For these, a one-dot-three variable is used. These are presently assigned:

F.CUS%	File channel for file system using CUS as variable group name
F.CUS$	File name for above channel
I.XXX% thru N.XXX%	Loop index variable for group XXX, following the commonly used I%, J%, etc.

These names are relatively short and have the same protection against interaction as all application variables. These names apply to any arrays that may be needed.

The use of F.XXX% and F.XXX$ is a special case and is covered in the section relating to file channel definitions.

2.2.3 Temporary Variables

Certain variables are defined as temporary in nature, and may be used freely, according to their own rules. These variables are used by the standard routines to save space as well as indicate the transient quality of their values.

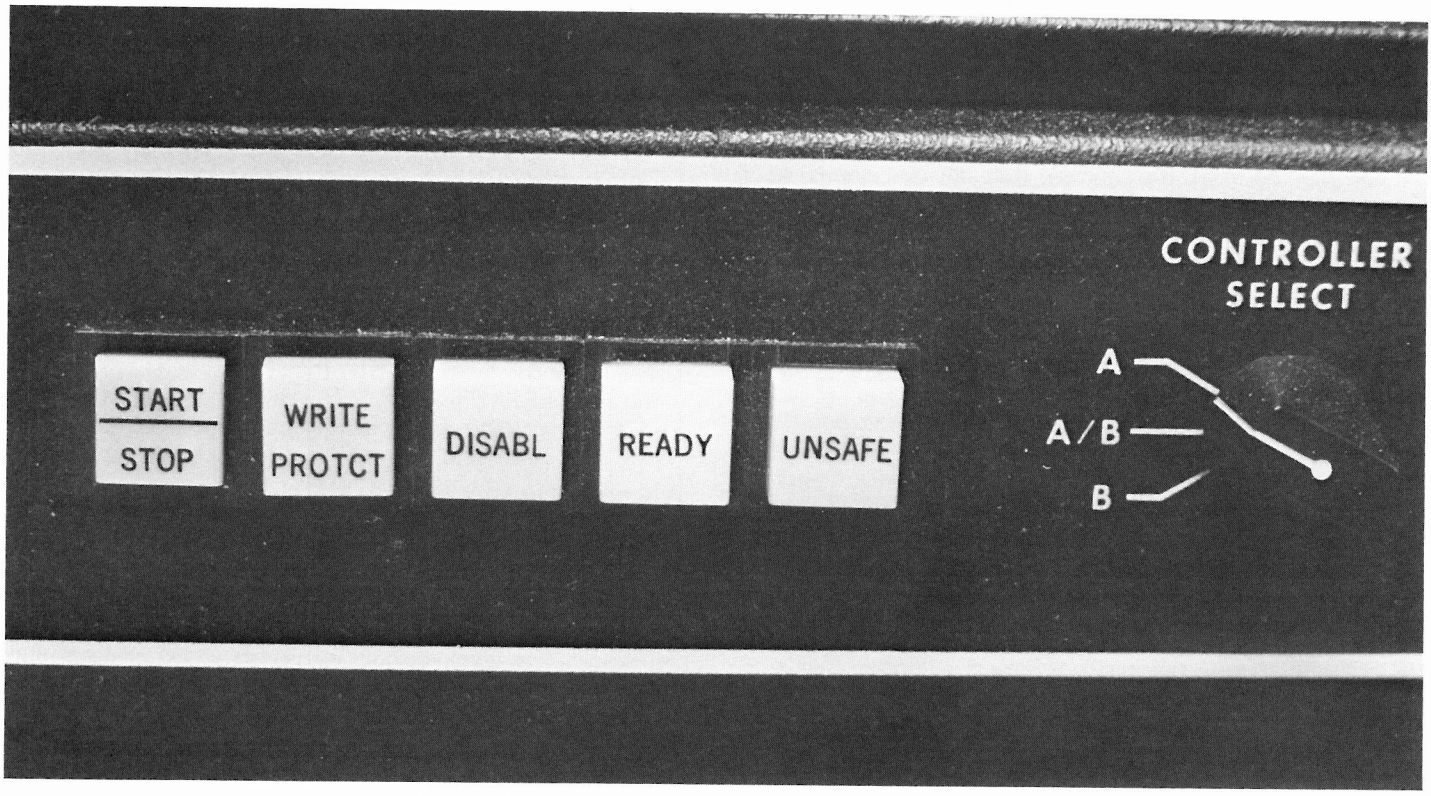
In application code, temporary variables are allowed, but their use is restricted. Temporary variables are for immediate use only. In no case may a value be assigned, a subroutine or function executed, and then that same variable expected to have the same value. Temps are best suited for intermediate calculations. They may be used to pass a parameter to another routine if that routine is called directly. They may also be used to return a value from a subroutine or function if they are tested immediately upon return.

Temporary variables are defined as a single letter, or a letter and then a single digit (non-extend mode variables). Do not use zero as the digit. An added bonus is that pre-extend mode code can be upgraded softly. Temps can take the form of integers, floating point, strings, and all kinds of arrays. Don't generate temporary function definitions.

E% is returned by many standard routines as a result of error trapping. But don't rely on the value of E% unless it is defined as returned by the routine. Use E% regularly as a return indicator. This should be the default unless there is some need for an additional application variable (perhaps the value is required later).

The following is a list of temporary variables that should be used before others (to save space). When possible, use these for their recommended purposes:

D$	Displacement place holder for fielding
E%	General error return from function or subroutine call. Positive values indicate a RSTS error of some kind, while negative values mean a program defined error or exception condition.
F%	Use this to hold a file channel if much fielding or I/O is performed in one section of code.
I% thru N%	Loop or other index usage
Z% thru Z9%	General temporaries. Use Z% first, then Z1% thru Z9% in order as needed.
Z thru Z9	General use.
Z$ thru Z9$	General use.



3. Standard Routines — Structure and Allocation

All lines 20000 through 32767 in a program are reserved for standard routines. No application programming may occur in this area. Line 32767 will always consist of END and nothing else.

As standard routines are developed and approved, their line numbers will be allocated as needed. The specific goal is to eliminate the need for program modifications to standard code.

All programs that need some specific standard routine will always have a spot for it. A routine will work the same way in every program that includes it, regardless of the programmer.

Some 'quick' program designs or enhancements may suffer. Four options are available. First, the feature in question may be disregarded. Second, further application programming may be employed to accomplish the desired result. Third, old standard routines may evolve to cover the special case. Finally, new standard routines may be developed.

. . . . to be continued

SHOULD YOU CONSIDER USING MACRO-11 UNDER RSTS/E?

By Richard A. Marino, Data Processing Design, Inc.

This article discusses some of the specific advantages of MACRO-11 as a programming tool under RSTS/E.

MACRO-11 is the common assembly language for the PDP-11 family of minicomputers produced by Digital. Its use is just now becoming 'respectable' under RSTS/E. Certainly it was possible under versions 6B and 6C to write programs in MACRO-11, however, only lately has it been acknowledged that doing so is acceptable. Version 7 of RSTS/E will finally mean true acceptability of MACRO-11, especially in the sense of providing reasonable documentation in the form of a System Directives Manual.[1]

If Version 7 permits all of the MACRO-11 hacks to come out of hiding, it will also no doubt encourage a wider group of companies and programmers to give MACRO-11 a try. In some ways using assembly language in these days of higher level languages is almost a religious commitment. Years ago we were promised that the days of higher level languages were upon us; though the promise was great, so was the dissension and dissatisfaction. While it is possible that languages like Pascal may in fact be a step away from assembly language, assembly language still offers the mystical attraction of dealing with the hardware directly.

Why use MACRO-11? It offers the advantages of all assembly languages — small in size, fast in execution. Admittedly, one can write slow and unnecessarily large assembly language programs (it may even be easier to do so in assembly language). However, one does have **total** control over the precise size and speed of an assembly language program.

Why use MACRO-11 under RSTS/E? If you have been using BASIC-PLUS, COBOL, DIBOL, or even FORTRAN, you may be in for a surprise. The good news is you can write small, fast programs. The bad news is they are a pain to write and debug. It is **not** realistic (in 99% of the cases) to have an entire application system in MACRO-11. What is a reasonable idea is to convert or write critical software in MACRO-11 for such a system.

RSTS/E with its file sharing and send/receive capabilities provides one of the best possible environments (short of virtual memory page sharing) to integrate a MACRO-11 program into a software system. As discussed previously in an article on performance measurement[2], the correct way to decide to convert part of an existing system to another language is to instrument the software and determine which parts of the application are using which system resources.

Where should MACRO-11 be used? It is ideal when the application would be helped by the following:

a. Smaller program (providing more memory space for buffering, indexes, etc.).
b. Fast execution of integer-arithmetic or simple character manipulation code.
c. Sharable code (either as run-time systems or resident libraries[3]).

MACRO-11 is not necessarily the best answer if one needs to run copies of the same program from several terminals simultaneously. First, consider multi-terminal I/O, a very powerful RSTS/E feature that can be effectively used in some applications requiring transaction or inquiry operations from several terminals. Second, consider using send/receive with small programs sending requests (in the case of data base inquiries) to a central and larger program whose task is to simply handle messages requesting information, read the data base, and supply information by return message. Finally, consider that the only way to create **sharable** code is to write a run-time system (or sharable resident library in Version 7). While I will not say writing a run-time system is dramatically more difficult than writing a MACRO-11 program, it is certainly harder to debug and there is less documentation about how to do it.

An example of a program that is a natural for MACRO-11 is a disk to tape backup system (SAVER) we developed to replace RSTS/E BACKUP. Written in MACRO-11 the disk save program is less than 5 KW of code (versus 16KW of multiple modules plus 15-16KW of BASIC-PLUS for RSTS/E BACKUP). In addition, the small program permits the use of memory for large disk/tape buffers. This is a true performance enhancement since using 16 x 512 byte buffers improves performance on a typical system save by reducing the number of disk accesses by a factor of 10 compared to RSTS/E BACKUP. The additional memory space (up to 27KW!) also permits caching of directory (UFD) information to further speed processing. Implemented as a run-time system to save the overhead of 2-4KW of the RSX or RT11 runtime systems and assure full control over error trapping, this program gains little by being sharable.

Other examples of systems implemented in MACRO-11 include KDSS[4], a system providing key-to-disk emulation under RSTS/E. For efficiency and sharability among numerous terminals, a run-time system is a natural solution. The WORD-11 word processing system Editor[5], is in the same vein; multiple users on a small machine necessitates minimum size sharable code in order to effectively utilize physical memory and user memory given the RSTS/E 32KW. Compilers and interpreters are also candidates for MACRO-11 and sharable code given their usage.

While MACRO-11 has distinct advantages in these and other instances it is not the answer for number crunching. Unless you can identify very, very small program segments that are very frequently executed and you want to reinvent multi-precision arithmetic in assembly language, a better solution is a language such as FORTRAN-IV-PLUS (now available under RSTS/E) which is an excellent and efficient compiler for number-crunching duties.

One of the by-products of using MACRO-11 is that is is available on all PDP-11 systems. Other languages offer some portability across operating systems but there can also be surprises, and those assuming BASIC-PLUS-2 worked the same on RSTS/E, RSX, and VAX have discovered. Certainly in MACRO-11 you lose the independent system calls and I/O offered by a higher level language. However, since Digital seems to be aiming at making RSX the defacto standard with such vehicles as the RSTS/E emulator (more fully described in the Version 7 System Directives Manual) and the VAX Application Migration Executive (described in the VAX/VMS documentation), the portability problem with MACRO-11 programs is solvable.

Another advantage of MACRO-11 is in product growth. You may develop an application in BASIC-PLUS, perhaps convert it to BASIC-PLUS-2, and then at a later time rewrite part of the application in MACRO-11. In developing a word processing product for PDP-11 operating systems, my company has taken just this route. Our first version included only the word processing Editor in MACRO-11 as a sharable run-time system. The second release converted the spooling/printing subsystem to MACRO-11. Our most recent release includes a converted list processing system, now also in MACRO-11. In both conversions we first developed the base level product in BASIC-PLUS and then having reached the size limit of 16KW and experiencing performance problems, we re-wrote the applications in MACRO-11 to improve performance, gain room to add additional features, and provide sharable run-time systems.

In converting programs from BASIC-PLUS to MACRO-11, in writing complex MACRO-11 run-time systems that includes thousands of lines of code, and in using and evaluating languages such as BASIC-PLUS-2 and FORTRAN, we have been convinced that there is no match for MACRO-11 in size and speed efficiency. Fortunately with experience and through the development of programming tools to aid in programming in MACRO-11, we do not believe the cost of assembly-language programming outweighs the benefits.

To offer an example of the comparative advantage of MACRO-11, we chose a relatively simple program to write in both BASIC-PLUS and MACRO-11. The problem was to take several strings (totaling some 376 bytes), concentrate them together and sort the individual characters in order using a bubble sort. From start to finish the programming and debugging took approximately three hours.

Each program calculates and prints the CPU time in seconds for the actual bubble sort excluding the input and output of the strings. A sample run of the two programs is shown below:

```
Ready

RUN BUBBLE.BAS

      ',,--...../llAAAABCCEFILMMOPRRSSSSTTUYaaaaaaaaaaaaaaaaabbbbbbbcc
ccddddddddddeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeffffffgggghhhhhhhhhhhhhhhi
iiiiiiiiiiiijkkllllmmmmnnnnnnnnnnnnnnnnnnnnnnnnnnooooooooooooooooooooooopp
ppppqrrrrrrrrrrrrrrrrssssssssssssssssttttttttttttttttttttttuuuuuuuuuuvvvv
wwwwxyyyyyyyyyzz

Consumed CPU time = 370.9


Ready

RUN BUBBLE.SAV

      ',,--...../llAAAABCCEFILMMOPRRSSSSTTUYaaaaaaaaaaaaaaaaabbbbbbbcc
ccddddddddddeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeffffffgggghhhhhhhhhhhhhhhi
iiiiiiiiiiiijkkllllmmmmnnnnnnnnnnnnnnnnnnnnnnnnnnooooooooooooooooooooooopp
ppppqrrrrrrrrrrrrrrrrssssssssssssssssttttttttttttttttttttttuuuuuuuuuuvvvv
wwwwxyyyyyyyyyzz

Consumed CPU time = .8

Ready
```

As you can see there is a definite difference in CPU time. This BASIC-PLUS program, however, does not really process the character string in the most efficient way possible. it uses the MID statement to compare individual characters of the string. We rewrote the BASIC-PLUS program to use a buffer, fielding, and the LSET statement and received somewhat better results:

```
RUN BUBBL2.BAS

      ',,--...../llAAAABCCEFILMMOPRRSSSSTTUYaaaaaaaaaaaaaaaaabbbbbbbcc
ccddddddddddeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeffffffgggghhhhhhhhhhhhhhhi
iiiiiiiiiiiijkkllllmmmmnnnnnnnnnnnnnnnnnnnnnnnnnnooooooooooooooooooooooopp
ppppqrrrrrrrrrrrrrrrrssssssssssssssssttttttttttttttttttttttuuuuuuuuuuvvvv
wwwwxyyyyyyyyyzz

Consumed CPU time = 64.1


Ready
```

The results from the various other runs are shown in the table below:

Program	CPU Time (Seconds)	--- Program Size Information (KW) --- Code	Data	Total + RTS Size
BASIC-PLUS (MIDS)	370.9	.47	2.08	4.0 + 16
BASIC-PLUS (LSETS)	64.1	.47	1.30	3.0 + 16
BASIC-PLUS-2 (LSET)	49.7	??	??	3.0 + 16
MACRO-11	0.8	.21	1.10	2.0 + 4

Yes MACRO-11 is faster. In this instance from 50 to 450 times faster than BASIC-PLUS and BASIC-PLUS-2.
The original BASIC-PLUS program is shown below:

```
1000    B$ =""
        ! String to be used as buffer

1010    !
        ! Read the text data to be sorted into the buffer.
        !

1020    CL% = 0%

1030    READ A$
        \ GOTO 2000 IF LEN(A$) = 0%
        \ B$=B$+A$
        \ CL% = CL% + LEN(A$)
        \ GOTO 1030
        ! Read in the data and insert it into the buffer.
        ! Terminate when a null string is read.

2000    !
        ! Record the CPU time before starting the sort.
        !

2010    ST.TIME = TIME(1)

3000    !
        ! The bubble sort.
        !

3010    CHA% = 0%
        ! Change flag to false.

3020    FOR I% = 0% TO CL% - 2%

3030            IF MID(B$,I%+1%,1%) > MID(B$,I%+2%,1%) THEN
                        B$=LEFT(B$,I%)+MID(B$,I%+2%,1%)+
                                MID(B$,I%+1%,1%)+RIGHT(B$,I%+3%)
                        \ CHA% = -1%
        ! If the two bytes are in the wrong order, then switch them and
        ! set the change flag.

3040    NEXT I%

3050    IF CHA% THEN GOTO 3010
        ! If anything changed, do it all again.

4000    !
        ! Record the CPU time afterwards.
        !
```

```
4010    END.TIME = TIME(1%)

5000    !
        ! Print out the sorted string and the consumed CPU time.
        !

5010     PRINT B$
5020    PRINT
        \ PRINT 'Consumed CPU time ='; (END.TIME - ST.TIME) / 10.

6000    !
        ! The data to be sorted.
        !

6010    DATA    "Four score and seven years ago, our forefathers set ",
                "foot upon this land.  ",
                "The quick brown fox jumped over the lazy dog.  ",
                "RSTS/E is the best operating system around.  ",
                "A penny saved is a penny earned.  ",
                "A bird in the hand is worth two in the bush.  ",
                "Many things may be done more effeciently in MACRO-11 ",
                "than in BASIC-PLUS.  ",
                "You don't have to be crazy to work here, but it helps.  "

6990    DATA    ""

32767   END
```

The version using LSETS and fielding required the following lines to be changed:

```
1000    OPEN 'NL:' AS FILE 5%, RECORD SIZE 380%
        ! This buffer will be used to hold the data during the sort.

1020    CL% = 0%

1030    READ A$
        \ GOTO 2000 IF LEN(A$) = 0%
        \ FIELD #5%, CL% AS X$, LEN(A$) AS X$
        \ LSET X$ = A$
        \ CL% = CL% + LEN(A$)
        \ GOTO 1030
        ! Read in the data and insert it into the buffer.
        ! Terminate when a null string is read.

3020    FOR I% = 0% TO CL% - 2%
        \       FIELD #5%, I% AS X$, 1% AS C1$, 1% AS C2$
        ! Field in for the current two characters.

3030            IF C1$ > C2$ THEN
                          SW$ = C1$ + ""
                        \ LSET C1$ = C2$
                        \ LSET C2$ = SW$
                        \ CHA% = -1%
        ! If the two bytes are in the wrong order, then switch them and
        ! set the change flag.

5010    FIELD #5%, CL% AS A$
        \ PRINT A$
```

Finally, here is the MACRO-11 program.

```
        .TITLE  Bubble sort example under MACRO-11.
        .ENABL  AMA,LC

        R0=%0                           ; Name the registers.
        R1=%1
        R2=%2
        R3=%3
        R4=%4
        R5=%5
        SP=%6
        PC=%7

        .MCALL  .TTYOUT                 ; Get this macro from the RT-11 macro
                                        ; library.

        .MACRO  .TIME                   ; Define this RSTS monitor call
        EMT     377                     ; This causes the call to go directly
                                        ; to RSTS.
        EMT     30                      ; Get time information.
        .ENDM

        .MACRO  .EXIT                   ; Define this RSTS monitor call
        EMT     377                     ; This causes the call to go directly
                                        ; to RSTS.
        EMT     46                      ; Exit program.
        .ENDM

        XRB=442                         ; Define the RSTS "Transfer Request
                                        ; Block".

        .CSECT  BUBBLE

START::         MOV     #TOS,SP                 ; Set up the stack pointer.

        ; Move the text to be sorted into the buffer.
        ;
        MOV     #LENGTH,R0
1$:     MOVB    TEXT-1(R0),BUFFER-1(R0) ; Move a byte.
        SOB     R0,1$                   ; Loop back for the next.

        ; Record the CPU time before starting the sort.
        ;
        .TIME                           ; Call to get time information.
        MOV     XRB+10,CPUHI
        MOV     XRB+0,CPULO

        ; The bubble sort.
        ;
2$:     CLR     CHA                     ; Change flag to false.

        CLR     R0

3$:                                     ; Top of the loop.
        CMPB    BUFFER(R0),BUFFER+1(R0) ; Compare two bytes.
```

```
        BLOS    4$

        MOVB    BUFFER(R0),R1           ; Wrong order, swap the two bytes.
        MOVB    BUFFER+1(R0),BUFFER(R0)
        MOVB    R1,BUFFER+1(R0)
        MOV     #-1,CHA                 ; And set the change flag.

4$:     INC     R0                      ; Advance the loop counter.
        CMP     R0,#LENGTH-2
        BLOS    3$                      ; And maybe loop back to the top.

        TST     CHA                     ; Did anything change?
        BNE     2$                      ; Yes- do it all again.

        ; Record the cpu time afterwords.
        ;
        .TIME                           ; Call to get time information.
        MOV     XRB+10,ENDHI
        MOV     XRB+0,ENDLO

        ; Print out the sorted string.
        ;
        CLR     R1

5$:                                     ; Top of the loop.
        .TTYOUT BUFFER(R1)              ; Note: this is a monitor call to
                                        ; the RT11 emulator.
        INC     R1
        CMP     R1,#LENGTH-1
        BLOS    5$                      ; Back for another character.

        ; Print out the consumed cpu time.
        ;
        CLR     R1                      ; First the text.
6$:                                     ; Top of the loop.
        .TTYOUT MESSAG(R1)              ; Note: this is a monitor call to
                                        ; the RT11 emulator.
        INC     R1
        CMP     R1,#MESLEN-1
        BLOS    6$                      ; Back for another character.

        ; Convert the CPU time to a string of ASCII digits.
        ;
        SUB     CPULO,ENDLO             ; Get the difference.
        SBC     ENDHI
        SUB     CPUHI,ENDHI

        MOV     ENDLO,R1

        MOV     #BUFFER+20,R3           ; Output pointer.

7$:     CLR     R0
        DIV     #10.,R0                 ; Divide out a digit.
        ADD     #'0,R1                  ; Make the remainder an ASCII digit.
        MOVB    R1,-(R3)                ; And put it in the buffer.

        MOV     R0,R1
        BNE     7$                      ; If any digits left, loop back.
```

```
        MOVB    BUFFER+17,BUFFER+20     ; Insert the decimal point.

        MOVB    #'.,BUFFER+17

        ; And print the digits.
        ;
8$:     .TTYOUT (R3)+                   ; Output the digit.
        CMP     R3,#BUFFER+20
        BLOS    8$                      ; Loop back for another.

        .TTYOUT #15                     ; Carriage-return, line-feed.
        .TTYOUT #12

        ; Exit the program.
        ;
        EMT     377                     ; This is a RSTS monitor call.
        .EXIT

        ; Text to be sorted.
        ;
TEXT::  .ASCII  "Four score and seven years ago, our forefathers set "
        .ASCII  "foot upon this land.  "
        .ASCII  "The quick brown fox jumped over the lazy dog.  "
        .ASCII  "RSTS/E is the best operating system around.  "
        .ASCII  "A penny saved is a penny earned.  "
        .ASCII  "A bird in the hand is worth two in the bush.  "
        .ASCII  "Many things may be done more effeciently in MACRO-11 "
        .ASCII  "than in BASIC-PLUS.  "
        .ASCII  "You don't have to be crazy to work here, but it helps.  "
LENGTH=.-TEXT

        ; Constant data.
        ;
MESSAG::.ASCII <15><12><15><12>"Consumed CPU time = "
MESLEN=.-MESSAG

        ; Variables and buffers.
        ;
BUFFER::.BLKB   512.                    ; Working buffer for the sort.
        .EVEN                           ; Make sure we are at an even addr.
CPUHI::         .BLKW
CPULO::         .BLKW
ENDHI::         .BLKW
ENDLO::         .BLKW
CHA::   .BLKW

        ; The stack.
        ;
        .BLKW   100.
TOS::   .BLKW

        .END    START
```

This MACRO-11 program utilizes the RT11 run-time system. In particular the TTYOUT macro is used to output a character to the terminal. The other common reference used in this program is the EMT. This is used to access a RSTS/E Monitor Directive (a monitor call). A more detailed discussion of the EMT instruction appears in the various PDP-11 Processor Handbooks while the Monitor Directives are described in the Version 7 System Directives Manual.

MACRO-11 is an alternative. Before using it consider other alternatives, consider the problems, and consider the costs. However, MACRO-11 can make some tasks which would be unacceptable in BASIC-PLUS acceptable. An example is the

dynamic priority allocation program (DYNPRI) which is used in various forms at many RSTS/E sites. Using significant CPU time in BASIC-PLUS and thereby using system resources it is trying to better allocate, the program is somewhat ineffective. The same program when rewritten in MACRO-11 can do more and use almost no appreciable CPU time even when running at very frequent intervals.

In your environment you may find parts of software systems that could similarly benefit from the advantages of MACRO-11. In most situations there is a heavily utilized software system, some part of which if carefully converted to MACRO-11 would improve the performance of the entire application or system. You should consider MACRO-11, but consider it with care.

References

[1] RSTS/E Version 7 System Directives Manual, May, 1979.

[2] "Performance Evaluation", RSTS-11 SIG Newsletter, May, 1978, Vol. 5, No. 3. This paper also has appeared in the CANADIAN DECUS Symposium Proceedings.

[3] Resident Libraries are a new feature of Version 7 of RSTS/E and provide the capability to have memory resident regions of sharable code or data that can be mapped by individual jobs into their own memory space.

[4] KDSS is a product of Evans, Griffith and Hart, Lexington, MA. It is discussed briefly in a paper presented at the Fall, 1978 DECUS Conference.

[5] WORD-11 is a product of Data Processing Design, Inc.

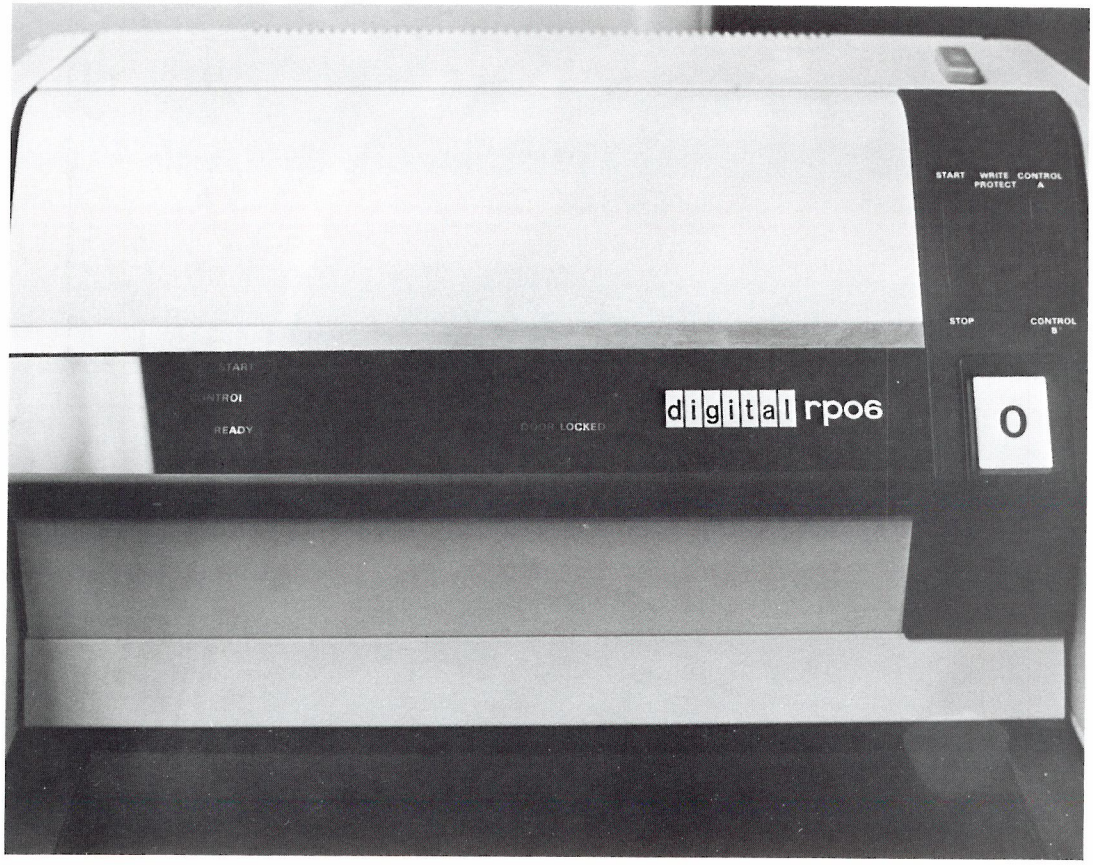
Acknowledgements

I'd like to acknowledge the help of Ted Marshall who helped write the sample programs presented in this article.

The Author

Richard Marino is Vice President of Data Processing Design, Inc., a systems and software house in Placentia, California. DPD specializes in developing high technology software for RSTS/E and other PDP-11 operating systems. Richard has presented several papers at DECUS conferences in both the United States and Canada and has authored several articles on software development and system performance analysis. In addition to managing software development at DPD, Richard provides system measurement and performance consulting to RSTS/E sites.


digital DECWRITER IV
TRANSNET CORP.
2005 ROUTE 22 UNION, NEW JERSEY 07083
SET-UP
ESC
TAB
CTRL
SHIFT
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
DELETE
RETURN


START
WRITE PROTECT
CONTROL A
STOP
CONTROL B
digital rpo6
0
READY
DOOR LOCKED

Our Wildest Card Yet

A programmable 16-line multiplexer that beats everything in its class*

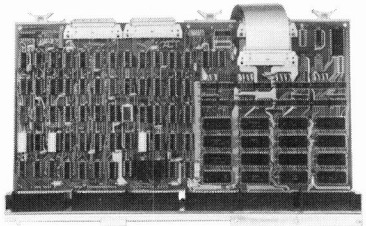
PDP-11 users, we have another winner for you. This time it's DMAX/16™, our new programmable multiplexer for connecting your PDP-11 to 16 asynchronous serial communications lines. DMAX/16 makes the most of the 11's DMA capabilities to establish computer overhead at a level far below that of competitive units like the DJ11 and DZ11. It also offers software compatibility with the DH11... in one-fourth the space!

Now, for the first time, you don't need an expansion box or special back planes. DMAX/16 consists of two hex boards which install easily into standard SPC slots and connect to the current loop or EIA/RS-232 panel by separate flat-ribbon cable. As many as 16 units can be placed on a single PDP-11 for a total of up to 256 lines. A DMUX/16™ option allows modem control for 16 channels.

DMAX/16 provides complete program control of the lines, each of which operates with several individually programmable parameters, such as character length and number of stop bits. Parity generation and detection are odd, even or none. The operating mode is half duplex or full duplex. Fifteen software programmable baud rates: 0 to 9600 baud — plus 19,200 baud – and an external baud rate. Breaks may be generated or detected on each line and the unit can echo received characters without software intervention.

Play the wild card now. You'll get top performance and a competitive price advantage of at least $1000 along with delivery from stock as usual.

Write for details and find out why we consider ourselves the leader among manufacturers of DEC enhancements.
Able Computer Technology, Incorporated,
1751 Langley Avenue, Irvine, California 92714.
(714) 979-7030. TWX 910-595-1729

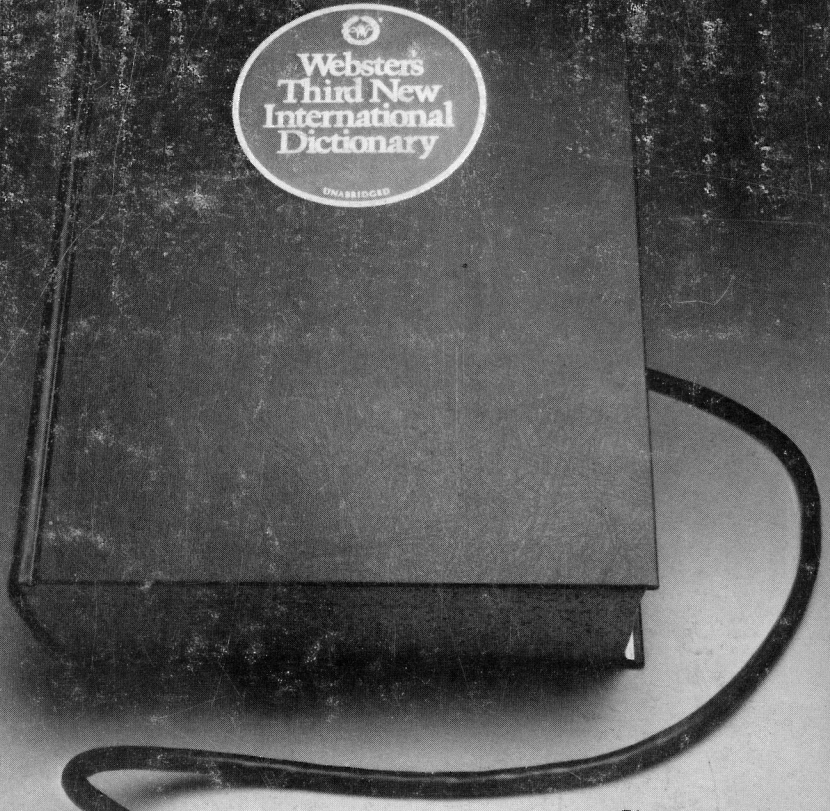


Able, the computer experts

DEC and PDP-11 are registered trademarks of Digital Equipment Corporation.

You will save half your bandwidth or run at twice the speed! Able does it again!

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.