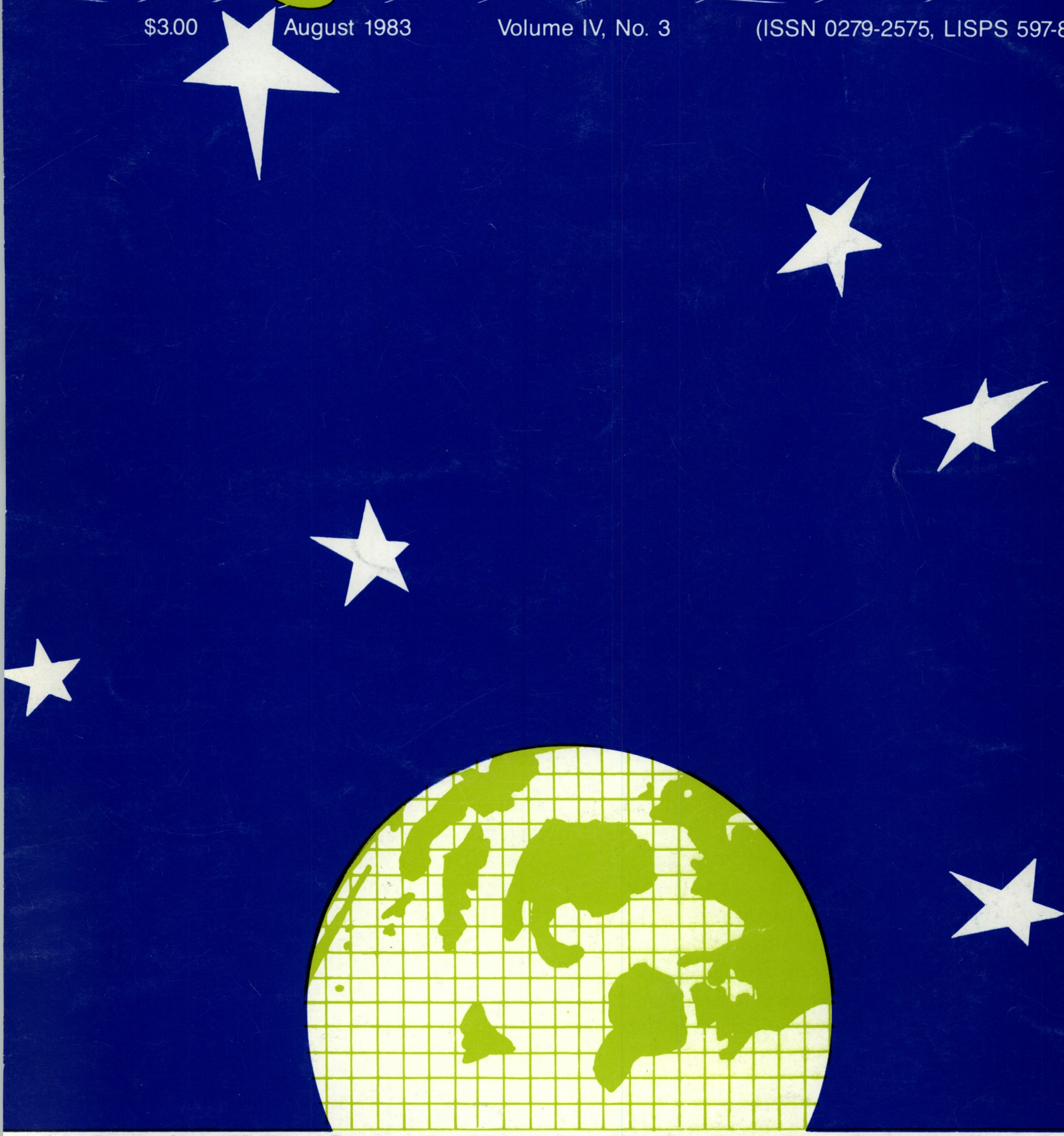
IFELINES™

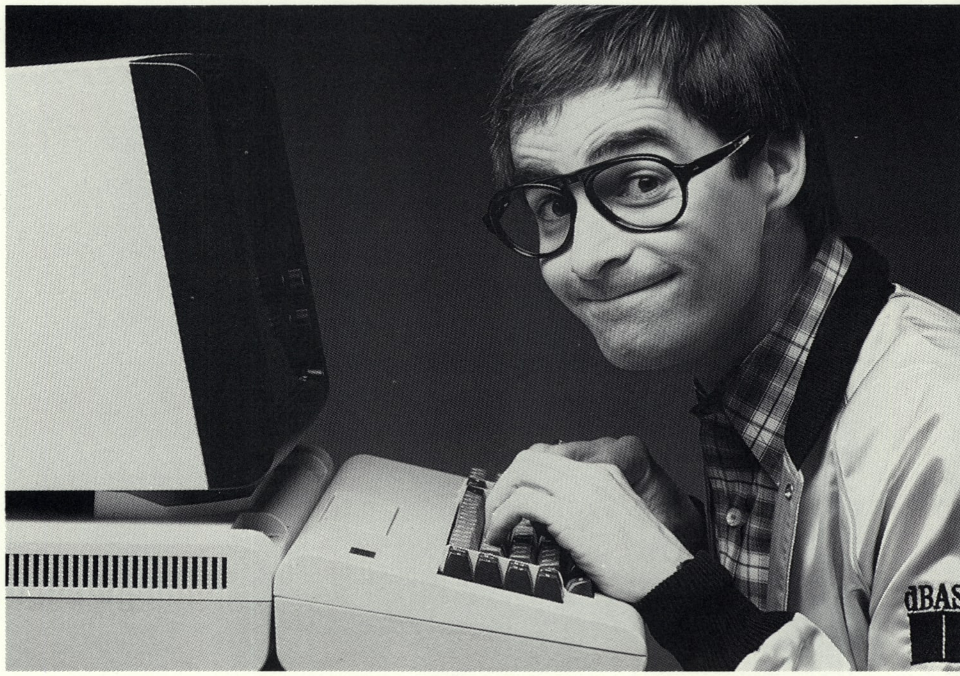
The Software Magazine™

$3.00 August 1983 Volume IV, No. 3 (ISSN 0279-2575, LISPS 597-830)



PL/I FROM THE TOP DOWN
IMPROVE YOUR MATE—PMATE

All you dBASE II™ hotshots are about to get what you deserve.



You've written all those slick dBASE II programs.

Business and personal programs. Scientific and educational applications. Packages for just about every conceivable information handling need.

And everybody who sees them loves them because they're so powerful, friendly and easy to use.

But that's just not good enough.

Uh-uh.

Because now you can get the gold and the glory that you really deserve.

Here's how.

We've just released our dBASE II RunTime™ application development module.

And it can turn you into an instant software publisher.

The RunTime module condenses and encodes your source files, protecting your special insights and techniques, so you can sell your code without giving the show away.

RunTime also protects your margins and improves your price position in the marketplace. If your client has dBASE II, all he needs is your encoded application. If not, all you need to install your application is the much less expensive RunTime module.

We'll tell the world.

With your license for the dBASE II RunTime module, we provide labels that identify your program as a dBASE II application, and you get the benefit of all the dBASE II marketing efforts.

We'll also provide additional "how to" information to get you off and running as a software publisher sooner.

And we'll make your products part of our Marketing Referral Service. Besides putting you on our referral hotline, we'll publish your program descriptions and contact information in *dBASE II Applied*, a directory now in computer stores world-wide.

Go for it.

But we can't do any of this until we hear from you.

For details, write RunTime Applications Development, Ashton-Tate, 10150 West Jefferson Boulevard, Culver City, CA 90230.

Or better yet, just call (213) 204-5570. And get what you deserve today.

dBASE™ II

ASHTON·TATE

Lifelines

The Software Magazine

Editorial

2 Bountiful Books And Bytes — Edward H. Currie

Features

3 Improve Your MATE — Steven Fisher

11 PL/I From The Top Down Chapter One—"Getting It Up" — Bruce H. Hunter

19 TXT.ASM DB Format — Thomas Hill

26 Hardware As Software: The Hayes Smartmodem — Davis A. Foulger

27 What To Do At The A>, Which Your Dealer Never Told You — Al Bloch

Software Notes

33 Macro Of The Month — Todd Katz

Product Status Reports

18 New Versions for CB-80 Compiler and Access Manager — Robert P. VanNatta

35 New Products

35 New Versions

Miscellaneous

32 Users Group Corner

Lifelines (ISSN 0279-2575,USPS 597-830) is published monthly at a subscription price of $24 for twelve issues, when destined for the U.S. Canada, or Mexico, $50 when destined for any other country. Second-class postage paid at Smithtown, New York, and other locations. POSTMASTER, please send changes of address to Lifelines Publishing Corporation, 1651 Third Avenue, New York, NY 10028.

Editorial

Bountiful Books And Bytes

by Edward H. Currie

Computer books as well as computer software are in abundance, but much of what might pass for important works is, in fact, more a result of greed than altruism. There are, of course, notable exceptions in each category and this column has made an effort to point out some of the good, the bad and the ugly.

Recently several excellent texts have appeared which are noteworthy. One such book is that of Lance A. Leventhal and Wink Saville entitled "8080/8085 Assembly Language Subroutines" which is published by Osborne/McGraw Hill (1983). This book was designed by the authors to serve as both a reference text and a source for the assembly language programmer. Particular emphasis has been placed upon code conversions, array manipulations, shifting functions, string manipulations, sorting, searching, bit manipulation and arithmetic subroutines. The text is beautifully designed with many, many examples complete with source code. The routines included can actually be used to save hours of tedious development time. Whether you program in assembly for fun or profit check this one out. You'll be impressed!

Mitchell Waite has made yet another important contribution with the latest release of a book by Waite and Lafore. You may recall that the CP/M Primer was reviewed in an earlier editorial. This latest contribution, entitled "Soul of CP/M" (can we expect "Return of CP/M," "CP/M II," "Son of CP/M," etc. ???), is subtitled "How to Use the Hidden Power of Your CP/M System." This is an excellent treatment for those of you who are anxious to learn assembly language and how to communicate with the CP/M operating system environment via assembly language. The book begins with defining program transportability, system calls, and CP/M's Golden Rule, "A chicken sandwich in one city is a chicken sandwich in all cities." It's left as an exercise for the reader of this column to determine the correct interpretation of the latter. One of the nicest features is that the book is designed to be used next to your CP/M system for true computer- aided instruction. One particularly important treatment is the interfacing of assembly language routines to BASIC. Waite and Lafore have made a valuable addition to microcomputer literature. Why not add this to your library? It's published by Howard W. Sams & Co., Inc.

Que Corporation, 7960 Castleway Drive, Indianapolis, Indiana 46250, (317) 842-7162 has just published the second edition of "IBM-PC Expansion & Software Guide." Three indexes are provided — Product Name, Vendor and Advertiser — to enable you to quickly find a particular product. In addition the book is divided into sections entitled Hardware, Software, Periodicals, Books and Directories, Supplies and Services, Future Products and IBM-PC Dealers. Take a look at this book if you are an IBM-PC owner or just enjoy reading about the wealth of products which exist for this machine.

If you are an aspiring FORTH programmer, or would just enjoy looking at a text on FORTH, get a copy of Leo Brodie's "Starting Forth" published by Prentice-Hall. This is undoubtedly the finest book available on FORTH and sets new standards for tutorial texts. Richly illustrated with examples and entertaining cartoon depictions of important concepts, both the professional and beginning FORTH programmer will find this an important addition to their library.

This text is designed to be used in conjunction with your favorite machine and FORTH implementation but can be used stand alone if desired. Once you start reading you'll find it difficult to put it down, and you frequently find yourself amused by the method of presentation. So venture FORTH and check it out!

If you are one of the many interested in learning IBM-PC BASIC take a look at "Learning IBM BASIC for the Personal Computer." This text offers Chapters entitled "Getting Started," "Speak To Me Oh Great Computer," "Strings," "Variable Precision Math," "Display Formatting," "Arrays," "Sound," Miscellaneous and "Program Control." Specific examples are presented for each BASIC function for the reader to enter and experiment with on his computer. This is a fine book for the beginning programmer but of little or no interest to the advanced BASIC programmer. This book is published by Compusoft Publishing of San Diego.

Recently, I connected my IBM-PC to my Godbout system in the mistaken belief that a sixteen-bit machine relegated to the status of expensive terminal could be used to communicate on occasion with my 8085/8088 machine. Sadly, the IBM-PC was unable to keep up with the 8085/8088 even at 9600 baud. Oh well, back to the drawing boards.

Mouse Systems Optical Mouse is an interesting device and offers some interesting capability in a variety of contexts. Microsoft has announced their 'mouse' and is currently offering products which support it. Mice are Nice but as yet are a long way from the Cat's Meow.

Virtual disks are increasing in popularity for both eight- and sixteen-bit machines as memory continues to decline in price. A number of suppliers offers 256K boards for S-100 machines and most IBM-PC memory suppliers offer virtual disk software with their products. The user should be aware, however, that while literal disk accesses are replaced by virtual disk accesses at some fifty fold speed increase, there is an increased risk of degradation in the event of memory chip or power failure. If you are using a virtual disk be sure to do frequent "SAVES" to the disk to protect your files. "An idea once conceived must be implemented" seems to be the Golden Rule of all of computerdom so watch out!

It looks as though the new Hayes Smart Modem (300/1200 baud) is causing a revolution in the world of **(continued on page 36)**

Feature

Improve Your MATE

by Steven Fisher

PMATE is an excellent text editor for working with program source code. However, we can enhance it with these changes: automatic use of the proper console and printer routines for MP/M and CP/M Plus; fully implementing instant commands with WordStar compatibility; and computing some of the installation controls. Figure 2 is an 8080 assembly listing of the version 3.21 PMATE customization area with these changes implemented.

Automatic interface

The original MATE (Michael Aaronson Text Editor) was written for an audience of CP/M-80 1.4 users. In the "old" days, programs bypassed the Basic Disk Operating System (BDOS) and went to the Basic Input Output System (BIOS) to use certain control characters or to avoid echoing keyboard inputs. This shortcut worked until Digital Research implemented multi-user and multi-task operating systems (MP/M-80 and CP/Net-80). Then programs that skipped the interrupt-management within BDOS would either exclude other tasks or "lock up" the system entirely.

How, then, were programs to get raw data from the console? With the Direct Console I/O Function (6), first provided in MP/M-80 version 1 and CP/M-80 version 2. We can write Direct I/O routines to mimic the BIOS logic for Console Input Status, Console Character Input, and Console Character Output. Such mimicry avoids extensive program changes. In PMATE, only the customization area needs new code.

In fact, we can write code that determines whether to use the BIOS or the BDOS. Such version sensitivity relies upon the obsolete Lift Drive Head Function (12), which was reassigned as the Return Version Number Function. CP/M-80 1.3 or 1.4 systems ignore the function.

Software authors need to test their wares on different computer types and configurations. Consultants often work in a variety of information environments, and want to use their own tools whenever possible. It is hard to look good when every keystroke must first be researched.

When our application programs adapt to different operating environments, we are freed from keeping an assortment of special versions. Maintaining multiple sets of a program is many times more work than keeping one version going. As our collection increases in size and complexity, the probability of using any one variation decreases. Thus, more (effort) is less (benefit).

We put version-sensitivity into the user initialization routine (UINIT) by inserting seven lines before the old BIOS linkage and 17 lines after it (USERAW). The BIOS routines are mimicked by three subroutines (RAWSTS, RAWIN, RAWOUT) and a single-character buffer (RAWCHR). When the BDOS logic is used, the List Output Status routine in the BIOS is also linked.

Instant commands compatible with WordStar

The *PMATE User Manual and Interface Guide,* Chapter 7, describes an Instant Command Table containing 59 entries. Much of the power designed into the instant commands was unavailable with the customization area furnished with PMATE. Only 36 of the commands were implemented, and not the most powerful ones.

By adding 31 bytes, we have defined all of the instant commands and implemented 42. Two inactive functions are "delete previous word" and "delete from cursor to start of line." The other 15 are confusing variations of implemented commands.

Reducing the instant command size (ICSIZ) from three to two characters saved 36 bytes while still allowing WordStar compatiblity. The command table (KEYTB) was put into numerical sequence to aid in finding entries and making changes. Although PMATE searches the command table sequentially, the slowdown caused by not having the most-frequently-used commands first is not discernable.

MicroPro's *WordStar* has become the dominant word processing package for microcomputers using Digital Research's CP/M-80 operating system. Just as PMATE offers text manipulation and command structures suitable for program authors, WordStar provides output formatting and add-on features important to document writers. Each of these products is appropriate for similar, but different, purposes.

People lose productivity when they switch between software tools with essentially the same function and radically different characteristics. Either they stop thinking about *what* they are doing long enough to remember *how* to do it, or they enter inappropriate command sequences. The instant command to move the cursor up a line in PMATE will delete the line in WordStar. Many an anguished cry is heard when tired programmers forget which editor they are using.

Michael Aaronson considered personalized keyboard commands when he designed his text editor. The initial $60 release of MATE in 1979 included information for redefining the Instant Command Table. Version 3.21 PMATE continues the tradition, at least in the 8-bit release. MicroPro requires that we pay $150 for Customization Notes before we can bend WordStar to our will. Therefore, PMATE learned to use WordStar commands and became WSMATE.

(continued on next page)

The command-abort character (ABRT) was changed to WordStar's control-U. The Overtype Mode inserted RETURN and TAB characters (ICRLF). The less-than symbol (<) began to indicate end-of-paragraph (CRCHR). Tilde (~) became the instruction to switch case during command line entry (SHFCHR). The instant command table (KEYTB) was redefined according to Figure 1.

Computing installation controls

PMATE uses data areas, or parameters, to control its operation. This is as it should be. However, some of the control values can be derived from other parameters. When a program can supply data, we shouldn't have to.

For instance, the size of the screen's text area (TDPSZ) is always three lines less than the display size (DPSZ). If our terminal provides insert-line (VIDIL) and delete-line (VIDDL) hardware support, we certainly want to use it for as many lines as possible (SCRLCT). If our terminal lacks these features, the parameter is ignored and is therefore harmless.

To keep screen redrawing to a minimum, we want the cursor to wander a half-screen from the center line in the text display area (WANDER). This lets us move over the entire text area without a screen refresh.

Instant commands ought to move not quite an entire text display area. When moving down through text a screen at a time, the last line in this screen is the first line of the next. This provides both continuity and efficiency. Therefore, we subtract 2 from the text size (TDPSZ) to derive the scroll size (SCRLNS).

When text extends beyond the screen, the display performs a lateral shift. That is, when we move the text cursor past the last column (defined by CHRLN), the text "window" slides to the right. The width of this shift, in characters, is the shift count (SHFTCT). The first version of MATE offered a shift count equal to half the width of the screen as a good compromise between continuity and speed. That's the value used here.

We make the text file page separator character (PAGSEP) zero. This prevents automatic insertion of the separator character into our text files. It also causes the program to count lines to determine when a page has been read or written, rather than scanning for the separator character.

Defining the text file page size (PAGSZ) larger than the available computer memory brings two benefits. The program automatically loads as much of the file as possible into memory, relying on the disk scrolling to manage the remainder. Global searches look at the entire file; they no longer terminate when reaching the end of the memory-resident portion of the file.

Such a deal

Since the idea behind all these improvements is to make PMATE easier to use, it follows that we can avoid retyping the 500 lines of 8080 assembler code listed in Figure 2. Controlled Information Environments will ship a diskette containing the WSMATE Instant Command summary and the unassembled WSMATE Customization Area via U.S. Mail upon receipt of $25. California residents must add $1.25 sales tax or include a duly signed resale card.

Orders paid by cash, Cashier's Check, or Money Order will be processed without delay. Orders paid by check will be held for four weeks to allow the check to clear. Checks must be drawn on a U.S. bank and payable in U.S. dollars.

Available diskette formats are 8-inch, soft-sectored single-sided, single-density CP/M standard (STD8); 5.25-inch Lifeboat NorthStar 1.4 CP/M 48-TPI 10-sectored single-sided single-density (N*SSSD); 5.25-inch Lifeboat NorthStar 2.2 CP/M 48-TPI 10-sectored single-sided double-density (N*SSDD); and 5.25-inch Epic Episode 96-TPI soft-sectored double-sided quad-density (EPIC96). Diskettes will be 8-inch unless otherwise requested.

Send orders to Controlled Information Environments, PO Box 457, La Mesa, CA USA 92041. Whether copied or rekeyed, these changes will improve your MATE.

Figure 1 — WSMATE Instant Commands

This table lists all the instant commands possible in PMATE version 3.21. Those that have been implemented mimic the WordStar command structure. Inactive entries have no command sequence indicated.

Command	Number	Action
↑↑	128	Move to beginning/end of in-memory text
	129	Move to end of in-memory text
↑H	130	Move cursor backward one text-character
↑A	131	Move cursor backward one text-word
↑J	132	Move cursor forward one text-character
↑F	133	Move cursor forward one text-word
	134	Move cursor up one text-line
	135	Move cursor up multiple text-lines
	136	Move cursor down one text-line
	137	Move cursor down multiple text-lines
↑G	138	Delete forward one text-character
↑Y	139	Delete forward from cursor to end of line
↑O ↑V	140	Begin Insert Mode
↑O ↑O	141	Edit Command Buffer
↑U	142	Abort
↑O ↑A	143	Shift case
↑B	144	Reformat and redraw display
↑K ↑B	145	Tag current cursor position
↑T	146	Delete forward one text-word
	147	Delete backward one text-word
↑P	148	Recover previous erasure/deletion
↑O ↑C	149	Enter Command Mode
↑O ↑R	150	Enter Overtype Mode
↑N	151	Insert New Line
↑S	152	Move cursor backward one screen position
↑D	153	Move cursor forward one screen position
	154	Move cursor up one position (mixed)
	155	Move cursor down one position (mixed)
↑K ↑V	156	Move text block into buffer
↑K ↑C	157	Get text block from buffer
↑E	158	Move cursor up one screen position
↑X	159	Move cursor down one screen position
	160	Move to beginning of in-memory text
	161	Move cursor backward one position (mixed)
	162	Move cursor forward one position (mixed)
↑Q ↑R	163	Move to beginning of file
↑Q ↑C	164	Move to end of file
	165	Move to beginning/end of file
↑\	166	Change case of one text-character forward
↑O ↑X	167	Swap two previous text-characters
↑Q ↑D	168	Move cursor to end of text-line
↑Q ↑S	169	Move cursor to beginning of text-line

↑Q ↑W	170	Move cursor up one text-screen
↑Q ↑Z	171	Move cursor down one text-screen
↑R	172	Move cursor up multiple screen positions
↑C	173	Move cursor down multiple screen positions
	174	Move cursor up multiple positions (mixed)
	175	Move cursor down multiple positions (mixed)
↑V	176	Enter Insert/Overtype Mode
	177	Delete backward to beginning of text-line
↑O ↑I	178	Set auto-indent to current column
↑K ↑K	179	Exchange tag and cursor
↑O ↑D	180	Increment auto-indent 4 columns
↑O ↑S	181	Decrement auto-indent 4 columns
↑Z	182	Scroll up
↑W	183	Scroll down
	184	Scroll backward
	185	Scroll forward
↑L	191	Repeat
↑Q 1	209	User Macro 1
↑Q 2	210	User Macro 2

Figure 2 — WSMATE Customization Area 8080 Assembly Listing

```
;               WSMATE I/O PATCH
;               05/31/83 SWF
;
; CUSTOMIZATION FOR PMATE EDITOR, VERSION 3.21
; TO USE THIS:
;               ASM WSMATE.AAA
;               DDT PMATE.COM
;               IWSMATE.HEX
;               R
;               G0000
;               SAVE 84 WSMATE.COM
;
; ALTERATION HISTORY:
; SWF = STEVEN FISHER CDP, P O BOX 457, LA MESA, CA 92041
; 10/05/82 SWF — IMMEDIATE-MODE KEYS NOW LIKE MICRO-
; PRO'S WORD-STAR
; 10/11/82 SWF — INITIALIZATION SETS DISPLAY AND SCROLL
; PARAMETERS
; 10/12/82 SWF — USES RAW-IO FOR ALL MP/M & CP/M LATER
; THAN 1.4
; 11/03/82 SWF — REDUCED ICSIZ FOR COMMAND TABLE &
; ADDED ALL DEFINED COMMANDS
; 12/29/82 SWF — INCREASED PAGSZ TO READ ENTIRE TEXT FILE
; AT START
; 01/01/83 SWF — DEFINED VIDEO TO CO-SUPPORT SOROC IQ-120
; & IMSAI VIO-C
; 05/31/83 SWF — REMOVED CONTXT FROM UINIT INITIALIZATION
FALSE   EQU   0       ; FOR CONDITIONS
TRUE    EQU   0FFH    ; FOR CONDITIONS
CTL     EQU   40H     ; FOR EQUATES
MEMMAP  EQU   FALSE   ; TRUE IF MEMORY-
                      ; MAPPED VIDEO
MACSZ   EQU   200     ; SIZE OF PERMANENT
                      ; MACRO AREA
BIOSPT  EQU   0001H   ; POINTS TO WARM
                      ; BOOT BIOS VECTOR
                      ; (SWF)
SYSTEM  EQU   0005H   ; ENTRY POINT TO
                      ; SYSTEM (SWF)
RAWIO   EQU   06H     ; SYSTEM FUNCTION
                      ; TO USE RAW IO (SWF)
VERSNO  EQU   0CH     ; SYSTEM FUNCTION
                      ; TO GET VERSION
                      ; (SWF)

        ORG   109H

                 ; JUMP VECTORS AND USER VARIABLES
UINITL:  JMP   UINIT    ; USER INITIALIZATION
UEXIT:   RET            ; USER EXIT ROUTINE
         NOP
         NOP
CI:      JMP   $—$      ; CONSOLE INPUT
                        ; VECTOR
CSTS:    JMP   $—$      ; CONSOLE STATUS
                        ; VECTOR
COUT:    JMP   $—$      ; CONSOLE OUTPUT
                        ; VECTOR
LO:      JMP   $—$      ; LIST VECTOR
LSTS:    DB    0,0,0    ; LIST STATUS VECTOR
MONTR:   RET            ; MONITOR VECTOR
         NOP
         NOP
KEYTAB:  DW    KEYTB    ; POINTER TO INSTANT
                        ; COMMAND KEY-
                        ; STROKE TABLE
ICSIZ:   DB    2        ; SIZE OF SLOT IN
                        ; TABLE (SWF)
INITMD:  DB    0        ; INITIAL MODE —
                        ; COMMAND (0),
                        ; INSERT (1),
                        ; OVERTYPE (2)
         DB    0,0      ; RESERVED
UCOM:    DW    USRCOM   ; POINTER TO USER
                        ; COMMAND EXE-
                        ; CUTED BEFORE
                        ; INPUT OF NEXT
                        ; COMMAND
                 ; DISPLAY PARAMETERS - "$—$" DERIVED
                        ; BY LOGIC IN UNIT
TDPSZ:   DB    $—$      ; HOW MANY LINES IN
                        ; TEXT SECTION OF
                        ; DISPLAY (SWF)
DPSZ:    DB    24       ; HOW MANY LINES IN
                        ; VIDEO DISPLAY
CHRLN:   DB    80       ; HOW MANY
                        ; CHARACTERS IN
                        ; EACH LINE
SHFTCT:  DB    $—$      ; SIZE OF SHIFT WHEN
                        ; CURSOR MOVES OFF
                        ; SCREEN (SWF)
SCRLCT:  DB    $—$      ; MAXIMUM LINES TO
                        ; SCROLL VIA HARD-
                        ; WARE INSERT-
                        ; AND DELETE-LINE
                        ; (SWF)
WANDER:  DB    $—$      ; HOW MANY LINES
                        ; CURSOR CAN
                        ; WANDER FROM
                        ; CENTER (SWF)
CONTXT:  DB    3        ; HOW MANY LINES
                        ; ALWAYS REDRAWN IN
                        ; FOREGROUND
NOLSTC:  DB    TRUE     ; TRUE IF LAST
                        ; CHARACTER OF
                        ; DISPLAY MUST BE
                        ; SUPPRESSED
DOWN:    DB    FALSE    ; TRUE IF DISPLAY
                        ; MUST DRAW FROM
                        ; TOP TO BOTTOM
EVRYLN:  DB    FALSE    ; TRUE TO DISPLAY
                        ; CURSOR BEFORE
                        ; DRAWING EACH LINE
BLNKCT:  DB    25       ; CURSOR BLINK
                        ; COUNT
```

(continued on next page)

```
                                ; -- 0 FOR NO BLINK
IGNRIC:     DB      3           ; IGNORE INSTANT
                                ; COMMANDS IF
                                ; DEEPER THAN THIS
                                ; IN BUFFER
MMAP:       DB      MEMMAP      ; TRUE IF MEMORY-
                                ; MAPPED DISPLAY
VRAM0:      DW      0F000H      ;BEGINNING OF
                                ; VIDEO RAM
LSPAC:      DW      80H         ;LINE SPACE — LINE
                                ; OFFSET IN VIDEO
                                ; MEMORY
DSPCUR:                         ; ROUTINE TO DISPLAY
                                ; CURSOR AT MEMORY
                                ; POINTED TO BY HL
            MOV     A,M
            ORI     80H
            MOV     M,A
            RET
            DS      16          ; 21 BYTES TOTAL
CLRCUR:                         ; ROUTINE TO CLEAR
                                ; CURSOR FROM
                                ; MEMORY POINTED
                                ; TO BY HL
            MOV     A,M
            ANI     7FH
            MOV     M,A
            RET
            DS      16          ; 21 BYTES TOTAL
DSPCHR:                         ; ROUTINE TO STORE
                                ; CHAR IN REG A IN
                                ; VIDEO MEMORY
                                ; POINTED TO BY DE
            STAX    D
            RET
            DS      19          ; 21 BYTES TOTAL
            ENDIF               ; END OF CONDITIONAL
                                ; ASSEMBLY IF MEMAP
            IF      NOT MEMMAP  ; FOR SERIAL
                                ; TERMINAL
                    ; VIDEO CODES — SEQUENCES END IN 0,
                    ; THEN NEXT BYTE IS HOW MANY
                    ; MILLISECONDS TO DELAY AFTER
                    ; SENDING SEQUENCE TO TERMINAL
VIDCLS:             ; SEQUENCE THAT CLEARS THE SCREEN
                    ; AFTER HOMING CURSOR
            DB      1BH,'*',1AH
            DB      10H,0,0
            DB      0,0,0
VIDCLL:             ; SEQUENCE THAT CLEARS FROM
                    ; CURSOR TO END OF LINE
            DB      1BH,'T',15H
            DB      0,0,0
            DB      0,0,0
            ; CURSOR ADDRESSING
VIDASC:     DB      FALSE       ; ASCII FLAG — TRUE
                                ; FOR ASCII, FALSE IF
                                ; BINARY
VIDXY:      DB      FALSE       ; XY FLAG — FALSE IF
                                ; Y (COL) FIRST, TRUE
                                ; IF X (ROW)
VIDOF1:     DB      ' '         ; OFFSET OF FIRST
                                ; COORDINATE
VIDOF2:     DB      ' '         ; OFFSET OF SECOND
                                ; COORDINATE
VIDCUL:             ; LEAD IN OF CURSOR POSITIONING
            DB      27,'=',0
            DB      0,0,0
            DB      0,0,0
VIDCUM:             ; MIDDLE OF CURSOR POSITIONING
            DB      0,0,0
            DB      0,0,0
            DB      0,0,0
VIDCUE:             ; END OF CURSOR POSITIONING
            DB      0,0,0
            DB      0,0,0
            DB      0,0,0

            ; HARDWARE INSERT- AND DELETE-LINE
            ; 15 MILLISECOND DELAY TYPICAL)
VIDIL:              ; SEQUENCE THAT INSERTS LINE —
                    ; END IN 0, THEN MSEC DELAY
            DB      0,0,0
            DB      0,0,0
            DB      0,0,0
VIDDL:              ; SEQUENCE THAT DELETES LINE —
                    ; END IN 0, THEN MSEC DELAY
            DB      0,0,0
            DB      0,0,0
            DB      0,0,0
            ENDIF               ; END OF CONDI-
                                ; TIONAL ASSEMBLY
                                ; IF NOT MEMAP
            DB      0           ; RESERVED
DELAY:      DB      100         ; DELAY TIME FOR
                                ; QD COMMAND
ABRT:       DB      'U'-CTL     ; ABORT CHARACTER
                                ; (SWF)
ICRFL:      DB      TRUE        ; TRUE IF <CR> AND
                                ; <HT> ARE INSERTED
                                ; IN OVERTYPE (SWF)
                    ; MEMORY PARAMETERS — "$-$" DERIVED
                    ; BY LOGIC IN UINIT
CORBEG:     DW      EDEND+MACSZ+1
                                ; FIRST AVAILABLE
                                ; CORE LOCATION
CORMX:      DW      $-$         ; LAST AVAILABLE
                                ; CORE LOCATION
                                ; (SWF)
GBGSZ:      DW      -1000       ; NEGATIVE
                                ; MINIMUM SIZE OF
                                ; GARBAGE AREA
TXTEND:     DW      $-$         ; LAST LOCATION
                                ; AVAILABLE FOR TEXT,
                                ; REST
                                ; IS RESERVED FOR
                                ; GARBAGE (SWF)
MACBEG:     DW      EDEND       ; FIRST AVAILABLE
                                ; LOCATION FOR
                                ; PERMANENT MACROS
MACEND:     DW      EDEND+MACSZ ; LAST AVAILABLE
                                ; LOCATION FOR
                                ; PERMANENT MACROS
                    ; THE FOLLOWING VARIABLES CAN BE SET
                    ; BY THE Q COMMANDS
UVAR0:      DW      0           ; USER DEFINABLE
                                ; VARIABLES
UVAR1:      DW      0
UVAR2:      DW      0
UVAR3:      DW      0
UVAR4:      DW      0
UVAR5:      DW      0
UVAR6:      DW      0
UVAR7:      DW      0
```

```
UVAR8:    DW     0
UVAR9:    DW     0
SHFCHR:   DB     '~'           ; UPPER OR LOWER
                               ; CASE SHIFT
                               ; CHARACTER (SWF)
CNTCHR:   DB     '↑'           ; CHARACTER
                               ; DISPLAYED FOR CONTROL
                               ; CHARACTER
PAGSZ:    DW     65000         ; LINES PER PAGE
                               ; (SWF)
PAGSEP:   DB     0             ; PAGE SEPARATOR
                               ; (SWF)
SCRLNS:   DW     $-$           ; LINES TO SCROLL
                               ; PER INSTANT
                               ; COMMAND (SWF)
BKUFL:    DB     TRUE          ; TRUE IF BACKUPS
                               ; ARE TO BE MADE
XMAX:     DB     250           ; MAXIMUM ALLOWED
                               ; X CURSOR POSITION
CRCHR:    DB     '<'           ; CHARACTER
                               ; DISPLAYED FOR
                               ; PARAGRAPH END
                               ; (SWF)
GLBLSZ:   DW     $-$           ; SIZE OF BLOCK FOR
                               ; DISK SCROLL
                               ; WRITES (SWF)
GLROOM:   DW     $-$           ; ROOM LEFT AFTER
                               ; GLOBAL DISK
                               ; OPERATIONS (SWF)
GLINSZ:   DW     1000          ; SIZE OF BLOCK
                               ; WRITTEN TO MAKE
                               ; ROOM FOR INSERT
          ORG    5239H
```

USER INITIALIZATION

```
UINIT:

; SET MEMORY-USAGE PARAMETERS
          LHLD   06H           ; POINTER TO
                               ; BEGINNING OF BDOS
          DCX    H             ; POINTS TO LAST
                               ; POSITION IN
                               ; PROGRAM AREA
          SHLD   CORMX         ; LAST AVAILABLE
                               ; CORE LOCATION
          XCHG
          LHLD   GBGSZ         ; NEGATIVE OF SIZE
                               ; ALLOWED FOR
                               ; GARBAGE AREA
          DAD    D
          SHLD   TXTEND
          LHLD   CORBEG
          LDA    TXTEND+1
          SUB    H
          RAR                  ; DIVIDE BY 2
          ANA    A             ; CLEAR CARRY
          RAR                  ; DIVIDE BY 2
                               ; AGAIN (SWF)
          ANA    A             ; CLEAR CARRY (SWF)
          RAR                  ; DIVIDE BY 2 AGAIN
          MOV    H,A
          MVI    L,0
          SHLD   GLROOM        ; LEAVE 1/8 OF THAT
                               ; ROOM FREE FOR
                               ; GLOBALS
          SHLD   GLBLSZ        ; AND USE SAME
                               ; AMOUNT FOR
                               ; SCROLL BLOCK SIZE
; SET ROW-ORIENTED DISPLAY PARAMETERS (SWF)
          LDA    DPSZ          ; GET DISPLAY SIZE
          SUI    3             ; REMOVE SIZE OF
                               ; COMMAND DISPLAY
          STA    TDPSZ         ; SET TEXT DISPLAY
                               ; SIZE
          STA    SCRLCT        ; USE INSERT/DELETE
                               ; TO SCROLL ALL
          ANI    0FEH          ; FORCE IT TO AN
                               ; EVEN NUMBER
          RAR                  ; DIVIDE BY 2
          STA    WANDER        ; CURSOR CAN MOVE
                               ; ALL OVER SCREEN
          LDA    TDPSZ         ; GET TEXT DISPLAY
                               ; SIZE
          SUI    2             ; KEEP CONTINUITY
                               ; IN A SCROLL
          STA    SCRLNS        ; A SCROLL IS
                               ; ALMOST ENTIRE
                               ; TEXT SCREEN
; SET COLUMN-ORIENTED DISPLAY
; PARAMETERS (SWF)
          LDA    CHRLN         ; GET DISPLAY WIDTH
          ANI    0FEH          ; FORCE IT TO AN
                               ; EVEN NUMBER
          RAR                  ; DIV IDE BY 2
          STA    SHFTCT        ; HOW FAR TO SHIFT
                               ; WHEN MOVING OFF
                               ; SCREEN
; LINK TO CONSOLE AND PRINTER
          LXI    D,0           ; INITIALIZE IN CASE
                               ; IT IS 1.4 CP/M (SWF)
          LXI    H,0           ; DITTO (SWF)
          MVI    C,VERSNO
          CALL   SYSTEM        ; GET VERSION
                               ; NUMBER (SWF)
          MOV    H,A           ; SEE IF HL=0 (SWF)
          ORA    L             ; 0 IF CP/M 1.4 (SWF)
          JNZ    USERAW        ; USE RAW I/O FOR
                               ; OTHER THAN CP/M
                               ; 1.4 (SWF)
          LHLD   BIOSPT        ; POINTER TO WARM
                               ; BOOT VECTOR
          LXI    D,3
          DAD    D             ; CONSOLE STATUS
                               ; VECTOR
          SHLD   CSTS+1
          DAD    D
          SHLD   CI+1          ; CONSOLE IN VECTOR
          DAD    D
          SHLD   COUT+1        ; CONSOLE OUT
                               ; VECTOR
          DAD    D
          SHLD   LO+1          ; LIST OUTPUT
                               ; VECTOR
          RET
USERAW:                        ; THIS LINKS TO RAWIO CONSOLE I/O AND
                               ; LIST STATUS CHECK (SWF)
                               ; IT WORKS FOR CP/M 2.X OR LATER,
                               ; AND MP/M 1.X OR LATER (SWF)
          LXI    H,RAWSTS
          SHLD   CSTS+1
          LXI    H,RAWIN
          SHLD   CI+1
          LXI    H,RAWOUT
```

(continued on next page)

```
        SHLD    COUT+1
        LHLD    BIOSPT          ; WARM-BOOT VECTOR
        XCHG
        LXI     H,12            ; OFFSET TO LIST
                                ; OUTPUT
        DAD     D
        SHLD    LO+1
        MVI     A,0C3H          ; PLACE JUMP
                                ; STATEMENT IN LIST
                                ; STATUS
        STA     LSTS
        LXI     H,52            ; OFFSET TO LIST
                                ; STATUS
        DAD     D
        SHLD    LSTS+1
        RET
```

RAWIO CONSOLE LOGIC

```
RAWCHR: DB      0               ; PENDING INPUT
                                ; CHARACTER (SWF)
RAWSTS:         ; CHECK FOR PENDING CONSOLE
                ; INPUT (SWF)
        LDA     RAWCHR
        ORA     A               ; STILL-PENDING
                                ; INPUT?
        JNZ     RAWSET          ; IF SO THEN SAY SO
        MVI     C,RAWIO         ; LOOK AT CONSOLE
        MVI     E,TRUE          ; GET STATUS OR
                                ; INPUT
        CALL    SYSTEM
        STA     RAWCHR          ; SAVE CHARACTER
                                ; OR STATUS
        ORA     A               ; WAS THERE
                                ; ANYTHING?
        RZ      ; ALL DONE IF NOT
RAWSET: ; SET ALL ACCUMULATOR BITS TO SHOW
        ; THERE IS INPUT
        SUB     A
        CMA
        RET
RAWIN:          ; WAIT FOR AN INPUT FROM THE
                ; CONSOLE (SWF)
        CALL    RAWSTS
        JZ      RAWIN           ; TRY AGAIN IF
                                ; NO INPUT
        LXI     H,RAWCHR
        MOV     A,M             ; GET PENDING
                                ; CHARACTER
        MVI     M,FALSE         ; CLEAR BUFFER
        RET
RAWOUT:         ; SEND A CHARACTER TO THE
                ; CONSOLE (SWF)
        MOV     E,C             ; BIOS USES C,
                                ; BDOS USES E
        MVI     C,RAWIO
        MVI     A,TRUE          ; ARE WE TRYING TO
                                ; SEND 0FFH?
        CMP     E               ; SUPPRESS 0FFH AS
                                ; IT CAUSES INPUT
        CNZ     SYSTEM          ; OTHERWISE SEND
                                ; TO CONSOLE
        RET

;
```

INITIAL COMMAND

```
USRCOM:         ; INITIALLY EXECUTED USER COMMAND
        DB      0
```

INSTANT COMMAND TABLE

```
KEYTB:          ; EACH ENTRY CONSISTS OF COMMAND
                ; NUMBER AND INPUT PATTERN
                ; SIZE OF INPUT PATTERN DEFINED BY
                ; VARIABLE ICSIZ (2) (SWF)
                ; THIS VERSION SIMILAR TO MICRO-PRO'S
                ; WORD-STAR (SWF)
        DB      0+128           ; -- MOVE TO MEMORY
                                ; BEGINNING/END
        DB      '↑'CTL
        DB      0
        DB      1+128           ; -- MOVE TO MEMORY
                                ; END
        DB      0
        DB      0
        DB      2+128           ; -- MOVE LEFT
        DB      'H'-CTL
        DB      0
        DB      3+128           ; -- MOVE LEFT ONE
                                ; WORD
        DB      'A'-CTL
        DB      0
        DB      4+128           ; -- MOVE RIGHT
        DB      'J'-CTL
        DB      0
        DB      5+128           ; -- MOVE RIGHT ONE
                                ; WORD
        DB      'F'-CTL
        DB      0
        DB      6+128           ; -- MOVE UP ONE LINE
        DB      0
        DB      0
        DB      7+128           ; -- MOVE UP MULTIPLE
                                ; LINES
        DB      0
        DB      0
        DB      8+128           ; -- MOVE DOWN ONE
                                ; LINE
        DB      0
        DB      0
        DB      9+128           ; -- MOVE DOWN
                                ; MULTIPLE LINES
        DB      0
        DB      0
        DB      10+128          ; -- DELETE
                                ; CHARACTER
        DB      'G'-CTL
        DB      0
        DB      11+128          ; -- KILL LINE
        DB      'Y'-CTL
        DB      0
        DB      12+128          ; -- GO TO INSERT
                                ; MODE
```

```
DB      'O'-CTL
DB      'V'-CTL

DB      13+128          ; -- EDIT COMMAND
DB      'O'-CTL
DB      'O'-CTL

DB      14+128          ; -- ABORT
DB      'U'-CTL
DB      0

DB      15+128          ; -- SHIFT CASE
DB      'O'-CTL
DB      'A'-CTL

DB      16+128          ; -- REFORMAT AND
                        ; REDRAW
DB      'B'-CTL
DB      0

DB      17+128          ; -- TAG
DB      'K'-CTL
DB      'B'-CTL

DB      18+128          ; -- DELETE WORD
                        ; FORWARD
DB      'T'-CTL
DB      0

DB      19+128          ; -- DELETE WORD
                        ; BACKWARD
DB      0
DB      0

DB      20+128          ; -- POP GARBAGE
                        ; STACK
DB      'P'-CTL
DB      0

DB      21+128          ; -- GO TO COMMAND
                        ; MODE
DB      'O'-CTL
DB      'C'-CTL

DB      22+128          ; -- GO TO OVERTYPE
                        ; MODE
DB      'O'-CTL
DB      'R'-CTL

DB      23+128          ; -- INSERT LINE
DB      'N'-CTL
DB      0

DB      24+128          ; -- MOVE LEFT
                        ; GEOMETRIC
DB      'S'-CTL
DB      0

DB      25+128          ; -- MOVE RIGHT
                        ; GEOMETRIC
DB      'D'-CTL
DB      0

DB      26+128          ; -- MOVE UP MIXED
DB      0
DB      0

DB      27+128          ; -- MOVE DOWN MIXED
DB      0
DB      0

DB      28+128          ; -- MOVE BLOCK
DB      'K'-CTL
DB      'V'-CTL

DB      29+128          ; -- GET BLOCK
DB      'K'-CTL
DB      'C'-CTL

DB      30+128          ; -- MOVE UP ONE LINE
                        ; GEOMETRIC
DB      'E'-CTL
DB      0

DB      31+128          ; -- MOVE DOWN ONE
                        ; LINE GEOMETRIC
DB      'X'-CTL
DB      0

DB      32+128          ; -- MOVE TO MEMORY
                        ; BEGINNING
DB      0
DB      0

DB      33+128          ; -- MOVE LEFT MIXED
DB      0
DB      0

DB      34+128          ; -- MOVE RIGHT MIXED
DB      0
DB      0

DB      35+128          ; -- MOVE TO FILE
                        ; BEGINNING
DB      'Q'-CTL
DB      'R'-CTL

DB      36+128          ; -- MOVE TO FILE END
DB      'Q'-CTL
DB      'C'-CTL

DB      37+128          ; -- MOVE TO FILE
                        ; BEGINNING/END
DB      0
DB      0

DB      38+128          ; -- CHANGE CASE OF
                        ; CHAR AT CURSOR
DB      '\'-CTL
DB      0

DB      39+128          ; -- REVERSE LAST
                        ; TWO CHARS
DB      'O'-CTL
DB      'X'-CTL

DB      40+128          ; -- MOVE TO END
                        ; OF LINE
DB      'Q'-CTL
DB      'D'-CTL

DB      41+128          ; -- MOVE TO BEGIN-
                        ; NING OF LINE
DB      'Q'-CTL
DB      'S'-CTL

DB      42+128          ; -- MOVE UP ONE
                        ; SCREENFUL
DB      'Q'-CTL
DB      'W'-CTL

DB      43+128          ; -- MOVE DOWN ONE
                        ; SCREENFUL
DB      'Q'-CTL
DB      'Z'-CTL

DB      44+128          ; -- MOVE UP MULTIPLE
                        ; LINES GEOMETRIC
DB      'R'-CTL
DB      0

DB      45+128          ; -- MOVE DOWN
                        ; MULTIPLE LINES
                        ; GEOMETRIC
DB      'C'-CTL
DB      0
```

(continued on next page)

```
DB      46+128          ; -- MOVE UP MULTIPLE
                        ; LINES MIXED
DB      0
DB      0
DB      47+128          ; -- MOVE DOWN
                        ; MULTIPLE LINES
                        ; MIXED
DB      0
DB      0
DB      48+128          ; -- TOGGLE BETWEEN
                        ; INSERT AND
                        ; OVERTYPE
DB      'V'-CTL
DB      0
DB      49+128          ; -- DELETE TO BEGIN-
                        ; NING OF LINE
DB      0
DB      0
DB      50+128          ; -- SET AUTOTAB
                        ; LEVEL TO CURRENT
                        ; POSITION
DB      'O'-CTL
DB      'I'-CTL
DB      51+128          ; -- EXCHANGE TAG
                        ; AND CURSOR
DB      'K'-CTL
DB      'K'-CTL
DB      52+128          ; -- INCREMENT
                        ; AUTOTAB LEVEL
DB      'O'-CTL
DB      'D'-CTL
DB      53+128          ; -- DECREMENT
                        ; AUTOTAB LEVEL
DB      'O'-CTL
DB      'S'-CTL
DB      54+128          ; -- SCROLL UP
DB      'Z'-CTL
DB      0
DB      55+128          ; -- SCROLL DOWN
DB      'W'-CTL
DB      0
DB      56+128          ; -- SCROLL LEFT
DB      0
DB      0
DB      57+128          ; -- SCROLL RIGHT
DB      0
DB      0
DB      63+128          ; -- REPEAT
DB      'L'-CTL
DB      0
DB      32+128+'1'      ; USER COMMAND -- .1
DB      'Q'-CTL
DB      '1'
DB      32+128+'2'      ; USER COMMAND -- .2
DB      'Q'-CTL
DB      '2'
DB      OFFH            ; END OF TABLE
; END OF EDITOR
EDEND:  DB      0
        END
```

Feature

PL/I From The Top Down
Chapter One—Getting It Up—Continued

By Bruce H. Hunter

This is a continuation of last month's (June 1983) article, all part of a continuing series of articles. The material for these articles has been taken from my book "PL/I From the Top Down" and rewritten for *Lifelines* in article form. The purpose of Chapter One is to introduce some elementary concepts of PL/I designed to give the reader enough basic knowledge to get PL/I "up and running." By "up and running" I mean learning enough to write elementary programs and be able to link and compile them at home. In most other languages this would not be difficult. PL/I is another story. It is a very sophisticated language, and getting this language "up" is a strenuous task, especially if you are relatively new to programming languages. The power and versatility of PL/I are going to astound those of you who come from teaching languages like BASIC and Pascal. Whole new programming worlds will open up for you, but it's going to take some work! So, let's get to it.

One of the advantages of writing an informal text is the casual way new material can be introduced. I like to use a spiral approach in teaching computer languages, especially complex ones like PL/I. We will cover only a little bit of each concept initially so you can get an overview of the language. As we go along we will go into more and more detail. The compiler used for the writing of this book is Digital Research's PL/I-80, a subset of PL/I Subset G.

PL/I
FROM THE TOP DOWN
By Bruce H. Hunter

CHAPTER ONE — GETTING IT UP (continued)

Edited I/O

In the first installment of this article we were just beginning to cover I/O (input/output) in PL/I. Input/output in PL/I has the advantage of being offered two ways, unformatted and formatted. I often call unformatted I/O "list I/O" because it is the I/O being used by "get list" and "put list" program statements. Unformatted or list I/O comes in and goes out in a relatively undisciplined manner. The programmer exercises very little control over where the output goes on a line. Input is nearly as uncontrollable — whatever the individual at the console types in is what the program gets. Edited or formatted I/O is quite another matter. Output can be controlled by column and line. Numeric output can be controlled for length, notation, and decimal length. Outputs can be controlled for either right or left justification. Any number of lines can be skipped and any number of parameters can be repeated. Formatted input is just as flexible. The type of data can be predetermined and the length of the input data can be easily controlled. Edited or formatted I/O is a carry-over from first generation computers, card readers and punches. This was the time when the large mainframes depended on punched cards to input data. Card fields have to be very exacting. The output fields are predetermined within the program by the programmer, and the input formats must be just as exact to read the card. It's interesting to note that in the United States card reading came to be used extensively, which resulted in our early languages becoming "card-oriented" and thus rigidly formatted. An example of a language developed in this era is FORTRAN, which indeed is a formatted language. European languages did not develop into heavily formatted languages because they relied on punched tape for their I/O. Thus, languages like ALGOL were developed with less formatting. But to continue my point, we all should give a quick thanks to those card reading days, in spite of all the jokes and sneers at the "do not fold, spindle or mutilate" era, because one of the benefits of that period is formatted I/O. Formatted I/O has many useful purposes. In the last article there was a simple program to get a name and return it to the CRT. It uses unformatted or list I/O:

```
instring:
  proc options (main):
    %replace
      TRUE by '1'b,
      CLEAR by '1L';
  dcl
      name char (128) var;
    put list (CLEAR);
    do while (TRUE);
      put skip (2) list ('enter your name:');
      get list (name);
      put list ('your name is ',name);
      end; /*do while*/
    end instring;
```

To briefly reiterate and also enhance where we left off, let's examine possible input for this program. When dealing with names being input, you are obviously going to be dealing with more than one "word." By that I mean that an entire name has spaces between the first, middle and last names such as this one:

William P. Hogan

In the above program there are already problems with this kind of input when dealing with "list I/O." Let's define a few terms. The first, middle and last names are called "tokens." The spaces between the tokens are called "delimitors." The important fact to concentrate on here is that spaces are string delimitors in PL/I. If we entered 'William P. Hogan' as input for the above program, the first token

(continued on next page)

('William') is going to be read as a separate string. The other two tokens will also each be read as separate strings —'P.' and 'Hogan'. To get the entire name entered without string delimitors, we can use the underscore character, like this:

William_P._Hogan

This eliminates the spaces that PL/I recognizes as string delimitors, and the entire name will be entered. For this sort of program, however, asking the operator to use underscore characters instead of spaces is obviously not suitable. Therefore, 'list I/O' will just not do for this particular program. Other languages have I/O restrictions. ISO Pascal, for example, has many restrictions on input. Pascal has a good excuse because it was created to be a teaching language, not a commercial applications language. Because PL/I is a commercial applications language, shortcomings in I/O cannot be tolerated. That's why PL/I has provisions for another kind of I/O, and the way around our dilemma is PL/I's edited I/O.

With 'edited I/O' there is no problem. We can take out this program statement

'get list (name);'

and substitute this one:

'get edit (name) (a);'

The '(a)' is used exclusively with edited I/O, and it specifies alpha data. The 'edit' indicates edited I/O. Now perhaps the significance of the program declaration makes more sense:

name char (128) var;

The 'name' is the string, 'char' specifies type character, '128' is the maximum number of characters and 'var' means "of varying length." To sum up, this statement tells the compiler that the 'name' variable will be a string of anywhere up to 128 alpha characters long:

```
edit_instring:
  proc options (main)
    %replace
      TRUE by '1'b,
      CLEAR by '1'L;
    dcl
      name char (128) var;/*the string can be this long*/
    put list (CLEAR);
    do while (TRUE);
      put skip (2) list ('enter your name :');
      get edit (name) (a); /*substitution is here*/
      put list ('your name is ',name);
      end; /*do while*/
    end edit_instring;
```

The substituted line 'get edit (name) (a);' tells the compiler to expect a string of up to 128 characters (the default length is 256 characters) which will be delimited by a carriage return.

So now when the name "William P. Hogan" is entered, it will be treated and stored as a single string. 'Edited I/O does not recognize anything except a carriage return as a delimiter. The advantage of all this is obvious. Now the language can act on a "what you see is what you get" basis.

Let's briefly compare list I/O and edited I/O and talk about a few advantages of each. List I/O is easier to write because you don't have to specify the length of the variable or constant being output (or input). When dealing with numeric data, list I/O is usually adequate, particularly if the numbers are within reasonable bounds. One specific advantage of list output is when you have to output something like this:

put skip list ('The amount is $',amount);

Whether the amount is $1.98 or $1,000,000.00 the output string will be contiguous:

The amount is $1.98

or

The amount is $1000000.00

Compare that with an edited I/O version of the same statement:

put skip edit ('The amount is $', amount) (a, f9.2);

This will output

The amount is $1000000.00

which is fine, but a smaller amount like $1.98 doesn't comes out very well because the field has to be specified, in this case nine characters in length:

The amount is $ 1.98

Some advantages of edited I/O are justification and precise columnar control. Edited output guarantees left justification and thus guarantees that all the decimal points will line up when numbers are expected to be in columns. When dealing with numbers, there are some specific advantages to limiting the input field. The following statement limits the input to a six digit number with no more than three decimal places:

get edit (float_number) (f(6,3));

The next program statement will output a 6 place decimal exponentiated or scientific number:

put edit (exponent_number) (e(6.3));

As we saw in our program example, there are some advantages in edited I/O when dealing with strings. The next program statement will limit the string input to no more than 32 characters:

get edit (name) (a(32));

Why would you want to limit the string input? There are many times when the parameters of the program will limit the number of characters that can be utilized in spite of the length of the input field — for example the output field of a report form or the field length within a file is usually restricted as to its length. A formatted input, or for that matter a formatted output, restricts the number of characters going to the field. The program will therefore be viable in spite of the potential for truncation. So much for comparisons. Input/output will be dealt with extensively throughout the article series, so let's go on.

Potential bugs with 'GET EDIT' statements

Let's stop momentarily and take a look at what we've learned. Trying to explain PL/I is like trying to explain an insurance policy — for every statement you make, there are often specific conditions and qualifications you have to point out as well. I'll capitalize this one. THE 'GET EDIT' STATEMENT READS EVERYTHING UP TO BUT NOT INCLUDING THE CARRIAGE RETURN. This fact is stated in Digital's language manual, but it might not register at first. Look at the following program segment:

```
put list ('enter first name');
get edit (name) (a);
put list ('enter second name');
get list (name2) (a);
```

This program segment operates as follows. The system reads 'name' and takes it into its buffer, but it leaves the carriage return to hang around and get in trouble. The next 'get edit' statement takes the itinerant carriage return and accepts it as a null string, putting nothing into its storage location. The result is that the program appears to skip over the second request for information. If this sounds like a pain in the neck, it is!

How to keep it from happening? Think of it as a garbage collection problem. Look at the program segment now:

```
put list ('enter first name');
get edit (name) (a);
get skip;
put list ('enter second name');
get edit (name2) (a);
get skip;
```

The 'get skip' does the garbage collection, picking up the stray ASCII 0dh (carriage return) and "throwing it away." Don't forget to add the last 'get skip', or the last carriage return will lurk around to haunt you. As a bonus, the 'get skip' can be used to get the program to stop execution. This sometimes comes in handy, especially involving input. Because the 'get skip' requires a carriage return to satisfy the statement, program execution cannot continue until a carriage return is received. So, you can do things like this:

```
put skip list ('press enter to continue ');
get skip;
```

A word of WARNING. The 'get skip' is not a "cure-all" because the stray carriage return is not entirely predictable. Sometimes it will not fly around loose and other times it will take two 'get skips' to trap the loose carriage return:

```
get (2) skip;
```

This may be corrected in future compilers. I would recommend that you either use the 'get skip' method and be prepared to remove those that overkill, or simply be prepared to put them in on an "as required" basis. The first takes less time.

Loops and repeating

In the first installment we briefly examined one form of a programmatic loop, a 'do-while'. Let's continue some discussion on various aspects of loops in programming in PL/I. The 'do while' keeps the program operating within the loop as long as the condition defined is true. By true, I refer to the boolean concepts of true and false. The do-while loop requires a boolean true condition in order to operate. When the condition is false, the loop is exited. Here's an abbreviated program segment where you see the do-while in action:

```
do while (TRUE);
  put list ('input name :');
  get edit (name) (a);
  .
  .
  end;/* do while */
```

The do-while loop will continue getting names as long as the predefined condition is true. The do-while loop is very similar to BASIC's 'while-wend':

```
true = -1
while true
  print "input name'
  input name$
  wend
```

How was the condition predefined as true in our example? Let's add two more lines to this code segment:

```
%replace TRUE
  by '1'b
do while (TRUE)
  put list ('input name :');
  get edit (name) (a);
  .
  .
  end; /*do while*/
```

In the last article we briefly discussed constants and how to use the %replace statement. Here's another use for %replace. The form '1'b or '0'b is called a bit string. In PL/I the bit string '1'b is defined as true, and '0'b is defined as false. So far, so good. We know how to predefine a simple true condition. In the above example, we are dealing with a do-forever because there is no specific provision for a false condition to exit the loop. Take a specific provision like "do while not end-of-file." This incorporates the use of the logical 'NOT', and this is the way it's done:

```
do while (name ↑= 'eof')
```

The caret sign (↑) is used to signify the logical 'not'. Now this statement causes the program to continue to loop until it reaches the end of the file. As long as the condition "not end-of-file" exists, the program will continue to loop:

```
%replace TRUE
  by '1'b;
do while (name ↑= 'eof')
  .
  .
  end; /*do while*/
```

A particularly useful form of the do statement is the iterative do. It will iterate for a finite number of times. The following mini-program prints the numbers 1 to 26 down the screen and stops:

```
dcl
  i fixed;

do i = 1 to 26;
  put skip list (i);
  end;
```

(continued on next page)

The BASIC equivalent would be this:

```
for i = 1 to 26
  print i
  next i
```

Let's put some of this knowledge to work using the iterative do and the do-while together. Here's a program to create a guest list for a party. The console asks the operator to input the name of the guest. The operator can input as many as 100 names, and everything is fine unless the program sees the infamous name of 'Uncle Harry':

```
guest__list:
  proc options (main);
    % replace
      CLEAR by '↑L';
  dcl
    name char (128) var;

  put list (CLEAR);
  do i = 1 to 100 while (name ↑= 'Uncle Harry');
    put list ('input the name of the guest');
    get edit (name) (a);
    get skip;
    call print__invitation (name);
    end; /* do while */
  print__ invitation;
    proc (name);
    dcl
      name char (128) var;
    put file (line__printer) list (name);
    put file (line__printer) list
      ('is cordially invited . . .etc.↑L');
    end print__invitation;
  end guest__list;
```

This incomplete program will print a guest list of no more than 100 guests as long as you don't invite Uncle Harry. If you do, the guest list stops right there.

There are a few loose ends I'd like to clarify. One of them is reminder statements (/* this is a reminder statement */). Anything preceded by a '/*' is ignored by the compiler. Any new line after '*/' is read. In the above program examples I have added the reminder statement '/*do while*/' after the 'end' statements of do-while loops. I do this to help me keep track of my end statements, specifically to make sure I have enough of them. Every procedure block and do-while loop needs an end statement to complete it. The compiler keeps rigorous track of the nesting levels in the program, and if the nesting levels don't balance, it gets upset. The program won't compile; and, adding insult to injury, the compiler gives you a nasty message on the console. This kind of error is called a nesting depth error. To help you keep track of your end statements, try using reminder statements. By the way, should you get blown out of the water with a nesting depth error, recompile the program using the N switch. Let's say the name of your program is BADPROG.PLI. Recompile it like this:

```
A>PLI BADPROG $N
```

A compilation using the 'n' switch forces the compiler to show the nesting level at the leftmost side of the code. A little detective work on your part will find the missing 'end'. Did you notice the control L at the end of the last 'put' statement:

```
put file (line__printer) list
  (' is cordially invited . . .etc.↑L');
```

By using the caret "↑" in the output line, PL/I will automatically alter the high order nibble to output a control character. We used ↑L to clear the screen in the last article, but here ↑L brings up a new sheet of paper (form feed) on the printer. Another loose end I want to tie up is the call, but that takes another section.

The call

I sneaked in a 'call' statement in our Uncle Harry program:

```
call print__invitation(name);
```

Another procedure (named 'print__invitation') is being "called" with this statement. Here is the entire procedure being called:

```
print__invitation:
  proc(name);
    dcl
      name char (128) var;
    put file (line__printer) list (name);
    put file (line__printer) list
      (' is cordially invited . . .etc.↑L');
  end print__invitation;
```

When another procedure is being called, the main procedure redirects the program flow to the called procedure. When the procedure being called is completed, the program flow automatically returns to the main procedure to the next line after the one that made the call. The calling of procedures is similar to BASIC's 'gosub', but gosub will not pass a parameter, and a call will. Before discussing what parameters are, I want to point out that some enhanced versions of BASIC have added a call, one of them being CB-80. If our Uncle Harry program were written in CB-80, the call statement would look like this:

```
call print.invitation (name$)
```

Back to parameters. A parameter is also referred to as an argument, particularly when it is used in the call statement. Parameters or arguments are the way that values (variables and constants) are passed to procedures and functions so that they may be acted upon (and in the case of functions a value returned). The procedure 'print__invitations' is called, and the string variable 'name' is "passed" to the 'print__invitations' procedure. In the Uncle Harry program example, the procedure being called ('print__invitations') is a "nested procedure" nested within the main program procedure, 'guest__list'. The name string (any name but Uncle Harry's) is handed or passed to the procedure by the call. Any name, except miserable old Uncle Harry, will be passed to the printer procedure, and an invitation will be printed.

Procedures do not have to be nested one inside the other, but sometimes nesting procedures have advantages. Nesting procedures keeps variables that are local to the outermost procedure global to the inner, nested, procedures. It also provides continuity to the program by associating the inner procedures logically with the outer procedures. If this explanation of the advantages of nested procedures is unclear to you at this point, never mind about that. Each article will gradually make these concepts clearer to you, and also procedures will be discussed some more a little later in this article.

By the way, I hope you noticed the use of list I/O in 'print__procedure'. The reason? I wanted the code in the called procedure to be as simple as possible because I only wanted to demonstrate the call and discuss nested procedures . Edited I/O would be much more appropriate for printing out invitations, but it takes more code. Edited I/O plain and fancy will be covered in another article.

Calling is an extremely powerful tool of not only PL/I, but any structured language that supports it. It allows the programmer to write a procedure just once, but call it as often as he needs to. It is also the way program flow is directed from one portion of the program to the other without using goto's. Proponents of "goto-less programming" can avoid goto's by 'call to's', but it is not appropriate to automatically substitute calls for goto's. Indiscriminate use of calls is bad, too. The most important aim of the programmer should be to make his code clear and easy to read. I feel that structured programming is the ONLY way to program. Indiscriminate use of goto's can make a program unstructured because they make the direction of the program flow confused or obscure. On the other hand, goto's are not always "bad" programming practice. In fact, sometimes they offer the only logical way to pursue your programmatic idea. Some languages like C provide break and continue statements to allow exiting a loop or returning to the loop test, but languages that do not provide these kinds of conveniences have to make use of goto's. In addition, goto's are almost impossible to avoid in exception processing. Another point — using a call as a substitute for a goto doesn't automatically make your program structured. Calls can be a cop out, too, if the person reading your code can't figure out why you put in the darned thing. Goto's should be used sparingly, and calls should be used with foresight. A lot of ground has been covered in a short time. The rest of the article will bring a lot of what we have been talking about into a more cohesive unit by reviewing what has been covered with extended explanations while we talk about a few more PL/I basics.

Blocks and scope of variables

When we talked about calls, we talked about procedures. Just what the heck are procedures anyway? They are one of many kinds of program blocks. PL/I, like Algol, is an Algol-like language, and all Algol languages like PL/I are block structured. A block is a grouping of one or more statements into "logical" units. Blocks come in all kinds of flavors. There are begin blocks, short form blocks, procedure blocks and function blocks (a variation of a procedure block). There is a lot to learn about blocks, and we are going to explore a few specifics right now.

First of all, blocks can be external or internal. An external block has no blocks surrounding it — it is not contained within any other blocks. The procedure " main" is an example of an external block. Internal blocks, on the other hand, are contained within another block. The nested procedure we saw in our Uncle Harry program is an internal block contained within the main procedure, an external block. The terms external and internal are also used to describe the relative positions of blocks. Internal blocks are said to be external to any other blocks within them. Hand-in-hand with the concept of external and internal blocks is the concept of "scope of variables." The scope of variables is a reference to where the variable is known by the program, and this is very closely related to the concept of external and internal blocks. For instance, variables declared inside an internal block are known to that block only — they are "local" to that block. On the other hand, variables declared outside of the main block are known to the entire program and are "global" to the program. Let's take a look at a basic begin block. A begin block starts with the word 'begin' and ends with the word 'end'. Begin blocks can be preceded with something called a 'label'. Look at the example below:

```
calvin:
  begin;
  dcl
    i fixed;
  do i = 1 to 32;
    put skip list ('this is an example of a block');
    end; /* do */
  end calvin;
```

Calvin is the label for this begin block. If it is necessary to refer to this begin block, the label will be used, and we will use it right now by referring to this block as "block calvin." Now let's look at the variables declared within the block. The 'i' variable will have its value only within the block calvin — it is 'local' (known only) to that block. What happens when a begin block is encountered by the program? The program flow will enter at the top, execute through to the bottom and exit. It's all easy enough, but remember this particular point when we look at procedures, another kind of block. The program flow works differently with procedures, as you will soon see.

Another kind of a block is the "short form block." It is used in certain programmatic applications like exception processing (error-proofing the code):

```
on endfile (disk__file)
  begin;
  close file (disk__file);
  goto end__loop;
  end;
```

The "on endfile (disk__file)" is an end-of-file condition, and whenever this condition is raised as the program executes, this entire begin-end block will be executed. Those of you familiar with Pascal will notice that PL/I's begin-end block is similar to Pascal's.

Now we can look at procedures. Procedures are blocks that must be called. A begin block has an optional label, but a procedure MUST have a label. (Otherwise, how would you call it?)

```
gucci:
  proc;
  dcl
    number fixed static init (1);
```

(continued on next page)

```
do while (number ~= 10);
   put skip list (number)
   number = number + 1;
   end; /*do while*/
end gucci;
```

'Gucci' is a label. The value of the variable 'number' in the declaration is local to the gucci procedure only. (By the way, don't let this declaration throw you just because it's different. Briefly, here's what it does. It initializes 'number' to one. Numbers cannot be initialized unless the storage class has been declared static. Thus the term "static" is included in the declaration, and it signifies that the storage allocated for the variable will be held open for the length of the program. The term "fixed" signifies fixed integer, like type integer in other languages. The '(1)' in the declaration means that 'number' starts off with a value of one. All of this will be covered later, but I just threw a different kind of declaration in so you can be aware that there are many different kinds. The important thing to remember right now is that the value of the variable 'number' is local to this procedure only. Incidently, the '~=' symbol in the program line 'do while (number ~= 10);' is another logical 'not', and the meaning of that line is simply "do while number is not equal to ten." What this program does is print the numbers 1 through 9 on the screen. How does the program flow enter this procedure? The only way this procedure can be entered is by calling it. The procedure will then execute and return to the line after the line that called it. Here's a program line that will call this procedure:

```
call gucci;
```

A procedure cannot be entered by the program flow "falling through" and into it. Remember in BASIC when you have forgotten to stop the program at the end of its logical execution, and it "falls through" into the subroutine you have put at the bottom? Then you get that irritating "return without gosub" error? It can't happen with a procedure. The procedure won't allow itself to be entered except by a call. Program flow is precisely controlled by this feature, and it's one of the joys of structure. Procedures are usually called from the main procedure. They can be called from any point in any procedure, but if they are called from the main procedure, the program logic is more apparent and it's easier to for someone reading the code to follow it. It is extremely important in any language that programs be easily "human" readable. If a program cannot be easily read, it cannot be easily maintained.

A procedure can have parameters passed to it the way we passed the value of 'name' to the 'print_invitations' procedure in our Uncle Harry program. We described what parameters are earlier in this article, but before we take another look at them, we had better define some more terms. A blanket use of the term "parameters" is not precisely correct, so let's get more specific. The data items that are to be "passed" to a procedure are called 'arguments' or 'actual parameters'. When they are received by the procedure, they are called 'formal parameters'. I used 'name' in our Uncle Harry program to signify the actual parameter and the formal parameter, but they are not always the same name. In fact, most often they are not. Enough talk. It makes more sense when you look at the code:

```
proc options (main):
.
call jordache (name); /*name is the actual parameter*/
next_line;
.
.
jordache:
   proc (string_in); /*string_in is the formal
      parameter*/
dcl
   string_in char(32) var;
put skip list ('preferred customer : ',string_in);
.
end jordache;
```

What do we have here? The main procedure calls the jordache procedure, and it passes to the jordache procedure the argument or actual parameter 'name'. The jordache procedure receives 'name' as the formal parameter 'string_in'. All this program fragment does with the data item being passed is print it out on the screen. Let's say "Ima Jeanslover" is the 'name' which has been input in the main procedure. It is passed to the jordache procedure which prints the following out to the screen:

```
preferred customer : Ima Jeanslover
```

Looking at the scope of variables in this program segment, 'name' is global to the program, but 'string_in' is local to the procedure jordache only. The various blocks we have discussed are pretty straightforward, but all this talk about local and global variables is probably a little confusing. Some of you might be muttering "Who cares whether the darn variable is global or local to one block or another?" That's a very good question. We are not going to go into that in this article, but keep that question in mind! For now, at least, you are cognizant of the fact that PL/I offers you the option of controlling where your variables will be known in the program. This offers you a significant degree of programming power, and in later articles you will see why.

Blocks can't be discussed without mentioning the return. Here's but a brief peek at the return. You can use the return at any point within a block to allow the program flow to go back to the line immediately following the line which called the block in the first place. You can do a simple return, or you can return a value.

```
return;
return (x);
```

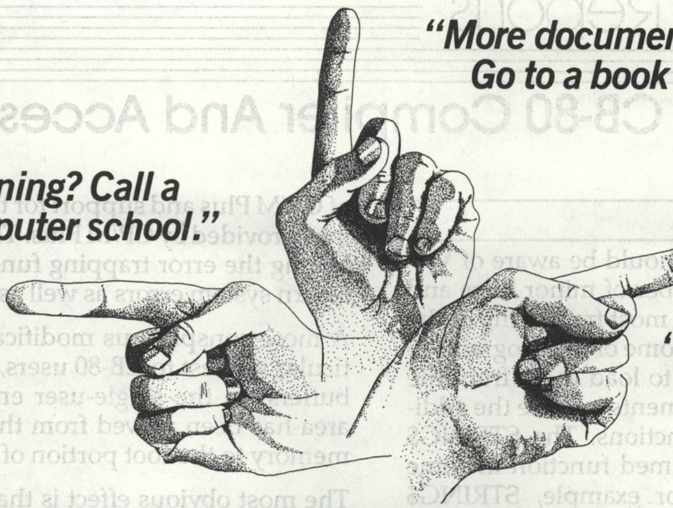
You can even do multiple returns within a block, and they can be downright handy:

```
if weight = 0 then
   return;
 else if weight > 200 then
   return ('fat')
 else
   if weight < 100
      return ('skinny');
      else
        return ('normal');
```

Let's stop here. In the future we'll finish up the very last of Chapter One with brief looks into declarations, variable types and files. We'll also be looking at Chapter Two which explores the fascinating world of edited I/O, plain and fancy.

"More documentation? Go to a book store."

"Training? Call a computer school."



"Technical support? Call the publisher."

Interested in dBASE II™ or 1-2-3™? Beware The Dreaded Finger Pointers!

Sound familiar? Does your dealer turn into a "finger pointer" when you need help?

At SoftwareBanc we offer a complete system that doesn't stop when your software is delivered.

Careful Product Selection

Do you get bewildered by the endless lists of software you find in most ads? Let us be your quality control department.

We only sell the best programs on the market. After a thorough evaluation we chose dBASE II™ for data processing, and 1-2-3™ for financial management.

Our complete line of add-on products help you to continue to get the most from your software.

Expert Technical Support

When you buy software from us, you can rest assured that help is only a phone call away. Just call us at (617) 641-1235 for all the free support you need.

Free dBASE II™ User's Guide

Order dBASE II™ from us, and you'll receive a free copy of our dBASE II™ User's Guide. You can also buy the User's Guide first for only $29, and then receive a full credit when you buy dBASE II.™

French Translation
La Commande Electronique
5 Villa Des Entrepreneurs
75015 Paris, France

Japanese Translation
JSE Int'l
9F Toyo Bldg. 6-12-20 Jingmae
Shibuya-ku Tokyo, Japan 150

1-2-3™ & dBASE II™ Classes

Want more in-depth information about dBASE II™ or 1-2-3™? Attend a SoftwareBanc Seminar near you. Each session runs from 9 to 5, and costs $175. Seminars are in lecture format with a custom sound & video system which is used to display taped interviews with prominent software personalities and sessions with various software programs.

Los Angeles
July 18-22

Washington, D.C.
Aug. 29-Sept. 2

Anchorage
August 11-12

New York City
September 19-23

Prices You Can Afford

†1-2-3™	$399
†dBASE II™	$479
†ABSTAT™	$379
dBASE II™ User's Guide	$29
DBPlus™	$95
dGRAPH™	$199
dUTIL™	$69
dNAMES™	$109
QUICKCODE™	$199
TEXTRA™	$60*

†No-risk 60 day money back guarantee
*Only available for IBM PC with MS-DOS.

Free Catalog

If you want to learn more about SoftwareBanc, call or write for our free product catalog.

SoftwareBanc

661 Massachusetts Avenue
Arlington, Mass. 02174
For technical support call:
(617) 641-1235
Dealer Inquiries Invited.

™Manufacturer's trademark
Payment may be made by: MasterCard, Visa, check, C.O.D., money order. Mass. residents please add 5% sales tax. Add $5.00 for shipping and handling. Prices subject to change.

SoftwareBanc
Order Toll Free
1-800-451-2502
(617) 641-1241 in Mass.
SoftwareBanc
661 Massachusetts Avenue
Arlington, MA 02174

New Versions for CB-80 Compiler And Access Manager

by Robert P. VanNatta

CB-80 COMPILER Version 1.4

Users of Digital Research CB-80 should be aware of Version 1.4. This version fixes a number of minor bugs and adds two new functions. Perhaps, most frustrating of the bugs fixed involved the refusal of some of the programs to function if an attempt was made to load them from the WordStar 'R' command. Enhancements include the addition of STRING$ and SHIFT functions. The STRING$ function imitates the similarly named function in some versions of Microsoft BASIC. For example, STRING$ (30000,'A') will generate a string made up of the character 'A' thirty-thousand characters long. String building accomplished using this function reduces memory fragmentation and executes faster than concatenation routines (such as A$ = 'A' + 'A'). It is a bit difficult to benchmark the generation of 30k strings in a microcomputer, however, a routine such as:

```
print fre,mfre                  rem  check memory
print "hit key to begin"
a% = inkey                      rem  start
for i% = 1 to 100               rem  loop 100 times
a$ = string(30000,"a")          rem  build 30k string
a$ = null$                      rem  destroy it
next i%
print "done"
print fre,mfre                  rem  recheck memory
```

was observed to execute in about 32 seconds. The FRE and MFRE functions returned the same answers both before and after execution suggesting that this loop caused no memory fragmentation or other memory loss.

The SHIFT function follows the format of SHIFT(I%,N%) and executes a shift right or binary division. Stated another way, this function will divide I% by 2 to the nth power.

The following program demonstrates the shift function and the equivalent divide function. It is, hopefully, obvious that if you attempt to 'shift' a 16-bit integer more than 15 bits to the right you will effectively shift it right out of the register and the result will be zero!

```
10   input "enter dividend"; I%
     input "enter bits to shift";N%     rem must be positive number
     print "divide";a%/(12↑N%)          rem divide function (old way)
     print "shift";shift (I%,N%)        rem shift function (new way)
     goto 10
```

This bit maneuver is dramatically more efficient than a division and ought to be used instead of the division function whenever possible.

ACCESS MANAGER Version 1.1

Access Manager (AM-80) version 1.1 became available from Digital Research in April of 1983. This is the Digital Research file management utility intended for use with the Digital Research language family. Besides some bug corrections, enhancements were made in two areas. The first area involves support for CP/M Version 3.0 (CP/M Plus). This includes support for the PASSWORD feature of CP/M Plus and support for the system level error trappings provided by CP/M Plus. The latter is accomplished by having the error trapping function (ERRCOD) of AM-80 return system errors as well as Access Manager errors.

A more conspicuous modification which will be of particular interest to CB-80 users, however, relates to the file buffers. In the single-user environment, the file buffer area has been moved from the data-segment (DSEG) of memory to the root portion of the code segment (CSEG).

The most obvious effect is that the code segment will be several thousand bytes larger than it was previously. (The exact size of the file buffer is defined by the programmer.) The trade off in this change involves the loss of the possibility of using the file buffer area for other purposes (when the files are not open) in exchange for the ability to keep files open across overlays.

Stated another way, reliance on the CB-80 exit routine to close all files is no longer possible. To the contrary, files opened under the control of Access Manager now stay open until explicitly closed. Program overlays may be freely exchanged during program execution without the bother of closing and reopening files.

Feature

TXT.ASM DB Format

by Thomas Hill

Introduction

As a programmer I find myself creating menus and 'help' screens many times for the various end user programs I write. Since much of my work is written in assembly-level code, I eventually wind up spending a large amount of programming time attempting to paginate, justify margins, and line up tabular columns for the screen and printer output. I finally decided to create a program to help me prepare formatted text material for assembly level programs. The program described here is the result.

Abstract

The program TXT2ASM accepts as input the name of a file containing the text to be included in the assembly program, and outputs a file containing the text reformatted into 'DB' statements suitable for assembly by any CP/M assembler. The format of the command line is:

A>TXT2ASM <infile> [<outfile>]

where <infile> is the unambiguous name of the text file and <outfile> is an optional output file name. If no output file name is specified the reformatted text is sent to the file <infile name>.DB . During reformatting, certain control character sequences are translated into equivalent assembler labels. In particular, the carriage return is output as "CR" and the line feed control is output as 'LF'. The formfeed control is also translated into "FF" in the output file. All other ASCII characters are passed untouched to the output file. The general format of each output line is:

<tab>DB<tab>'<text>'

In other words, each line begins with a leading tab, followed by the assembler operator "DB" and another tab. Any text is enclosed within single quotes. Imbedded quotes are translated to double quotes, as required by the assembler. Output lines are 'broken' at sixty characters, since Digital Research's MAC macro assembler is upset by text lines longer than 64 characters. This feature is controlled by an equate near the beginning of the program, if you wish longer (or shorter) output lines.

Note that the program as written utilizes the Z80 macro library which is included with the D.R. MAC program. For those of you with 8080s, it should not be too difficult a task to rewrite the Z80 operations into their 8080 equivalents. The primary Z80 operations used are the relative jumps and double register stores and recalls. Also used is the block transfer instruction, LDIR.

About the program

The program is written in a modular (almost) fashion to allow easy alterations in the future. The entry point is the label FILTER, which is reached after skipping over the embedded copyright notice (which is also used as the signon message). Note that the copyright notice is terminated by a CTRL-Z (EOF). This allows the user to TYPE the .COM file if s/he wishes to examine the signon message without executing the program. The code at FILTER examines the primary File Control Block (FCB) to find out if the user specified an input file. If no input file was entered, the program prints a short description of usage and format and returns to the CP/M level. If an input file has been entered, the program checks the secondary FCB looking for the optional output file name. If the secondary FCB contains a file name, it is transferred to the internal output FCB, else the input file name is used with the type changed to " .DB ".

After setting up both FCBs the next step is to open the files for input and output. The input file is first opened, with an ERROR exit if the BDOS says the file doesn't exist. Then the output file is first deleted, then re-created to make sure we don't wind up with two files with the same name on the disk. If an error occurs here, we abort to the CP/M level with an error message indicating a "no disk space" condition.

Now that both files are ready for I/O, the program initializes the various pointers, flags, and counters used later and calls the FILL$BUF routine. This subroutine reads the entire input file into memory starting at the label IN$BUF. A check is made after reading each sector to make sure we don't overwrite the BDOS. If the input file tries to exceed the available TPA, an error message is output and the program aborts to the CP/M level. (It might be better to just process what we have in memory, but I don't really anticipate text files that large for assembly programs.

The input text has been loaded to memory. We can now begin processing it for output. This is accomplished in the loop at label LP2. The loop calls SET$LINE, MAKE$LINE, and WRT$LINE in sequence until MAKE$LINE detects an end-of-file marker and sets the EOF$FLG flag. When the program detects the change in the flag, it purges the partial output buffer to disk, prints a count of the number of input and output lines, closes both files, and returns to CP/M level.

Most of the processing is performed in the label MAKE$LINE. The subroutine SET$LINE preloads the output line buffer with the leading tab,'DB', and tab and clears the FIRST and IN$QUOTE flags. The MAKE$LINE routine fetches each character from the input buffer and examines it for the special characters that must be translated. If a carriage return, line feed, form feed, or single quote is detected, the program vectors to the appropriate routine, which places the proper label(s) in the output

(continued on next page)

line. The PUT$CR routine requires special attention because it indicates the end of the input line. Line feeds following the initial carriage return are taken care of and a real carriage return, line feed pair is placed in the output line. Output lines are counted here.

Notice that each of the special character routines must keep track of whether the output line is currently 'inside' a quoted string, and must close the quote if needed. We must also know whether we are at the beginning (FIRST) of a text line, in order to decide whether to place a separating comma or not.

When the MAKE$LINE routine detects an EOF in the input stream, it creates a special output line consisting of a quoted dollar sign ('$'), which is the CP/M End-of-message marker. This line is the last line of the output file. You may, of course, place your own EOM markers within your text for multiple messages. The '$' will be passed to the output file without modification.

During construction of the output line, a count is kept of the number of characters which have been placed in the output line buffer. If the count exceeds the value of MAX$LEN (default 60), the current output line is terminated and control returns to the processing loop. This is in deference to certain assemblers which become upset with text lines longer than a certain number of characters.

After each output line has been built, it is written to the disk buffer by the WRT$LINE routine. As WRT$LINE moves the formatted output line to the buffer, it watches the buffer capacity. When the buffer becomes full, it is written to the output file and the buffer pointer is reset to the beginning of the buffer. After each write to the disk, an error check is made for a disk full condition. If the disk becomes full during writing, the output file is closed to retain what has been formatted and the program aborts to CP/M.

At the end of input processing, any partial buffer contents are flushed to the disk by the subroutine FLUSH. FLUSH computes the number of sectors containing valid data, adds one for good luck, and fills the balance of the buffer with EOF markers. It then writes the calculated number of sectors to the output file.

After all processing has been performed and the output buffer has been flushed, counts for input lines and output lines are calculated and displayed. Both line counts are maintained in double precision memory locations (probably overkill, but I might have more than 255 lines of text). The routine HL2DEC converts the contents of the HL register pair into ASCII decimal digits and prints the resulting value. Leading zeros are suppressed, except for a (possible) zero in the units position. The value conversion is performed using conventional power-of-ten subraction in the routine CNVRT.

The balance of the program is data area, containing error and status messages, pointer and flag storage, and buffer areas.

Further modifications and bugs

To my knowledge, there are no bugs present, but I wouldn't swear to it. Future versions are not really being considered right now, since the program does just about what I want it to do. It may be advisable to add translation of the CTRL-G (bell) character into the label "BELL", just to be complete. Other special control codes or characters may be translated with ease in the MAKE$LINE routine, at the user's option. Echo of the output lines to the terminal or printer may be added, if desired.

Afterthought

Notice that the formatted output contains the symbolic labels "CR", "LF", and "FF". These labels MUST be established in your program, somewhere. The usual method is to place the following equates near the front of the program:

```
CR  EQU  0DH
LF  EQU  0AH
FF  EQU  0CH
```

If these labels are not defined, you will get all kinds of error reports when the main program is assembled.

Mercenary thoughts

If you are like me, you don't care to type reams of source code into your system when you see a great program in a magazine. For those of you who prefer to spend a little lucre rather than develop letter imprints on your fingertips, send me $25.00 and I will mail you a single density 8" disk (sorry, that's the only format I have) with the source code for ERAQ, SETIO, SETATR, and TXT2ASM, plus the text files for the articles describing each of these programs. (Like Kelly Smith, I hope to make millions this way).

```
; Author        Thomas Hill
;               200 Oklahoma St.
;               Anchorage, AK 99504
;               (907) 337-1984

; Modifications & updates (in reverse order):
;
; 10/09/82      Version 1.1
;               Cleaned up code
; 09/30/82      Version 1.0

; This program accepts as input a text file created by any of the CP/M
; text editors. It outputs a file formatted as "DB" statements suitable for
; inclusion in .ASM and .MAC assembly files. Carriage return, linefeed
; sequences are translated into the label sequence "CR,LF." Tabs are
; passed unaltered and form feed control codes are translated into the
; label "FF." In deference to the MAC assembler, which dislikes quoted
; text lines longer than 64 characters, input lines longer than 60
; characters are broken into two or more output lines.

; Note that this version assumes that the input text will fit in available
; TPA.

; SYSTEM EQUATES

CPM     EQU     0
BDOS    EQU     CPM+0005H       ; BDOS ENTRY POINT
FCB1    EQU     CPM+005CH       ; CP/M FILE CONTROL
                                ; BLOCK
FCB2    EQU     CPM+006CH       ; SECOND FILE
                                ; CONTROL BLOCK
CBUF    EQU     CPM+0080H       ; DEFAULT COMMAND
                                ; BUFFER
TPA     EQU     CPM+0100H       ; USER PROGRAM
                                ; AREA
```

```
; NON-DISK I/O FUNCTIONS
CONIN     EQU    1         ; CONSOLE INPUT
CONOUT    EQU    2         ; CONSOLE OUTPUT
LSTOUT    EQU    5         ; LIST DEVICE OUTPUT
PRTBUF    EQU    9         ; SEND A STRING TO
                           ; THE CONSOLE
RDBUF     EQU    10        ; GET A STRING FROM
                           ; THE CONSOLE
CONSTAT   EQU    11        ; CONSOLE STATUS
VERS      EQU    12        ; RETURN CP/M (MP/M)
                           ; VERSION NUMBER

; DISK I/O FUNCTIONS
SELDSK    EQU    14        ; SELECT DISK
OPENF     EQU    15        ; OPEN FILE
CLOSEF    EQU    16        ; CLOSE A FILE
DELETF    EQU    19        ; DELETE A FILE
READF     EQU    20        ; READ A RECORD
WRITEF    EQU    21        ; WRITE A RECORD
MAKEF     EQU    22        ; CREATE A FILE
SETDMA    EQU    26        ; SET DISK DMA
                           ; ADDRESS

; THOSE FUNCTIONS REQUIRING A BYTE ARGUMENT WILL
; EXPECT THAT BYTE TO BE IN THE E REGISTER. ADDRESS
; ARGUMENTS ARE PASSED IN THE DE REGISTER. RETURN
; CODES ARE PASSED IN THE ACC. IN GENERAL, A RETURN OF
; ZERO INDICATES SUCCESS. WHILE A 0FFH INDICATES FAILURE.

; character equates
CR        EQU    0DH       ; carriage return
LF        EQU    0AH       ; line feed
ESC       EQU    1BH       ; escape code
EOF       EQU    1AH       ; end-of-file, control-z
BELL      EQU    07H       ; terminal bell
BS        EQU    08H       ; backspace
TAB       EQU    09H       ; tab char
APOS      EQU    ''''      ; apostrophe
FORMF     EQU    0CH       ; form feed
;
FALSE     EQU    00H
TRUE      EQU    0FFH
;
MAX$LEN   EQU    60        ; maximum length for
                           ; output lines

MACLIB    Z80

          ORG    TPA

TXT2ASM:
          JMP    FILTER                      ; over copyright notice
SIGNON:   DB     'Text to .ASM Formatting Program',cr,lf
          DB     'Copyright October, 1982 by '
          DB     'Thomas N. Hill',cr,lf,LF,'$',eof

FILTER:   LXI    D,SIGNON
          CALL   PMESS
          LDA    FCB1+1                      ; check for input file
          CPI    ' '
          JZ     USAGE                       ; tell how to use.

; check for second file name as output

          LDA    FCB2+1
          CPI    ' '
          CZ     USE$SAME                    ; use the same file
                                             ; as output
          LXI    H,FCB2
          LXI    D,OUT$FCB                   ; move to output FCB
          MVI    B,12
          LDIR                               ; move it
          MVI    B,24
LP1:      MVI    M,0                         ; fill rest of FCB with zeros
          INX    H
          DJNZ   LP1

; input and output FCBs are set. Now open input file and fill buffer
          LXI    D,FCB1
          MVI    C,OPENF
          CALL   BDOS                        ; try to open the input
          INR    A
          LXI    D,NO$OPEN
          JZ     ERROR                       ; nothing there to open.
          LXI    D,OUT$FCB
          MVI    C,DELETF                    ; remove any existing
                                             ; output file
          CALL   BDOS
          XRA    A
          LXI    B,24
ZERO:     STAX   D
          INX    D
          DJNZ   ZERO                        ; re zero fcb
          LXI    D,OUT$FCB
          MVI    C,MAKEF                     ; and re-create it
          CALL   BDOS
          INR    A
          LXI    D,NO$MAKE
          JZ     ERROR                       ; can't make file, no room

; both files open, set up counters and pointers

          LXI    H,0
          SHLD   I$LN$CNT                    ; input lines
          SHLD   O$LN$CNT                    ; output line count
          LXI    H,IN$BUF
          SHLD   IN$PTR                      ; input buffer pointer
          LXI    H,OUT$LINE
          SHLD   OUT$PTR                     ; output line buffer
          LXI    H,OUT$BUF
          SHLD   OUT$DSK                     ; disk output buffer

; pointers & counters set, begin filtering operation

          CALL   FILL$BUF                    ; fill the input buffer
LP2:      CALL   SET$LINE                    ; set up the output line
          CALL   MAKE$LINE                   ; and make one
          CALL   WRT$LINE                    ; write the line to disk
          LDA    EOF$FLG                     ; was there an EOF
                                             ; during input?
          ORA    A
          JRZ    LP2                         ; nope, continue
          CALL   FLUSH                       ; yes, flush remaining
                                             ; output
KILL:     LHLD   I$LN$CNT
          CALL   HL2DEC                      ; print number of input
                                             ; lines
          LXI    D,IN$MSG
          CALL   PMESS
          LHLD   O$LN$CNT                    ; output lines
          CALL   HL2DEC
          LXI    D,OUT$MSG
          CALL   PMESS
          LXI    D,OUT$FCB
          MVI    C,CLOSEF
          CALL   BDOS                        ; shut things down
          INR    A
          LXI    D,NO$CLOSE
          JZ     ERROR                       ; oh,oh . . . can't close file!
          JMP    CPM                         ; return to operating
                                             ; system

; fill the input buffer. set EOF flag if we bump into end of file.

FILL$BUF:
          LXI    H,IN$BUF
```

(continued on next page)

```
FILL1:     SHLD    IN$PTR
           XCHG
           MVI     C,SETDMA
           CALL    BDOS         ; tell BDOS where to
           LXI     D,FCB1
           MVI     C,READF
           CALL    BDOS         ; read something
           ORA     A
           JRNZ    FILL4        ; got EOF, set flag
           LHLD    IN$PTR
           LXI     D,80H
           DAD     D            ; next sector
           LDA     BDOS+2       ; check for no room in
                                ; memory
           CMP     H
           JRNC    FILL1
           LXI     D,NO$MEM
           JMP     ERROR
FILL4:     LXI     D,CB1
           MVI     C,CLOSEF
           CALL    BDOS         ; done with that file
           LXI     H,IN$BUF
           SHLD    IN$PTR       ; set pointer to start
                                ; of buffer
           RET

; set up the output line buffer.
; preload the initial tab, DB, and tab
SET$LINE:
           LXI     D,OUT$LINE
           LXI     H,LN$MSK
           LXI     B,MSK$LEN
           LDIR                 ; put the line beginning
                                ; in place
           SDED    OUT$PTR      ; save current location in
                                ; output line
           XRA     A
           STA     NUM$OUT      ; reset output character
                                ; count
           STA     FIRST
           STA     IN$QUOTE     ; and flags for comma
                                ; control
           RET

; here we do the work.
; Get characters from the input buffer, translating CR,LF, and FF
; controls to proper labels. Place quotes around text strings, making
; sure to double imbedded quotes. Watch placement of commas and
; length of output line.
MAKE$LINE:
           LHLD    IN$PTR       ; current buffer pointer
           LDED    OUT$PTR      ; output line pointer
MAKE0:     MOV     A,M          ; get char from input
           CPI     EOF          ; end of input?
           JZ      END$IT
           CPI     CR
           JZ      PUT$CR       ; mark newline
           CPI     LF
           JZ      PUT$LF       ; some lines don't have
                                ; CRs
           CPI     FORMF
           JZ      PUT$FF       ; and form feeds
           CPI     APOS         ; imbedded quote?
           JNZ     MAKE1
           CALL    PUT$APOS     ; put in two apostrophes
           CALL    PUT$APOS
           INX     H
           JR      MAKE0
; if we got here, then plain character
```

```
MAKE1:     MVI     A,TRUE
           STA     FIRST        ; not first char anymore
           LDA     IN$QUOTE     ; are we inside quoted
                                ; string?
           ORA     A
           JRNZ    MAKE2        ; yes, don't place apos
           CALL    PUT$APOS     ; no, mark start of text line
           MVI     A,TRUE
           STA     IN$QUOTE     ; we are now.
MAKE2:     MOV     A,M          ; recover character
MAKE3:     INX     H
           STAX    D            ; place character in
                                ; output line
           INX     D
           CALL    OUT$CNT      ; count output chars
           CPI     MAX$LEN      ; reached end?
           JNZ     MAKE0
           LDA     IN$QUOTE
           ORA     A            ; inside quoted line?
           JRZ     MAKE4        ; nope.
           CALL    PUT$APOS     ; end of quoted text
MAKE4:     MVI     A,CR
           STAX    D            ; real end of line
           INX     D
           MVI     A,LF
           STAX    D
           SHLD    IN$PTR       ; save current buffer
                                ; pointer
           RET
END$IT:    LDA     IN$QUOTE
           ORA     A            ; inside quoted string?
           JRZ     ENDIT1
           MVI     A,FALSE
           STA     IN$QUOTE
           CALL    PUT$APOS     ; yes, mark end
ENDIT1:    LDA     FIRST
           ORA     A            ; first position in line?
           JRZ     ENDIT2
           CALL    PUT$COMMA    ; nope, need a comma
                                ; here
ENDIT2:    CALL    PUT$APOS
           MVI     A,'$'        ; mark end of text
           STAX    D
           INX     D
           CALL    PUT$APOS     ; with CP/M EOM
                                ; marker
           MVI     A,TRUE
           STA     EOF$FLG
           JMP     MAKE4        ; end end line

; here are the various "PUT" subroutines
PUT$CR:    PUSH    H
           LHLD    I$LN$CNT
           INX     H
           SHLD    I$LN$CNT     ; count input lines
           POP     H
           LDA     IN$QUOTE
           ORA     A            ; inside quoted string?
           JRZ     PUTSC1
           CALL    PUT$APOS     ; close it first
           MVI     A,FALSE
           STA     IN$QUOTE     ; reset flag
PUT$C1:    LDA     FIRST
           ORA     A            ; beginning of line?
           JRZ     PUT$C2       ; yep, no comma
           CALL    PUT$COMMA
PUT$C2:    MVI     A,TRUE
           STA     FIRST        ; not beginning anymore
           MVI     A,'C'
```

```
         STAX   D               ; put "CR" label in place
         INX    D
         MVI    A,'R'
         STAX   D
         INX    D
PUT$C3:  INX    H
         MOV    A,M             ; take care of LF
         CPI    LF
         JNZ    MAKE4
PUT$LF:  LDA    IN$QUOTE
         ORA    A
         JRZ    PUT$L1
         CALL   PUT$APOS
         MVI    A,FALSE
         STA    IN$QUOTE
PUT$L1:  LDA    FIRST
         ORA    A
         JRZ    PUT$L2
         CALL   PUT$COMMA
         MVI    A,TRUE
         STA    FIRST
PUT$L2:  MVI    A,'L'
         STAX   D               ; put "LF" label in place
         INX    D
         MVI    A,'F'
         STAX   D
         INX    D
         JR     PUT$C3          ; look for some more

PUT$APOS:
         MVI    A,APOS
         STAX   D
         INX    D               ; put an apostrophe in
                                ; output line
         RET
PUT$COMMA:
         MVI    A,' , '
         STAX   D
         INX    D
         RET

PUT$FF:  MVI    A,'F'
         STAX   D
         INX    D
         STAX   D
         INX    D
         RET

; count output chars, return count in A

OUT$CNT:
         LDA    NUM$OUT
         INR    A
         STA    NUM$OUT
         RET

; write output line to buffer. When buffer fills, write it to disk.

WRT$LINE:
         PUSH   H
         PUSH   D
         LDED   OUT$DSK
         LXI    H,OUT$LINE
WRT1:    MOV    A,M
         STAX   D               ; move line till see a LF
         LXI    B,O$BUF$TOP     ; buffer top
         MOV    A,B
         CMP    D
         JNZ    WRT4
         MOV    A,C
         CMP    E
         JNZ    WRT4

; buffer full, write it out
         PUSH   H               ; save current line
                                ; pointer
         MVI    A,BUF$SIZE
         CALL   WRT$BUF
         LXI    H,OUT$BUF
         SHLF   OUT$PTR
         POP    H               ; output line pointer
         JMP    WRT1            ; continue with move, if
                                ; any left
WRT4:    MOV    A,M             ; get char again
         INX    H
         INX    D
         CPI    LF              ; end of line?
         JNZ    WRT1
         SDED   OUT$DSK         ; save current location
                                ; in buffer
         LHLD   O$LN$CNT
         INX    H
         SHLD   O$LN$CNT
         POP    D
         POP    H
         RE

; flush rest of output buffer to disk

FLUSH:   LHLD   OUT$DSK         ; fill rest of buffer with
                                ; EOFs
FLUSH1:  MVI    M,EOF
         INX    H
         MOV    A,L             ; till next page break
         ORA    A
         JRNZ   FLUSH1
         LXI    D,-OUT$BUF
         DAD    D
                                ; calc. number of sectors
                                ; to write
         MOV    A,H             : pages
         ADD    A               ; double for sectors
         INR    A               ; plus one for safety
WRT$BUF:
         LXI    H,OUT$BUF
WRT2:    PUSH   PSW
         SHLD   OUT$DSK
         XCHG
         MVI    C,SETDMA
         CALL   BDOS
         LXI    D,OUT$FCB
         MVI    C,WRITEF
         CALL   BDOS
         ORA    A
         JRNZ   WRT5
         POP    PSW
         DCR    A
         RZ
         LHLD   OUT$DKS
         LXI    D,80H
         DAD    D
         JR     WRT2

; write error, close what we have and abort
WRT5:    LXI    D,NO$WRITE
         CALL   PMESS
         JMP    KILL

; double precision convert HL to decimal

HL2DEC:  LXI    B,10000
         CALL   CNVRT             10,000s digit
         MOV    A,E
         ORA    A               ; skip leading zeros
         JRZ    HLD1
         CALL   DIGOUT
```

(continued on next page)

```
HLD1:       LXI     B,1000
            CALL    CNVRT           ; 1,000s digit
            MOV     A,E
            ORA     A
            JRZ     HLD2
            CALL    DIGOUT
HLD2:       LXI     B,100
            CALL    CNVRT
            MOV     E,A
            ORA     A
            JRZ     HLD3
            CALL    DIGOUT
HLD3:       LXI     B,10
            CALL    CNVRT
            MOV     A,E
            ORA     A
            JRZ     HLD4
            CALL    DIGOUT
HLD4:       MOV     A,L             ; print last digit,
                                    ; even a zero
            CALL    DIGOUT
            RET
CNVRT:      MVI     E,-1
CONV1:      INR     E
            DAD     B
            MOV     A,H
            ORA     A
            JP      CNV1
            MOV     A,B
            CMA
            MOV     B,A
            MOV     A,C
            CMA
            MOV     C,A
            INX     B
            DAD     B
            RET

DIGOUT:     ADI     '0'
PUSH        H
            PUSH    D
            PUSH    B
            MOV     E,A
            MVI     C,CONOUT
            CALL    BDOS
            POP     B
            POP     D
            POP     H
            RET

; no input file, tell user what to do

USAGE:      LXI     D,USE$MSG
            CALL    PMESS
            JMP     CPM

; error routine, print message at (DE) and abort to CP/M

ERROR:      CALL    PMESS
            JMP     CPM

CRLF:       LXI     D,CRLF$MSG      ; save some code here
PMESS:      MVI     C,PRTBUF
            JMP     BDOS

; using same input file name for output. keep name, make type = .DB

USE$SAME:
            LXI     H,FCB1
            LXI     D,FCB2
            LXI     B,9
            LDIR                    ; move the name
            XCHG
            MVI     M,'D'
            INX     H
            MVI     M,'B'           ; set type
            INX     H
            MVI     M,' '
            RET
```

```
; messages

USE$MSG:
            DB      CR,LF
            DB      'Usage: TXT2ASM <infile> [<outfile>]',CR,LF
            DB      'Converts standard text input file to DB format'
            DB      'for assembly',CR,LF
            DB      'programs. <infile> is input text, optional  '
            DB      '<outfile>is the', CR,LF
            DB      'output file. If no output file is specified,  '
            DB      'output is sent',CR,LF
            DB      'to <infile name>.DB
            DB      '[Oct. 9, 1982 VI.1]', CR,LF,CR,LF
            DB      '$'
NO$OPEN:
            DB      bell,'Cannot open input file, check directory
                    and spelling.  '
            DB      CR,LF,'$'
NO$MAKE:
            DB      bell,'Cannot create output file, check space
                    remaining.  '
CRLF$MSG:
            DB      CR,LF,'$'
NO$CLOSE:
            DB      bell,'Cannot close output file, definite
                    problem here.  '
            DB      CR,LF,'$'
NO$WRITE:
            DB      bell,'Cannot complete write to output file.
                    Disk probably full  '
            DB      CR,LF,'$'
NO$MEM:     DB      bell,'insufficient memory for input file'.,cr,ff,'$'
IN$MSG:     DB      '  lines input.  ',cr,lf,'$'
OUT$MSG:
            DB      ' lines output.',cr,lf,'$'
LN$MSK:     DB      TAB,'DB',TAB
MSK$LEN
            EQU     $ - LN$MSK

; byte and address storage

EOF$FLG:    DB      0
FIRST:      DB      0
IN$QUOTE:
            DB      0
IN$PTR:     DW      IN$BUF
OUT$PTR:    DW      OUT$LINE
OUT$DSK:
            DW      OUT$BUF
NUM$OUT:
            DB      0
I$LN$CNT:
            DW      0
O$LN$CNT:
            DW      0

; file control block

OUT$FCB:
            DB      0                       ; drive
            DB      '        '              ; name  (8 spaces)
            DB      '   '                   ; type  (3 spaces)
OUT$FCB$EX:
            DS      30                      ; the rest

; buffers

BUF$SIZE
            EQU     16                      ; 16 sectors for buffer
OUT$LINE:
            DS      80
OUT$BUF:
            DS      128 * BUF$SIZE
O$BUF$TOP
            EQU     $
IN$BUF:     EQU     $
            END
```

Lifeboat handles
Tools for the Professional. . .

Lifeboat is the foremost supplier of professional tools for the software developer. We have the best and the most, including:

- compilers
- interpreters
- linkers
- program editors
- cross-compilers
- emulators
- program generators
- graphics interfaces
- data base managers

and much, much more. . . .We have all the popular standards, plus the rare but indispensables, plus some hot products that nobody else has, including:

Lattice™ C

The 16-bit C compiler that everyone's raving about: faster than the competition, more complete (full implementation of Kernighan and Ritchie) and with a growing product base. . . . Order it with **C Food Smorgasbord**, a subroutine package including screen and i/o utilities, IBM® PC BIOS utilities, and a BCD decimal arithmetic package. A complete development system, and no run-time or license fee . . . .

Halo™

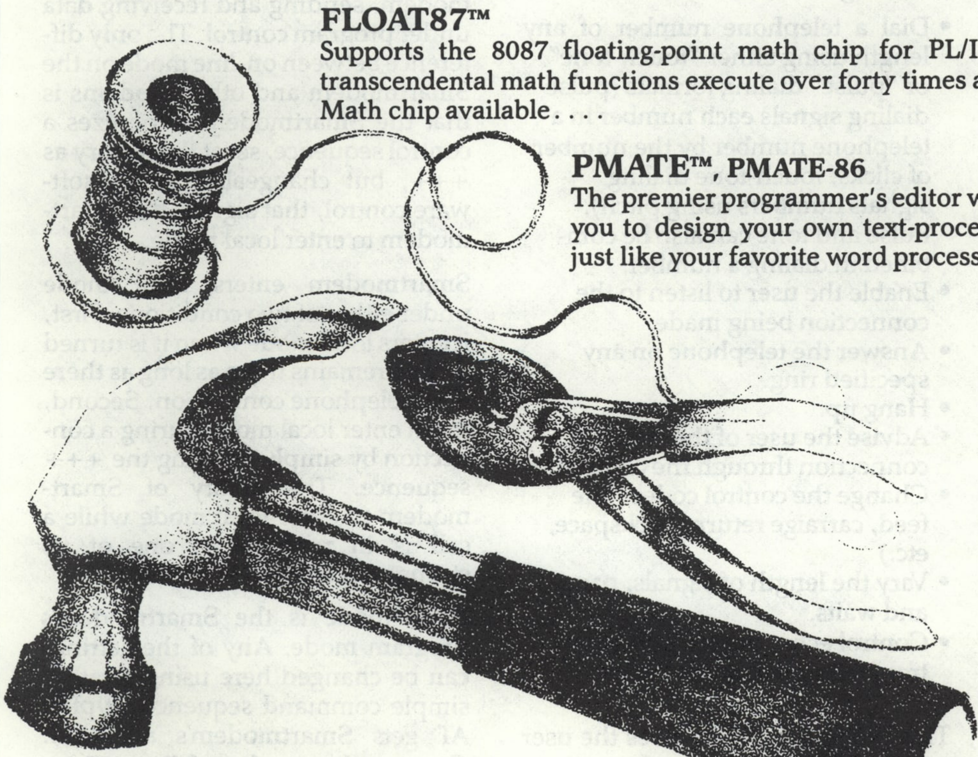
Add color graphics power to your favorite development language. Complete library of graphic primitives with interfaces available for Basic, Basic Compiler, Lattice C, Fortran, Pascal, and Assembler. The emerging graphics standard for PC-compatible systems . . . .

PLINK™-II, PLINK-86

Two-pass linkage editors allow you to create programs larger than available memory using complex overlays. Indispensable for the developer of large applications programs . . . .

FLOAT87™

Supports the 8087 floating-point math chip for PL/I and Lattice C compilers. Real and transcendental math functions execute over forty times as fast with greatly increased accuracy. Math chip available . . . .



PMATE™, PMATE-86

The premier programmer's editor with a built-in macro language enabling you to design your own text-processing functions. Customize it to work just like your favorite word processor, then add more features . . . .

Panel™

Generates commented source code for custom input screens equipped with entry editing, up to nine field attributes, up to 16 display attributes, plus provision to add your own entry validation. Includes terminal definition and multi-key file maintenance systems. Code in PL/I-80, PL/I-86, Pascal MT+, Cobol-80, Cobol-86, Lattice C, MS Pascal, others under development.

Lattice C. . . $500. C Food. . . $150.
HALO. . . . $150 (one language).
PLINK-II. . . $350. PLINK86. . . . $395
Float87. . . $125. PMATE. . . $195.
PMATE86. . . . $225. Panel. . . $350.
8087 chip. . . $225.

Lifeboat Associates
1651 Third Avenue, NY, NY 10028 • (212) 860-0300
TWX: 710-581-2524 (LBSOFT NYK) • Telex: 640693 (LBSOFT NYK)

OEM and dealer inquiries welcome.

Prices and specifications subject to change without notice. Prices F.O.B. New York. Shipping, handling, C.O.D. charges extra. Lattice, TM Lattice, Inc. HALO, TM Media Cybernetics. IBM, reg. TM International Business Machines. MS, TM Microsoft. Programmer's Apprentice, TM PLINK and PMATE, TM Phoenix Software. Panel, TM Roundhill Computer, LTD. FLOAT87, TM Microfloat.

Feature

Hardware as Software: The Hayes Smartmodem

by Davis A. Folger

Sitting somewhere in the never-never land between computer hardware and software sits the everpresent, but generally transparent land of firmware — software that is hardware; hardware that is software. It's everpresent because a microcomputer system will rarely be found that doesn't have some firmware, disguised as ROM, hiding somewhere on the system board. It's transparent because the typical user rarely has to think about it. ROM carries the essentials — a cassette operating system, a disc boot routine, some machine language I/O, and, more often than not, at least a little bit of BASIC.

Except for BASIC, where it is implemented, in whole or in part, in ROM, this firmware is the machine's software not the user's, and while the user's software may address the ROM for various I/O routines, the user doesn't. This tends to be even more true of ROM driven computer peripherals like modems. The Hayes Stack Smartmodem provides something of an exception to this general rule.

The Hayes Smartmodem is a $279, Bell 103 compatible, direct connect, auto-dial, auto-answer modem that operates at 300 baud (300 bits per second or roughly 300 words per minute, take your choice). It also comes in a 1200 baud version that sells for $649. None of these descriptors is particularly unique. There are several modems on the market that fit this general description.

The only thing that really makes the Hayes Stack Smartmodem at all unique is its ROM, its built-in software. The Smartmodem is more than a piece of hardware that attaches to your computer through a serial port. It is a piece of software — indeed, an entire communications programming language — that can be used by the microcommunicator in the same way that BASIC might be. Indeed, the true flexibility of the Smartmodem is probably best expressed in the ease with which a BASIC program can address the Smartmodem. From the user's perspective, the Hayes Smartmodem can be addressed in two ways. First, it can be addressed directly through the keyboard when using virtually any asynchronous microcommunications software. Second, it can be addressed directly by the program itself. From a programming perspective, one follows the other. Almost anything that can be done through the keyboard can also be done from within a program.

The value of the Hayes Smartmodem's programmability is in the things it allows the user to do without taking apart the modem and resetting switches. Among the things the Smartmodem can do under program control are the following:

- Dial a telephone number of any length using either "touch-tone" or "pulse" dialing formats (pulse dialing signals each number in a telephone number by the number of clicks; touch-tone dialing signals numbers using pitch). Pulse and tone can also be combined in dialing a number.
- Enable the user to listen to the connection being made.
- Answer the telephone on any specified ring.
- Hang up.
- Advise the user of the status of a connection through the computer.
- Change the control codes (line feed, carraige return, backspace, etc.)
- Vary the length of signals, pauses and waits.
- Control echo, duplex and other transmission characteristics.

This programmability gives the user an unusual level of control over communications. It simplifies the task of writing high-powered microcommunications software packages. It opens the door to microcommunications software that has more user friendly features. It extends the range of things that can be done with the microcomputer. Some of the applications of the Smartmodem have little to do with microcommunications.

It can, for instance, be used as the major hardware component of an automatic telephone dialer and accounting system. Such systems are highly useful to lawyers and other professionals who must bill calls back to clients. It can also be used as the telephone control unit of a home-built telephone answering machine.

The primary application of the Smartmodem is, however, as a "smart" modem that can be used flexibly in a variety of microcommunications tasks. The key to the Smartmodem's flexibility is in its two modes. In on-line mode, the Smartmodem acts just like any other modem, sending and receiving data under program control. The only difference between on-line mode on the Smartmodem and other modems is that the Smartmodem recognizes a control sequence, set at the factory as +++, but changeable under software control, that signals the Smartmodem to enter local mode.

Smartmodem enters local mode under either of two conditions. First, it enters local mode when it is turned on and remains there as long as there is no telephone connection. Second, it can enter local mode during a connection by simply entering the +++ sequence. The ability of Smartmodem to enter local mode while a call is in progress is one of its strongest features.

Local mode is the Smartmodem's program mode. Any of the settings can be changed here using a rather simple command sequence. Typing AT gets Smartmodem's attention. Commands can then follow. There are more than fifteen of these commands, most of which are complicated by additional parameters. The sequence "AT D P 6558931 ,,,, T 6037895645;", for instance, would dial a telephone number (D) using (continued on page 36)

Feature

What To Do At The A►, Which Your Dealer Never Told You

(or, Backing Up Disks For The Beginner)

by Al Bloch

In the two years I've been fielding software support calls, by far the most prevalent scenario has been something like the following: you finally decided to take the leap and invest in a personal computer, and it arrived in fine working order, but with no instructions other than the notoriously cryptic Digital Research documentation on the CP/M operating system. Or, your tried and true familiar office Word Processor has just been converted to run CP/M programs as well as the dedicated packages for which you originally bought it. In either case, you may have penetrated the maze as far as inserting the distribution diskette containing your CP/M operating system and utilities, properly oriented (which generally means with the read oval, which you NEVER, NEVER touch on either side, inserted first, label last, with the **label towards the drive door** — not necessarily towards the operator! My boss loves to observe that "there are eight ways to put a disk into a drive, only one of which is interesting") and in the correct drive (almost all machines in the market will "boot," or bring up the system, from only the drive labelled A — or perhaps 0 or 1?). Now, you're staring at the enigmatic "A>" which follows the sign-on message.

What next? Which page of the manual to turn to? How to start playing with the juicy application package you bought the machine to run? There are user-friendly programs written to make this phase of operation easier, many of which have been reviewed recently in *Lifelines*, but experience proves that ordinary mortals with adequate instruction can learn all that they need to know to get the machine and software up in less than half an hour, simply by using a few of the tools which came with your system.

The author of the CP/M operating system (which, although not known for friendliness, still bears considerable responsibility for the existence of the industry), and the programmer who configured it for your particular microcomputer, threw in some essential utility programs along with the system itself. Many of them the non-programming user of the 1980s will never use, other than to wonder what practicality might lurk behind such names as DDT or XSUB; but a nodding acquaintance with a few of them will confer the necessary competence and rewarding confidence in handling disk files which will transform the neophyte's fear of the machine into an infectious and addictive delight. You can begin to get acquainted with these new friends by name by typing **dir** (upper or lower case, it makes no difference) after the A>, and sending the command by hitting the ENTER or CARRIAGE RETURN key on the right side of the keyboard (hereafter indicated by **<cr>**), to get a DIRectory of the files on the disk which came with the machine. You will notice that generally they show both a first and last name (they all have a "middle name" consisting of a period (.), but this is not displayed in the DIRectory — we will learn how to use it later), and that the most common "last name" is COM. These files may be thought of as COMmand programs, sitting there on disk ready to do whatever they know how to do when you call them by their first name. We will use just a handful of them to get you started doing what you really want to do, use the machine.

For security purposes, the first step is to 'back up' whatever disks you've received, whether CP/M system, programming languages, or applications packages. In contrast with lesser game machines, NO software available under CP/M (at least as far as I've heard) comes copy-protected; on the contrary, you are expected to make copies immediately (only enough for proper use on your own machine, please, in accord with the license agreement, unless of course we're talking about public domain software), and then to lock up the original distribution disks in some safe place against the day when you'll need an update to a newer version. Almost all vendors will require the original to be returned to qualify for that important service, to prove ownership; the authors are understandably paranoid about piracy!

The goal of the present exercise is to create with your own hands a disk or disks containing a verified, clean copy of whatever software is of immediate interest, along with the support programs which it requires — perhaps a language file, certainly the operating system which relates it to the machine (so as to avoid having to switch disks between boot-up and operation, which is a distasteful procedure to CP/M). I'll assume that your machine has at least two disk drives, and that you are equipped with a moderate supply of blank disks (without which it is just a conversation piece, like a typewriter without paper). Unless you're using the DEC Rainbow or an NBI or Dictaphone word processor (which require you to buy preformatted disks from the company), our first step is to write onto any disk we'll be using the proper pattern which allows your particular machine to recognize the disk as its own. (There are at least a hundred mutually incompatible disk formats in the 5¼-inch realm, and a handful in the eight-inch, one of which — single-density, single-sided, soft-sectored — is the industry standard exchange format used by over a hundred different microcomputers.) Anyhow, our first step will be to type:

A>FORMAT<cr>

thus calling the program listed on the DIRectory as FORMAT.COM. (If you're on a Xerox computer, the file is called INIT instead; on the Wang machines it is called INITDISK, and subsumes the next command, SYSGEN, as well. OKI calls if FDDUTY, and it works differently.)

FORMAT will sign on and ask you a

(continued on next page)

question or two; don't be scared to play with various responses. (You can almost always back out of whatever you're into over your head by typing **CONTROL-C** (henceforth indicated by **↑C**), using the CONTROL key as you would a shift key to get upper-case letters, once you've located that important key; it's usually on the left of the keyboard, and several recent machines make it a different color. Wang calls it GL, many others call it the CODE key.) FORMAT will at least ask you to identify in **which drive** the blank disk is to be found which we want to FORMAT; generally it's quite all right to insert it into B. (If you get into trouble later, with scary error messages such as 'BDOS Error on A: Bad Sector', you might want to come back to this point and FORMAT disks in A; if things work better this way, it will imply that your two different drives are aligned sufficiently differently that they can't read the same disk, and it's time to get them both aligned — this is routine maintenance for all machines, somewhat like changing oil and spark-plugs in your car.) FORMAT generally has a repeat loop built into it, so that you can FORMAT several disks without reloading the program; so don't be frustrated if it asks you the same question again after you think you've succeeded. If the program asks about **which density** or **how many sides of the disk** you want to FORMAT, it's generally all right to go for the maximum allowed, IF the blank disks you're using are rated for it. If FORMAT complains about being incapable of finishing a particular disk satisfactorily, try once more, reseating the disk carefully in the drive, and then reject the disk; if that happens several times in a box of disks, take them back to the dealer and request another brand. (I've heard nothing but good about brands such as BASF, Dysan, and 3-M Scotch, but there are probably several other equally reliable brands.)

After we've FORMATted as many disks as we wish, we can generally get back to the A> (which already looks less formidable, doesn't it? It simply means YOUR TURN) by hitting the RETURN key (<cr>). Our next step will be to write a copy of your operating system (the program which allows all the other programs to get along with your machine) to one or more of the disks we've FORMATted, using a utility which is almost universally known as

A>**SYSGEN**<cr>

(Hewlett-Packard users won't find one; they will accomplish the same thing by using COPY with the SYSTEM option instead of ALL, and please be sure to set VERIFY to YES!) SYSGEN will ask you the **source drive name,** which is almost always A (since only the disk in drive A presently contains the system), and the **destination drive name,** usually B; it will ask you to hit <cr> after each of the above entries, to allow you the opportunity to change disks in the drives if desired until the proper one is in each. (If you decide to FORMAT in A, you should use A also as the destination drive, for the same reason.) SYSGEN also has a repeat loop built in, so that you can write the same copy of the system onto several disks at a sitting. Again, don't panic when you are asked again for the destination drive name; just give it a <cr> when you've run out of FORMATted disks, to return to the A>. (Generally it costs you nothing in disk space overhead to write the system onto any disk you will be using — on most disk formats it sits on the outer two tracks of the disk, which are reserved for it so that it's not competing for precious file space — and it adds to your operation the flexibility of being able to "boot up" from any disk which happens to be in the A drive.)

At this point, if you type **DIR B:** to get a DIRectory of the files on the B disk, you will see NO FILES on almost all machines (the Wang, among a few others, writes a system file into the directory); the implication is that the operating system is not a directory file, and the corollary is that you cannot discern which disks contain it by looking at their DIRectories; the standard way is to put the disk in question into the A drive, reset the machine, and see if it boots.

Now to put some good stuff on the disk which we've FORMATted and SYSGENned; for this we'll utilize the third and last of the CP/M utilities you really need to learn in this session,

A>**PIP**

(The Cromemco system CDOS calls it XFER. There are compelling but complex reasons for preferring it to COPY, which most machines offer as well, unless there's no alternative.) First off, let's back up your distribution system diskette; it contains a good number of utilities you won't be using frequently, but it's still a good idea to have some good copies around, to preserve the original. Here's where it gets a bit cryptic for a little while; but after a few minutes of using PIP, you will feel less intimidated by its syntax and options. Thousands of users have picked it up with minimal trauma. The command which we will now type at the A> is

PIP B: = A:* . *[VO<cr>

(or PIP /V B: = A:* . *<cr> if you're using the supercharged CP/M known as SB-80; the major difference of interest is where the verify option comes), and it's not as bad as it looks. For a start, PIP is totally insensitive (at this stage) to case; upper or lower will do equally well, although file names are generally displayed by DIR in upper case. Next, notice that there is precisely ONE space in the command line, between PIP and B (two in SB-80, around the /V); more or less can screw things up. What this command means is, "instruct the PIP command to create on the **B:** disk (the destination in this case) a copy of every file on the **A:** (the source) disk, and verify each file for accuracy." The * . * part uses the asterisk as a "wild card," to request transfer of 'all files with any first name, and all files with any last name', which boils down to all the files on the disk; you may think of the period as the mandatory middle initial for every file name, although it doesn't appear in a DIR listing. The **[VO** part (with or without a closing] — it makes no difference in this case) requires verification of all files and thorough handling of object code files, which on some machines could be truncated by PIP in the absence of the O behind the bracket; in any case it's a good habit to get into, since **you never care enough to move a file around without being concerned that it get there in one piece.** (SB-80 requires no O option, just /V.)

This command will start a series of file moves and verifies; each file name will appear on the screen as it is being handled, so that if there's any trouble reading a particular file you'll know which one needs attention. PIP will complain with an error message (see below) if necessary, after retrying a troublesome spot ten times; if no trouble arises, our familiar A> will appear at the end of the list after the last file is successfully PIPped.

Making working disks

But everything we've accomplished so far is routine housekeeping; the real fun stuff is yet to come. The payoff is when you can bring up the application you want to run off of a disk which you've made yourself. The application disk will still need to have been FORMATted and SYSGENned, and we will still use PIP to put it together, but we will be more selective as to which files we carry over; no need to fill up the disk with rarely-used system utilities. In this case, we'll want to get inside PIP. Let's insert another prepared disk in drive B, and announce its presence to the system by **control-C**; now, if you type at the A> **PIP**<cr> alone, you'll get an asterisk prompt at the beginning of the next line on the screen, after which everything proceeds as before with the omission of the space which followed PIP in the one-line form of the instruction. Let's tell the asterisk, ***B:=A:PIP.COM[VO** (**!/V B:=A:PIP.COM** if under SB-80), which uses PIP to move a verified copy of itself onto the B: disk; and after that's through and you get another * (or !), **B:=A:STAT.COM[VO**, which moves over the other CP/M utility you'll be using every hour; PIP moves things, STAT shows you how big they are (among other uses). (On the Lanier, you'll want to copy FMTSEL.COM as well.) At the next prompt, a <cr> will get you back home to the A>.

Now, a **DIR B:** will indicate that both PIP.COM and STAT.COM have in fact been added to the disk in drive B, which we are building up to be your application disk. (Remember that it was first FORMATted, and then received a copy of the operating system via SYSGEN.) Let's remove the original or back-up system disk we've been using in A, put it away safely somewhere, and (here comes the tricky part!) **insert the disk from B into drive A**; we'll be booting off of it from now on. You should also **insert into drive B the original distribution disk** of whatever application package (or at least one of them) you want to install on the working disk, and then **reset the machine.** This is frequently accomplished by pressing a black button on the back (or side) of the box, or perhaps a special combination of keys from the keyboard; a few machines (including the Wangwriter, NBI and A. B. Dick) require you to turn the power off and on again (usually after a cooling off period of ten to 15 seconds) to accomplish this function, known in computerese jargon as a cold boot. (If this is your case, please learn immediately to NEVER, NEVER turn the power switch or key on or off with disks engaged in any drive; at least open the drive door to retract the write heads first, to avoid the transient power surge from the switch which is notorious for zapping disk files! Of course, you'll want to close at least the A drive door fairly promptly after turning the power back on, so that the system can be booted from that disk.) This is a case in which CP/M is designed to take care of you perhaps beyond your wishes. If you try to write to a disk which was not in that drive at the time of the most recent boot-up, you will usually be treated to the impressive message, 'BDOS Error on A (or B): R/O', which tells you that CP/M has decided to label that disk Read-Only (even though it is not write-protected physically), since it is not the one it expected to find there. (This second, less radical reset function could be performed equally well with a 'warm boot', the escape-hatch CONTROL-C mentioned above under FORMAT.)

Anyhow, if we type A>**DIR** at this point, we should see only PIP.COM and STAT.COM on the A: disk, which is as expected; **DIR B:** will display the DIRectory of the files on whatever disk you have in drive B. Presuming that the A disk has enough room for all of them, you can copy them all with full verification by typing

A>PIP A:=B:* . *[VO

(or PIP /V A:=B:* . * if under SB-80); note that this time, in contrast to the full system distribution disk back-up defined above, A is the destination drive, and B is the source. Once more PIP will list on the screen each file being moved, and the A> will appear when and if the disk is completely copied. An error message after a particular file name at this point might be either 'BDOS Error on B: Bad Sector', indicating that the system is having trouble reading that file on the source drive; in this case you may be able to get through by hitting <cr>, requesting PIP to continue (remember which file it was, though — it may turn out later to be unusable, in which case it should be remade). The other common error is likely to be 'BDOS Error on A: R/O' or 'Write Error', which probably means that you've run out of room on the disk in A. You can confirm this easily by typing A>**STAT**<cr>; if STAT reports 'drive A: 0k', it doesn't mean that everything is okay; it means that drive A has zero k (short for kilobytes, or 1024-character size units — the CP/Mer's dozen) left to write to. If this is the case, the DIRectory of disk A will probably show as its last entry a file with a 'last name' of $$$, indicating a sick or incomplete file; you can clear it off by typing A>**ERA A:* . $$$**<cr>, which will ERAse all files on A: with the $$$ last name. (Some systems use DELete instead of ERAse.)

But let's assume that neither of the above calamities has befallen, and your PIP * . * has run to completion as indicated by the friendly A> at the end of the file list. If (a.) the program you're trying to install came on more than one original distribution disk, and (b.) the disk format you're using on your machine has enough room left on A as indicated by STAT, you may well be able to copy over most (if not all) of another original disk by simply inserting it into drive B and repeating the previous PIP command. In this case no reset is required, since the disk we've changed in the drive is only being read, not written to.

Application software requiring a support language

The final interesting wrinkle in the

(continued on next page)

back-up exercise is the situation in which the package you want to run needs the programming language in which it was written to be present on the disk in order to run; this is usually the case with applications written in either the CBASIC or MBASIC dialects of the BASIC language. One or the other of these languages is frequently given away with recently marketed machines (or are they charging you for the software, and giving away the hardware?); but if what you want to run needs the language and you don't own a copy, you should order it along with the application (NOT mooch a copy from your neighbor, even if his machine reads the same disk format; that could constitute him a pirate, and potentially liable for prosecution!).

In this case you will receive, as well as the application disks, a separate disk containing the language itself; it may be backed up as previously described, but specifically one file from the language disk must be added to our application disk. Here is an opportunity to use one last feature of PIP, that of renaming a file while moving it (although this may also be accomplished by PIPping as above, followed by RENaming newfile.nam=oldfile.nam at the A>).

Let's say you want to run one or more of the early generation Peachtree accounting modules, which some vendors are still unloading at bargain-basement prices, although they have since been rewritten in a more modern version of BASIC. You'll need, in addition to your Peachtree disk, BASIC-80 from Lifeboat (unless somebody else is still providing the earlier version of the language), which contains, in addition to the current version 5.21 of the interpreter (MBASIC.COM on the disk), the classical 1977 version 4.51, listed as OBASIC.COM. Due to the nature of the enhancements to the language between the versions, software written under the earlier version won't run under the later. The hidden 'gotcha' here is that when the software was written, OBASIC was called MBASIC, and under some circumstances the software may come up looking for the language file under that name! (O may be thought of as indicating Old, but M means Microsoft, the author, not Modern.) It is a modest stretch to your rapidly-developing back-up skills to conceive of typing

A>PIP A:MBASIC.COM= B:OBASIC.COM[VO<cr>

which will instruct PIP to pick up the OBASIC file (only) from the B disk, and copy and verify it to drive A under the name MBASIC. Of course, if you're running a current Peachtree release, you'll want to move over MBASIC unrenamed to run it, unless you get the already compiled version of the software, which runs much faster and requires no interpreter support. Many other authors also distribute programs written in MBASIC, which need the language to make them run (unless they're compiled, in which case you may need, from either the author or a vendor, BRUN.COM).

CBASIC applications

The other prominently popular support language is CBASIC, a pseudo-compiler (don't worry now about what that means) famous for its two-letter error messages (such as ERROR OM — the primary reason most users need the manual for the language is to look up error codes, about which they can usually do little, since many more people run programs written in CBASIC than actually do original programming in it). The distribution disk comes with five .COM files on it (three if you are on a new-generation 16-bit machine under CP/M-86 and buy CBASIC-86), only one of which is needed to make the software come alive. Depending upon your version of operating system, the file you want is called

SYSTEM VERSION	CBASIC FILE
Turbo-Dos, SB-80, or CP/M version 2.2 or 3	CRUN238.COM
MUON, CDOS, or CP/M vers. 1.4	CRUN2.COM
(or maybe the earlier	CRUN204P.COM)
CP/M-86 (16-bit only)	CRUN86.COM

However, since all these names are complicated to type correctly, and the language file must be called before the application (imagine asking all your secretaries to type **A>CRUN238 HELLO<cr>** every time they want to run HELLO), it's far simpler to rename the file while moving it over to the application disk, by typing

A>PIP A:RUN.COM=B:CRUN238.COM[VO<cr> or whatever variation thereof may be called for, after which the application may be invoked by typing **A>RUN HELLO<cr>**.

Summary

Step 0.) Never turn machine on/off with disks engaged in drives!

Step 1.) Insert distribution system disk in drive A, blank in B.
- a.) **A>FORMAT<cr>** blank disk(s), answering a few questions.
- b.) **A>SYSGEN<cr>** system onto blank(s); source A, destination B.
- c.) If backing up entire system disk,
  A>PIP B:=A:*.*[VO<cr>
  If creating an application disk,
  A>PIP<cr>
  ***B:=A:PIP.COM[VO <cr>**
  ***B:=A:STAT.COM[VO <cr>**
  ***<cr>** to return to **A>**

Step 2.) Remove distribution system disk from A; insert disk from B into A; insert desired original into B. RESET. **(If language needed, insert it first in B; if CBASIC,**
A>PIP A:RUN.COM= CRUN238.COM[VO <cr>)
A>PIP A:=B:*.* [VO<cr>

Reading the above, followed by a few minutes' experience at the keyboard, will certify you a qualified disk backer-upper; now you're ready for the next challenge, actually trying to run whatever application is your interest. That's another story. . . . ■

YOU SPENT $4,000 ON A PERSONAL COMPUTER. FOR ANOTHER $12.50, YOU CAN GET YOUR MONEY'S WORTH.

Today's personal computers have an extraordinary range of capabilities.

For a variety of reasons, however, many business people are unaware of just how much their computers are capable of.

As a result, they aren't realizing the full potential of their investment.

THE KEY TO GREATER PRODUCTIVITY IN A WORD: SOFTWARE.

Computers do the work. Software does the thinking.

Expanding the amount of work a personal computer can do is merely a matter, then, of gaining access to a broader array of software.

And the software programs available to business and professional people number in the *thousands.*

But where do you go to find them?

THE KEY TO SOFTWARE IN A WORD: *LIST.*

LIST is the first publication that puts software first.

It contains articles by some of the most respected names in the computer field. Written to help you get the most out of your personal computer. No matter what brand it is.

No matter what you need it to do.

More importantly, *LIST* contains the *LIST Software Locator,*™ a comprehensive guide to over 3,000 personal computer programs—conveniently indexed by application, industry, operating system and hardware. You'll find detailed descriptions of applications software that pertains specifically to the type of business you're in. And the type of needs you have.

LIST is sold at leading computer stores and bookstores. Or, you can phone our toll-free number (1-800-821-7700, Ext. 1110) or send in the coupon below, and receive a copy by mail. The price, exclusive of postage and handling, is $12.50.

Which, when you think about it, is a pretty small price to pay for something that can maximize a much larger investment.

LIST *is published by Redgate Publishing Company, an affiliate of E.F. Hutton.*

I'D LIKE TO GET THE MOST OUT OF MY PERSONAL COMPUTER.

Please send me ______ copies of *LIST* at $12.50 a copy plus $2.00 each for postage and handling. (Tax will be added where applicable.)

☐ VISA ☐ MasterCard (Interbank No. ______)

Card No. ______ Exp. Date ______

Signature ______

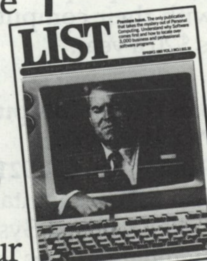
Print Name ______

Address ______

City ______ State ______ Zip ______

Send to *LIST,* Redgate Publishing Co., 3407 Ocean Drive, Vero Beach, FL 32960.

Or phone, toll-free: **1 800 821-7700 Ext. 1110**



MasterCard VISA

LIST™

The Software Resource Book For Personal Computer Users

Users Group Corner

Editors Note: We hope you will write in and give us information about your users group or computer club. Our Users Group Corner is designed to help all of you readers find computer clubs in your area or new clubs that your existing club can exchange information with.

KAPPA
P.O. Box 1563
Gulf Breeze, FL 32561
The KAYPRO Association of Professionals, Programmers and Analysts is starting a exchange program with other groups. They will maintain any KUG on their mailing list from whom they receive a newsletter,and will share their public domain software with any group on an exchange basis or for $5 per disk (to groups only). As of this writing KAPPA has nearly 30 user groups on their list. (This may be a good way to find a KAYPRO Users Group in your area. If you write for information be sure to include a self addressed stamped envelope.)

CJPC
Computer Systems Labs., Inc.
808 Shrewsbury Ave.
Tinton Falls, NJ 07724
The Central Jersey IBM/PC Users Information Exchange has just been formed. Some of their goals include: to serve as a center of information exchange for PC users, to provide an information exchange concerning hardware and software options for the PC, to provide a "Clearing House" of latest new information, and to provide an exchange between experienced and new users for application ideas.

Microcomputer Users International
c/o Jack Decker, newsletter editor
1804 W. 18th St. Lot #155
Sault Ste. Marie, MI 49783
The MUI is a group that serves the microcomputer users of Sault Sainte Marie (USA and Ontario, Canada) area. They are interested in exchanging newsletters with other computer clubs and user groups. The MUI newsletter is entitled Northern Bytes.

TRS-80 Users Group
Gebruikersvereniging Afdeling
Rotterdam, Havikhoek 48,
3201 Spijkenisse, Holland
The TRS-80 Users Group has a goal to stimulate the use of TRS-80s. They have compiled a small library of selfmade software. They are interested in getting closer contact with other groups to exchange: software, hardware layouts, newsletters and anything that can be done together.

ACSCI SWAP
P.O. Box 28606
Columbus, OH 43228-0606
The Amateur Computer Society of Central Ohio has the following users groups: Osborne, ACE (Atari Comp),Timex/Sinclair,COACH (Apple Users), TRS-80 COLOR, TRS-80 Users, Kaypro, Pet Users, CP/M Users, and ROBOTICS Group. Their newsletter is entitled I/O.

PLUMB
PO Box 300
Harrods Creek, KY 40027
PLUMB (probing the world of personal telecommunications) is interested in computer bulletin boards and personal telecommunication. PLUMB tells micro users how they can plug into free software, personal message systems, online gaming systems, and x-rated bulletin boards and services that appeal to people who work at home with their computers. Its newsletter is $20 a year.

San Diego Computer Society
P.O. Box 81537
San Diego, CA 92138
The SDCS maintains a Community Bulletin Board System for use by its members. Dues are $15 per year. Renewal is $10 per year if received before the expiration date.The SDCS special interest groups include the following: MicroComputer Innovators (Z-80, CP/M, PASCAL, S-100), TRS-80, Commodore, Robotics, dBASErs, Kaypro, Exidy, IBM-PC, 68XX, Forth Systems, Osborne, Disabled Interest Group, Personal Investment, Atari, Heath, Texas Instruments, and South Bay Commodore.

CPMUG
1651 Third Ave.
New York, NY 10028
The CP/M Users Group's last five volumes, 86-90, contain a package entitled "Businessmaster II" which includes programs for inventory/fixed asset accounts, mail list, payroll, purchase order/payables, order entry/receivables and general ledger. The programs are written in CBASIC 2. An updated version of Businessmaster II is sold commercially and this older version has been put into the public domain. CPMUG now has KAYPRO format available. Price is the same as North Star and Apple formats.

Software in the library, obtainable exclusively on diskettes, is available for a prepaid media and handling charge, as follows:

FORMAT	DESTINATION
8" IBM	U.S., Canada, Mexico—$13
8" IBM	All other destinations—$17
North Star/Apple	U.S., Canada, Mexico—$18
North Star/Apple	All other destinations—$21

PLEASE CLEARLY SPECIFY THE FORMAT YOU WANT WITH YOUR ORDER

This payment covers the cost of the diskette(s), packaging, and postage. Domestic shipping is via UPS where a full street address is given; all other orders are via U.S. Postal Service.

NAMU
711 W. 14th St.
Austin, TX 78701
The National Association of Microcomputer Users completed the first issue of its newsletter in May 1983. Its dues are $25 a year which entitles you to a free subscription to the newsletter. NAMU's goal is to bring together all the components of the microcomputer industry. ■

Software Notes

Macro Of The Month

by Todd Katz

PMATE, our text editor of choice, is also a structured language interpreter — and a sophisticated one at that! This month the SUPERDIR macro illustrates a small portion of the power inherent in the PMATE language and challenges MICROSOFT's MS-DOS 2.0 in what will go down in history as the great "Disk Directory Sorting Contest."

This month's macro also makes an attempt to combine two longstanding M. of the M. goals: an alphabetized directory and a columnized directory. It addition it makes it possible for you to perform standard I/O operations on disk files without having to go back to PMATE's command line.

SUPERDIR also pays Ron Finley of Technical Services, Co., Harley, Oregon the highest MACRO of the MONTH complement: we have taken his macro and incorporated it into our own.

Mr. Finley writes, "Way back in the January, 1982 issue of *Lifelines* you complained of the slowness of the sorted directory macro in the PMATE manual. As an alternative, you presented a macro to produce an unsorted, four-column directory. I decided that I still wanted a sorted directory macro, and since I have seen only the one in the (PMATE) manual, I wrote another myself."

Indeed, the binary sort in the SUPERDIR program is very fast, and it can be run independently of the rest of the macro. The sort performs a simple function: alphabetizing the directory and listing it on the screen. In addition, wild cards are available so that you could, if you load SUPERDIR into the '.D' permanent macro, type '.D*.COM' on the command line to get an alphabetic listing of all '.COM' files on your logged-in drive.

Mr. Finley mentioned that he was getting error messages when using the wild-card option, however the 1QA command, which limited the number of macro arguments to one, seems to eliminate this problem.

```
;sortdir July 4, 1983 1.0
80f                                  ; set screen for 80 columns
[                                    ; begin loop #1
  [                                  ; begin loop #2
    t                                ; tag cursor
    bkb6kb7k                         ; kill buffers 0,6,7
$0gthinking somewhat slowly$         ; flash message
; this section contributed by Ron Finley
    be1qa                            ; buffer 0 edit
                                     ; #of macro arguments to one
    xL↑AA$                           ; list directory
; NOTE: the ↑AA makes it possible to specify wildcards when you call this macro!
    99qa                             ; set # of arguments to 99
    ab7k$                            ; top of buffer, kill b7
    1v0                              ; set var. 0 to 1
    [                                ; loop No. 3
      @t=0__                         ; if a null is found exit loop
      b7e                            ; edit buffer 7
      1v1                            ; set var. 1 to 1
      @0v2                           ; set var. 2 equal to var. 0
      [                              ; loop No. 4
        @2=@1__                      ; if var. 2 = var. 1 exit
        ((@2-@1)/2)v3                ; subtract var. 1 from var. 2
                                     ; divide by 2 and assign
                                     ; to variable 3
        @1va3                        ; increment var. 3 by var. 1
        a                            ; go to top of directory
        (@3-1)L                      ; subtract 1 from var. 3
                                     ; move that # of lines forward
        @h↑A@0$>0                    ; compare string in buffer 0
                                     ; to string where cursor is
                                     ; if greater than 0
        [                            ; enter loop No. 5
          @2=@3[                     ; IF var. 2 = var. 3
              -1va3]                 ; subtract 1 from var. 3
          @3v2                       ; set var. 2 = var. 3
            ][                       ; ELSE
            @1=@3[                   ; IF var. 1 = var. 3
              va3]                   ; increment var. 3
            @3v1                     ; set var. 1 to var. 3
            ]                        ; end loop #5
      ]                              ; end loop #4
      a                              ; go to top of buffer
      (@1-1)L                        ; move var. 1 - 1 lines
      @h↑A@0$>0[][                   ; compare string in buffer 0
                                     ; to string at cursor
                                     ; IF greater than 0 go on
      L]                             ; else move forward a line
      be                             ; enter buffer 0
      b7n                            ; insert line to buf. 7
      va0                            ; increment buffer 0
    ]                                ; end loop #3
                                     ; end Ron Finley's contribution
    16v1                             ; set var. 1 to 16
    b7e                              ; enter buffer 7
    z                                ; go to end of buffer
    @lv2                             ; set var. 2 = to #
                                     ; of lines in file
    @2<20[5v0]                       ; set var. 0 to 5 if
                                     ; less than 20 lines
    (@2>19) & (@2<40)[               ; if var. 2 is between
                                     ; 20 and 39
    7v0]                             ; set var. 0 to 7
    (@2>39) & (@2<66)[               ; if var. 2 is 40-65
```

(continued on next page)

Although sort times will vary depending on the speed of your computer and the number of items in the directory, we got quite respectable 10-second marks for 30-item directories.

Intriguingly, Mr. Finley adds, "While (my sort) is certainly not the fastest method possible, I have found it to be adequate for my needs. . . ."

Question: What is the fastest method possible?

SUPERDIR does two other things:

1. It displays the directory in a five-column format, thus assuring that all the files on the disk can be seen on one screen. The alphabetical listing is vertically oriented, which we believe to be easier to read than the horizontal alphabetical listings found on so many sorted DIR programs.

2. Having placed SUPERDIR on the screen, a micro menu appears on PMATE's command line that reads: F for Fetch; P, print DIR; I, insert; V, view; X, delete; S, set drive; E, to exit.

To perform any of these operations, detailed below, you just point your mouse or move your cursor to the beginning of the name of the file you want acted upon. Then press the appropriate command key — F,P,I,V,X,S or E. Lower case can be used too.

Briefly, here is what each of these functions do:

F "XFs" the file, logging it into the system. This will only work if you are not already logged onto another file. If you are, the macro will abort without harming the file being edited.

P Sends (XTs) the contents of buffer 6 — which should be the SUPERDIR — to your printer. This is a painless way of typing the contents of a disk on a diskette label.

I Inserts (XI's) a file at the end of the text editing buffer. In this way you can merge several files through the SUPERDIR menu — bearing in mind that you must live within available RAM.

V Allows you to view a file for as long as you wish. Press any key to return to the SUPERDIR and its menu.

X Deletes (XX's) a specified file after making sure that you want the file deleted by insisting that

```
    11v0]                            ; set var. 0 to 11
  @2>65[                             ; if var. 2 is > 65
    18v0]                            ; set var. 0 to 18
  aii$                               ; go to top of buffer
  @0L                                ; move down var. 0 lines
  27iiAL$                            ; insert "$AL" (ESC)
  5[                                 ; do loop #3 five times
    @0[                              ; do loop #4 var. 0 times
                                     ; to format columns
      @1 \                           ; insert var. 1 (indent amt.)
      IQXI$                          ; insert QX I commands
      L-m                            ; go to end of line
      32i27i"Li                      ; insert " $L"
      L                              ; move one line
    ]                                ; end loop #4
    @t=0__                           ; if null exit loop
    27iiaL$                          ; insert "$AL"
    16va1                            ; add 16 to var. 1
  ]                                  ; end loop #3
  a                                  ; go to top of buffer 6
  m                                  ; move
  "<i13i                             ; insert "<" and return
  @0L                                ; move var. 0 lines
  ">i                                ; insert ">" end marker
  bte                                ; enter text buffer
  tf                                 ; tag point and reform
  .7                                 ; execute buffer 7
                                     ; which inserts sorted
                                     ; directory in text buffer
  #                                  ; change tag and cursor
  (@0+2)L                            ; move var. 0 plus 2 lines
  #b6d                               ; append directory to buffer 6
  #                                  ; exchange tag and cursor
  1L                                 ; move one line forward
                                     ; options menu
  [                                  ; begin loop #3
    qbqdqb                           ; ring bell twice
GF for Fetch ; P print DIR ; I insert ; V view ; X delete ; S set drive ; E to exit$
                                     ; place menu on command line
    t                                ; tag position
    es↑S$                            ; search for terminator
    #B9c                             ; copy string to buffer 9
    @k=("S!"s){                      ; if choice is "S" or "s'
      gwhich drive?$                 ; ask question
      @k=("A!"a){                    ; IF answer is 'A'
        xsa}{                        ; log into drive A
        xsb}                         ; ELSE log into drive B
        es>$                         ; search for directory end
      -(@0+1)k                       ; delete var. 0 + 1 lines back
      —                              ; exit menu loop
                                     ; and go to beginning of macro
    }                                ; end "S" loop
    @k=("V!"v){                      ; IF choice is "V" or "v"
      bkbe                           ; empty and enter buffer 0
      xi↑A@9$                        ; insert file whose name
                                     ; is stored in buffer 0
      qra                            ; redraw screen and go to top
      3{qbqd}                        ; ring bell three times
gpress any key after viewing$; flash message
      bte↑                           ; enter text buffer return to menu
    }                                ; end "V" loop
    @k=("X!"x){                      ; IF choice is "X" or "x'
      #                              ; exchange tag and cursor
ghit X to delete$                    ; flash message
      @k="X{                         ; IF "X" begin No. 2 loop
        0gbye-bye!$                  ; flash message
        xx↑A@9$                      ; delete file name in buf. 9
        "xi                          ; insert "x" before file name
↑                                    ; return to command menu
        }                            ; end No. 4 loop
    }                                ; end X loop
    @k=("P!"p){                      ; IF choice is "P" or "p"
        b6e                          ; enter buffer 6
      xt$                            ; print buffer
```

you type a second capital "X."

S Set or select (XS') a drive. Gives you an opportunity to switch PMATE to another logged in drive and does a SUPERDIR on that drive so that menu operations may continue.

E Exit the macro.

You are probably wondering how many days all this will take. Actually, thanks to the Finley binary search, the speed is quite respectable. With a 32-entry directory running on a 8086 MS-DOS system, we were able to get a sorted, alphabetized SUPERDIR in 15 seconds!

How does this compare with a sophisticated operating system like MS-DOS 2.0 on an 8086? Well, getting a sorted, columnized directory listing in MS-DOS requires that the program SORT.EXE be on the disk. Then, using UNIX-like pipes, you type "DIR/W|*SORT*<RETURN>. The /W asks that the directory be placed in four columns so it does not scroll off the screen. I tried this on a 54 entry directory on a Wang PC with MS-DOS 2.0. DOS grinds away for awhile — apparently attempting to sort the items on disk rather than in memory — and comes back about 75

```
            bte↑                              ; enter text buffer
            ↑                                 ; return to menu
            }                                 ; end P loop
      @k=("I!"i){                             ; IF choice is "I" or "i"
0ginserting file$; flash message
            z                                 ; go to end of file
            13i                               ; insert <RETURN>
            xi↑A@9$                           ; insert file help in buf. 9
            qrqd                              ; redraw screen and display
            a                                 ; return to top of buffer
            ↑                                 ; return to menu
      }                                       ; end I loop
      @k=("E!"e){                             ; IF choice is "E" or "e"
            %                                 ; end macro
      }                                       ; end I loop
      @k=("F!"f){                             ; IF choice is "F" or "f"
            xk                                ; clear screen
            Xf↑A@9$                           ; assign primary file to
                                              ; file help in buffer 9
            a                                 ; go to top of buffer
            ff                                ; format file
            %                                 ; exit macro
            }                                 ; end F loop
    ]                                         ; end loop #3
  ]                                           ; end loop #2
]                                             ; end loop #1
```

seconds later with a columnized, sorted directory. Wait a minute — that's not alphabetical order. Oh well, back to the drawing board.

Given the same task PMATE and SUPERDIR.MAC managed to columnize and sort the same 54 item directory on the same computer in about 18 seconds. And sure enough, it was alphabetical.

Incidentally, in a compacted form, SORTDIR occupies only 750 bytes. The highly commented stuff listed here with lots of spaces, tabs and carriage returns takes much longer than 18 seconds to execute. ■

Product Status Reports

New Products

INDEXED RELOCATABLE LIBRARY

Active Software Marketing
1953 E. Apache
Tempe, AZ 85281

This Indexed Relocatable Library allows programs written in CB80 to create and directly access files in dBASE II format. Many of the 37 functions such as SELECT, USE, APPEND, SKIP, COUNT and RECALL are very similar to their dBASE II counterparts while others, such as FLDNAM (which returns the name of a field) and NEWFLD (which changes the name of a field), are powerful extensions to the language. A CB80 program using the dBFILE library can process as many as 10 ".DBF" or ".DBR" files at the same time (the dBASE II limit is 2) and can create and modify structures that are fully compatible with dBASE II.

Price: $295, Manual and Demo: $50

NUTRI-BYTES

Center for Science
in the Public Interest
1755 S. St. NW
Washington DC 20009

This program is developed to encourage proper diet. It includes nutrition and food additive quizzes, a diet analysis, and a food additive data base. It is menu-driven and simple to use. It is intended especially for health clinics, health fairs, doctors' offices, and schools. Nutri-Bytes provides an understanding of the healthfulness of our food supply and some of the corporate and governmental forces that help shape America's diet and food policies. The "Chef Pennypincher" diet analysis asks the user about the types of foods he or she eats and provides a diet score as well as personalized advice based on the answers. Quizzes on food additives and nutrition also provide scores to rate the user's knowledge.

Requirements: CP/M, 64K

New Versions

BUG & μ BUG v3.4 for Z80 only
FLOAT-87 for BASIC-86, LATTICE "C", & PL/I-86 under CP/M-86
Insurance (UNIVAIR 9000) w/ & w/o PAYABLES v2.01
Medical (UNIVAIR 9000) v2.07
Dental (UNIVAIR 9000) v2.07
Legal Time Accounting (UNIVAIR 9000) v2.01
General Ledger (UNIVAIR 9000) v2.0
MAGIC KEYBOARD v 2.0/1.1
MAG/base-1, -2, & -3 v3.01 (compiled under CB-80)
PAS-3 Dental v1.75 (compiled under CB-80)
PLINK-86 v1.25 for MS-DOS

(continued from page 26)
pulse dialing (P) for the first seven numbers (6558931); then pause for 8 seconds (,,,,); then use touch-tone dialing (T) for the last ten numbers (6037895645); then stop dialing (;).

This sequence might be used to dial a long distance call on MCI (which requires touch - tone) when using pulse dial local telephone service (which is generally less expensive than touch-tone service). The eight second pause allows MCI to connect. The final semicolon allows you to pick up the phone and talk without having to worry about a modem screaming in your ear.

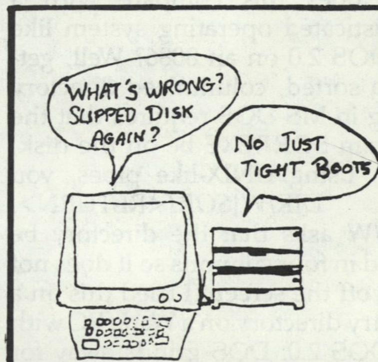
The Hayes Smartmodem is an excellent product that is currently gaining wide acceptance among microcommunicators. Indeed, recent microcommunications software releases for the IBM Personal Computer have almost invariably included some form of direct support for the device. Many of the microcommunications software packages which will be reviewed in coming months, support the smartmodem, and if the trend toward supporting its software features continues, they may soon become something of a software standard which could well be emulated by other modem makers. ■

(continued from page 2)
micros. The higher baud rate is causing many to think seriously of the feasibility of using micros for telecommunications of files of all types in a wide variety of applications.

There is currently more of an opportunity than ever for the independent software developer to make serious inroads into the software market. The availability of machines such as the IBM-PC and its many clones, together with the excellent development tools such as Lattice C, is stimulating a tremendous amount of development effort among the cottage industry.

If you are looking for an excellent way to supplement your income give serious thought to writing applications software, documentation for software already in the marketplace and in widespread use, or books on hardware/software. The larger corporations have brute force at their command but a competent programmer working in his basement can be a very tough competitor.

If you are a frustrated author, try writing articles for computer magazines. Computer publications are in such great abundance that they are all scrambling for articles of virtually any kind. Writing is a good way to achieve recognition in the microcomputer field and they pay you for your words too! ■



dBASE II™ made easy!

QUICKCODE™

The dBASE II Program Generator

Now dBASE II is made easy with Quickcode by Fox & Geller. QUICKCODE is a program generator, a computer program which writes computer programs.

FAST AND SIMPLE

With QUICKCODE you can generate a customer database in 5 minutes. Its that fast. All you have to do is draw your data entry form on the screen. It's that simple!

NO PROGRAMMING REQUIRED

QUICKCODE writes concise programs to set up and maintain any type of database. And the wide range of programs cover everything from printing mailing labels and form letters, to programs that let you select records based on your own requirements. There are even four new data types that are not available with dBASE II alone.

YOUR CONTROL

And since you work directly with your information at your own speed and your own style, you maintain complete control. Telling your computer what to do has never been so easy.

QUICKCODE, by Fox & Geller. Absolutely the most powerful program generator you've ever seen. Definitely the easiest to use.

Ask your dealer for more information on QUICKCODE and all the other exciting new products from Fox & Geller.

FOX&GELLER

Fox & Geller, Inc. Dept. LIF 001 ***604 Market Street Elmwood Park, N.J. 07407 (201) 794-8883***

Lifelines™/ **The Software Magazine**™
1651 Third Ave., New York, New York 10028

Second Class Postage Paid
At Smithtown, N.Y.

EXPIRATION DATE: 12/83
P