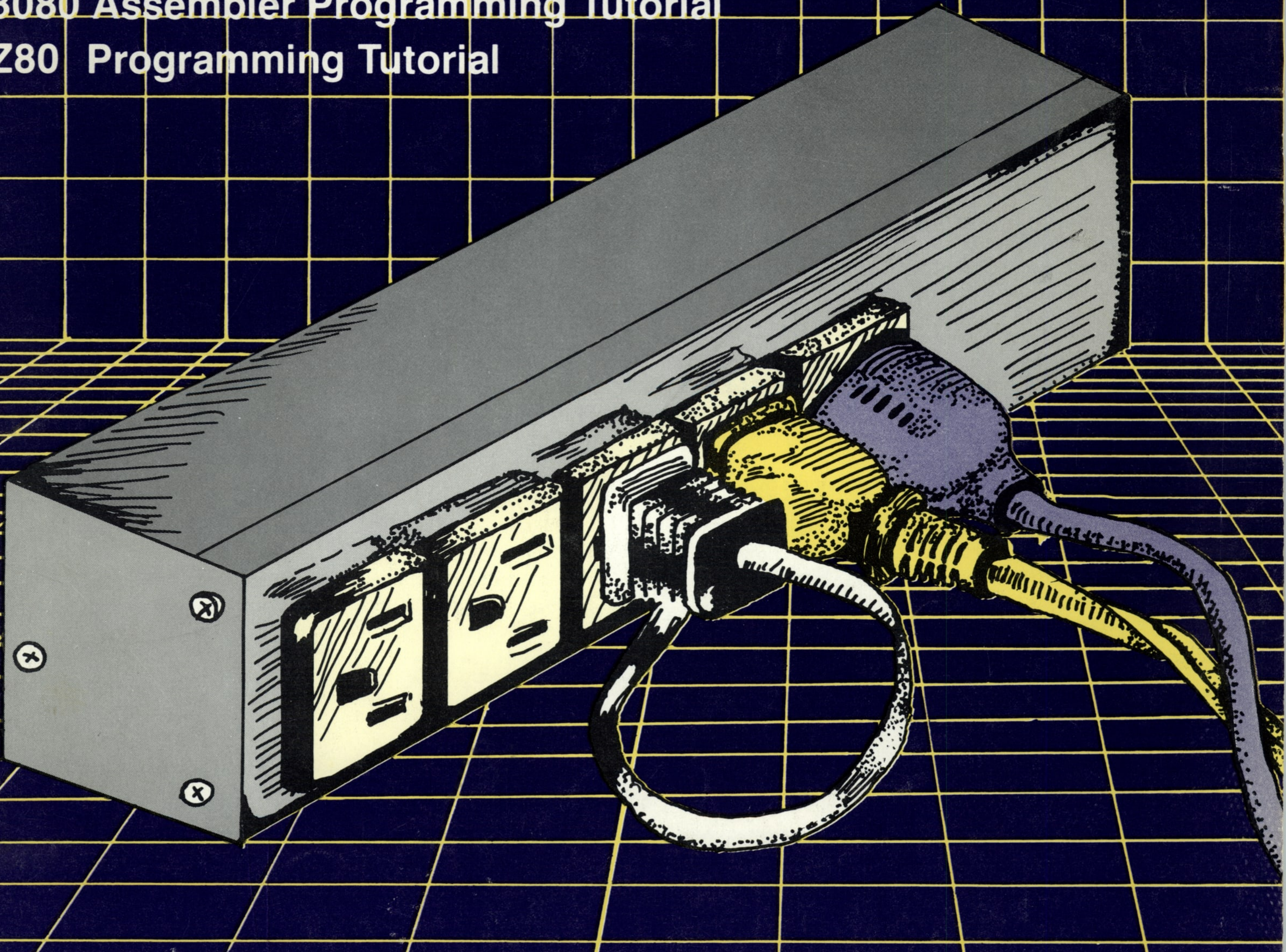
LIFELINES™
The Software Magazine™

$3.00 September 1982 Volume III, No. 4 (ISSN 0279-2575, USPS 597-830)



TIM™ III

The Non-Programming Approach to Data Base Management

Data Base Management

Data management packages were created to save time and money in the development of software solutions to information problems. Many have been designed to accomplish just that, although most have only the programmer in mind. Sure they would save time in the long run, but what of the initial investment in time and effort required to learn the new language? What about the non-programmers in the world who would like an easy yet powerful applications generator? The solution is one of the most highly acclaimed software packages of our time, T.I.M. III.

What is T.I.M.?

T.I.M. is **Total Information Management.** Programmers love it due to its original solutions to classic data management problems. Non-programmers adore it since they can use it to achieve the same results as with other more complicated programming-like packages.

What Makes T.I.M. So Simple to Use?

We at Innovative Software, Inc. designed T.I.M. from day one with the end user in mind. Maybe he is a programmer who doesn't have time to learn a new language. Or perhaps a neophyte who fears coding pads and lines numbered by tens. We felt that a data management package should be able to be used by anyone from a systems analyst to a secretary. That's why T.I.M. takes a full *menu-driven* approach, uses multiple *HELP* screens, and has a manual that sets a new standard in documentation.

The Manual

Many people believe that the manual is just as important as the software itself, a view that we at Innovative Software, Inc. tend to share. The manual for T.I.M. is divided into two sections, the Reference section and the Primer. The Reference section describes all of T.I.M.'s commands and subcommands. This is done in English, not in technical terms or in our own language. Even if you have never seen a computer before in your life, you'll be able to read and understand our manual immediately. The second section is a primer which goes through several examples for you, again in plain English. These true-to-life examples take the beginner by the hand, and instructs him what to do and when. You will be able to see for yourself that T.I.M.'s only limitation is the imagination of the user.

Features of T.I.M.

T.I.M. has all of the features one has come to expect from a data management package, as well as many new ones. For example, a *word processing* interface that allows you to merge information from a T.I.M. file with letters or other documents created by a word processor. Now you can automatically send personalized letters to hundreds or thousands—quickly and easily. T.I.M.'s *Select* command enables you to pull specific information from a file. For example. "All customers who live in a certain ZIP code, whose last name begins with the letter A to L, whose balance due is less than $50.00." A sophisticated *report generator* and even a *list generator* are also included.

How powerful is T.I.M.? With a maximum record size of 2400 characters and the ability to keep up to forty fields sorted properly at all times, T.I.M. is powerful enough to handle just about any application. T.I.M. can handle over 32,000 records per file, and two files can be linked together for reports if your application requires a many-to-one relationship. T.I.M. also includes all of the same editing commands as your word processor, thus making data entry and editing a snap. You can also pull selected records from one file to place them into another. Files may be restructured to add or subtract fields and/or change field lengths or types. T.I.M. even has it's own utility for backing up hard disks onto floppies.

Where to Find T.I.M.

T.I.M. is available from **Lifeboat Associates.** Or you may purchase from us direct by calling 913/383-1089. Either way you will have the finest data management program available.

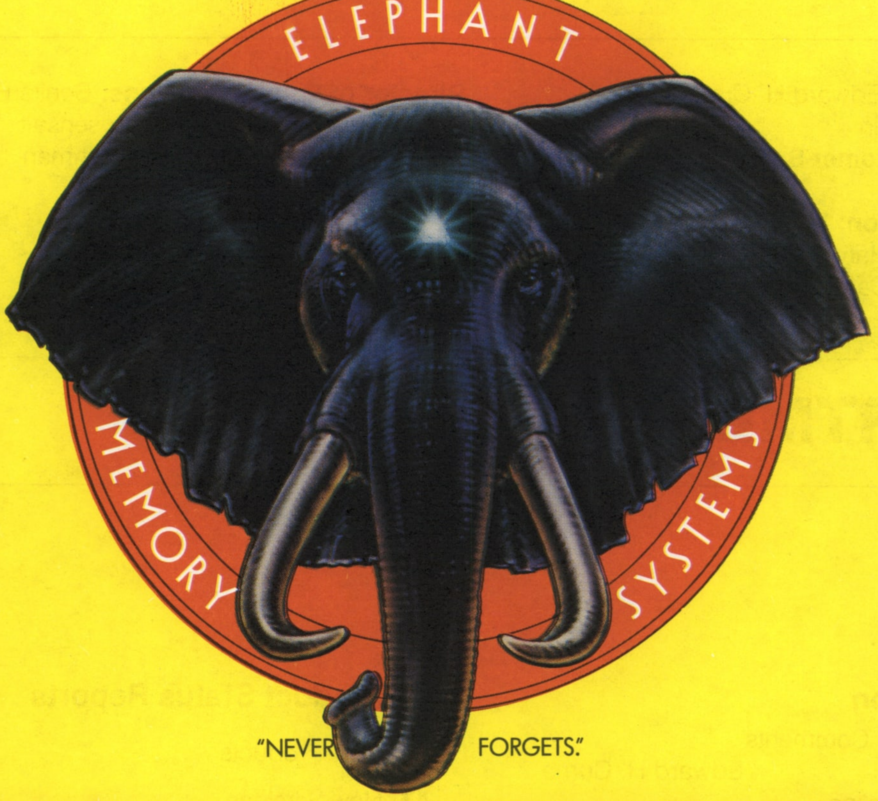
Innovative Software

Available for CP/M,* and IBM PC DOS.**
CP/M version—$695. IBM PC version—$495.

Innovative Software, Inc.
9300 W. 110th Street, Suite 380
Overland Park, Kansas 66210 USA
913/383-1089

TIM is a Trademark of Innovative Software, Inc.
*CP/M and MP/M are Trademarks of Digital Research
**Trademarks of IBM

REMEMBER:



MORE THAN JUST ANOTHER PRETTY FACE.

Says who? Says ANSI.

Specifically, subcommittee X3B8 of the American National Standards Institute (ANSI) says so. The fact is all Elephant™ floppies meet or exceed the specs required to meet or exceed all their standards.

But just who is "subcommittee X3B8" to issue such pronouncements?

They're a group of people representing a large, well-balanced cross section of disciplines—from academia, government agencies, and the computer industry. People from places like IBM, Hewlett-Packard, 3M, Lawrence Livermore Labs, The U.S. Department of Defense, Honeywell and The Association of Computer Programmers and Analysts. In short, it's a bunch of high-caliber nitpickers whose mission, it seems, in order to make better disks for consumers, is also to make life miserable for everyone in the disk-making business.

How? By gathering together periodically (often, one suspects, under the full moon) to concoct more and more rules to increase the quality of flexible disks. Their most recent rule book runs over 20 single-spaced pages—listing, and insisting upon—hundreds upon hundreds of standards a disk must meet in order to be blessed by ANSI. (And thereby be taken seriously by people who take disks seriously.)

In fact, if you'd like a copy of this formidable document, for free, just let us know and we'll send you one. Because once you know what it takes to make an Elephant for ANSI . . .

We think you'll want us to make some Elephants for you.

ELEPHANT.™ HEAVY DUTY DISKS.

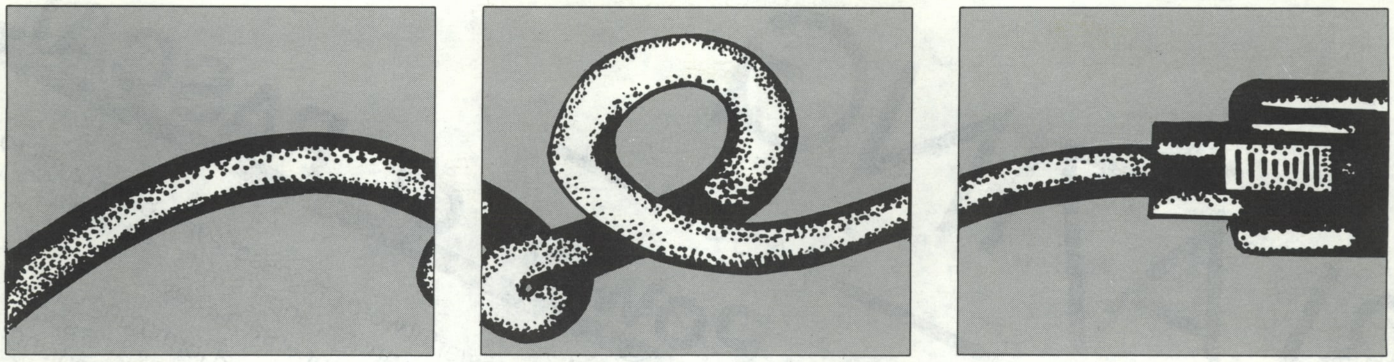
For a free poster-size portrait of our powerful pachyderm, please write us.
Distributed Exclusively by **Leading Edge Products, Inc.**, 225 Turnpike Street, Canton, Massachusetts 02021
Call: toll-free 1-800-343-6833; or in Massachusetts call collect (617) 828-8150. Telex 951-624.

LIFELINES The Software Magazine

September 1982 Volume III, No. 4

Editor-in-Chief: Edward H. Currie
Editor: Jane Mellin
Circulation/Customer Service: Patricia Matthews, Sharon Petro
Design/Production: K. Gartner
Typographers: Harold Black, Carole Mayer
Director of Communications: Bonita E. Taylor
Data Processing: David N. Rosensaft, Lawrence Fishman
Accounting: Bryn Seaberry, Valerie Arthur, Kim Blanchard
Cover by K. Gartner

DEPARTMENTS



FEATURES

BLAZING PASCAL™— Harnessing the Power of PASCAL

Blazing Pascal is state of the art applications software that brings main frame power to the micro. These programs are written in UCSD Pascal*, the most powerful language available for applications programming. If it's more powerful for the programmer, it's more powerful for you. The result? Unparalleled sophistication and ease of use. These programs run on nearly every microcomputer, including the Apple*, the IBM Personal Computer* most 8080/Z80*-based systems, and the new generation of 16-bit microcomputers. They require the UCSD operating system, (the UCSD p-System* or Apple Pascal*). The Blazing Pascal collection meets your needs for business applications, word processing, software tools for the professional programmer, and UCSD-optimized hardware.

The Incredible Text Printer™
This text formatter combines the most powerful word processor available on a micro with unparalleled ease of use. ITP can drive any printer, and it's at its best with a daisy-wheel. ITP's proportional-width fonts will take your breath away.

Format is independent of the text. Routine text requires no added formatting commands to be printed perfectly. Before starting to print, select the margins, line spacing, text font, and other controls. Print the same text in different ways without editing. Save and retrieve up to twelve formats with a keystroke. For specialized printing: Underlining. Boldface. Subscript. Superscript. Alternate colors. Strike-thru. Spectacular tabs and line drawing. Figure and table processing. Automatic column alignment. Automatic section numbering. Automatic index and table of contents. "Mail-merge." Macros.

ITP is available for every popular UCSD system. Full system $249, demo $60, manual $25 (price of demo or manual credited to full system purchase within 60 days).

MINIAC™
MINIAC is a revolutionary software system that speeds business program development by a factor of ten. MINIAC is the first to assist both with the invariant routine of business programming and the countless special circumstances. It consists of a powerful Pascal macro language with added file management, completely under the control of the programmer. Use MINIAC calls to write a skeletal UCSD Pascal program, which is processed by the MINIAC translator and compiled with the Pascal compiler. Common functions are implemented with one or two MINIAC macros, allowing the program source to remain small and easily updated. Macros offer an enormous range of built-in options (including in-line Pascal code) to support demanding requirements.

MINIAC is available for every major system that runs UCSD Pascal. Full program $490, demo $80, manual $30. (Price of demo or manual credited to full system purchase within 60 days.)

213/472•8825

1433 ROSCOMARE ROAD, LOS ANGELES, CA. 90077

DATAMED RESEARCH INCORPORATED

*TRADEMARKS:
Z-80: Zilog Corp.
MINIAC: Shai Microcomputers, Ltd.
Apple & Apple Pascal: Apple Computers.
The Incredible Text Printer: James Gagne.
IBM Personal Computer: International Business Machines Corp.
UCSD Pascal & UCSD p-System: Regents of the University of California.

Go FORTH and Prosper

With Timin FORTH, the unusually fast, elegant and versatile superset of FIG FORTH.

Lifeboat Associates offers this powerful, threaded, interpretive and structured language including a memory resident operating system, text editor, assembler and debugger. This extended FORTH is also enhanced by:

- a visual screen editor with string search and replace
- CP/M-80 file format compatibility
- many additional FORTH words
- array handling (implemented in machine code)
- FORTH assembler for 8080/Z80TM machine instructions
- full floating-point capability

Use this complete interactive software development system to slash software development time and shrink system memory requirements.

For more information about Timin FORTH or any of the other 200+ software packages available for use in professional, personal, and programming environments under SB-80TM or other CP/M® -80 compatible operating systems, IBM PC DOS, or MSTM-DOS (SB-86) contact:

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

SB-80, SB-86, trademarks Lifeboat Associates
MS, trademark Microsoft, Inc.
Z80, trademark Zilog, Inc.
CP/M, registered trademark Digital Research, Inc.
Copyright©1982, by Lifeboat Associates

Copyright © 1982, by Lifelines Publishing Corporation. No portion of this publication may be reproduced without the written permission of the publisher. The single issue price is $3.00 for copies sent to destinations in the U.S., Canada, or Mexico. The single issue price for copies sent to all other countries is $4.30. All checks should be made payable to Lifelines Publishing Corporation. Foreign checks must be in U.S. dollars, drawn on a U.S. bank; checks, money orders, VISA, and MasterCard are acceptable. All orders must be pre-paid. Please send all correspondence to the Publisher at the below address.

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 New York, New York. POSTMASTER, please send changes of address to Lifelines Publishing Corporation, 1651 Third Ave., New York, N.Y. 10028.

ANNOUNCING
THE FOX & GELLER

dBASE II PROGRAM GENERATOR!

QUICKCODE™

Now, without *any* programming, you can create these in seconds:

- * DATA ENTRY PROGRAMS
- * DATA RETRIEVAL PROGRAMS
- * DATA EDIT/VALIDATION PROGRAMS
- * MENUS
- * dBASE FILES

INTRODUCING FOUR *NEW* DATA TYPES:

DATE • DOLLARS • TELEPHONE • SOC. SEC. NO.

With QUICKCODE, you can **have** your program, but you don't have to **write** it. So, you can do things like knocking out an **entire accounting system** over the weekend! And QUICKCODE includes a powerful new version of our popular QUICKSCREEN™ screen builder, so you will put together screens and reports that'll dazzle even the most skeptical (you can even use Wordstar™ to set up your screen layouts).

YOU MUST SEE IT TO BELIEVE IT.

And is QUICKCODE EASY TO USE? You never saw **anything** so easy. You don't have to know how to program. You don't even have to answer a lot of questions, because there **aren't any!**

QUICKCODE $295

ALSO FROM FOX & GELLER

QUICKSCREEN	
Microsoft BASIC version	$149
CBASIC version	149
dBASE-II version	149
dUTIL dBASE utility	75

**Fox & Geller Associates
P.O. Box 1053
Teaneck, NJ 07666 (201) 837-0142**

dBASE-II TM Ashton-Tate
Wordstar TM Micropro Int'l

Opinion
Editorial Comments

Edward H. Currie

Software Publishing – A New Era

Many of you have written and asked about the opportunities for publishing the "intellectual properties", i.e., application programs, which you have written.

As a major software publisher, Lifeboat Associates, like a book publisher, has a standard procedure for submission of "manuscripts". This procedure is detailed in an excellent brochure known as "Guidelines for Software Authors". Those of you who ware interested in obtaining a copy should contact the New Products Department at Lifeboat Associates.

Some of you have also wanted to know exactly what kinds of programs are most desirable to software publishers. There are in fact two basic approaches that an author may use to develop programs of interest.

First, programs which are extensions of applications already available are useful, provided that they are supported by good documentation, reference cards, demonstration programs, etc. One interesting aspect of producing applications of this type is that a "living specification" already exists in the program that you have used as a prototype. This is not to suggest that authors should engage in wholesale imitation or copying of programs, but rather that by studying carefully the programs which are successful, you will have a sound basis for determining what is likely to gain widespread market acceptance. And you may find that the weak points in an already-successful package provide bases for improvement in the software you are developing. If you are following a model in applications design, know your future competition very well indeed. Don't just follow your own judgment; read software reviews and talk with people you know – find out what they do and don't like about the products already out there.

The key to developing good applications is careful definition of the problem, careful design of the solution (i.e. the application package) and good execution. Important to all three of these factors is your knowledge of the end user.

As the microcomputer software market matures, end users are rapidly coming into focus; they are developing a profile which should be kept in mind by applications programmers and software designers. Typically –

a) They don't/won't read detailed documentation during initial stages of learning to use an application package; they are eager to get started and don't see why their screens can't be just as informative as a manual.

b) They object strongly to "rude" (as opposed to the famous "friendly") i.e., abrupt, overly technical, messages which question their intellectual integrity, worth or value as a human being – can you blame them? Don't insult your customers or talk down to them.

c) They love reference cards which summarize the control sequences, etc. and offer enough information to allow them to get started without any other documentation. It's much easier to cope with a few details at a time, in a convenient format.

d) HELP files are very popular, particularly if they may be invoked at any point in a program and do in fact offer relevant information.

e) Demonstration programs, which in fact serve as animated "cartoons" illustrating all of the important features of a program, are extremely useful for sales promotion and training.

f) A few options are better than an infinite number.

g) INSTALL programs with all the standard terminals included should be provided in cases where special terminals require changes in the program.

h) Menu-driven programs are a must, preferably with clear statements of available options.

i) End users are quite likely to want to access files created by a given application program with other applications programs which may or may not have a direct correlation to the particular application which was used to first create the files.

j) Wherever possible, standard escape, control sequences, etc. should be used to make it easy for the end user to move from using one application to another.

All of these tips really point to one central fact: you must put yourself in the end user's place and remember that he or she is primarily interested in getting from point A to point B (the task at hand) in the speediest, most efficient, most painless fashion possible. To make a rather crude comparison, the end user is no more interested in wading through incomprehensible documentation than you are in reading instructions on operating a household gadget; like you, they just want to get the job done.

Work on your attitude. If you discover that you just can't approach your own software in an objective way, find someone who really is naive about computers and software, maybe even someone who typifies your "target audience". Watch them use your product, and learn to put yourself in the end user's place.

Put considerable thought and effort into those things which assure that the first time user has the full benefit of your best talents to insure that they quickly and painlessly become conversant with your application.

It should be like hang gliding — just lift your feet and you're airborne, and if you don't cause any problems you will soar like a bird . . .

Opinion Pipeline

Ann Rogers

Understanding terminal interfacing requirements facilitates software- and hardware-design tasks

Ubiquitous throughout the data-processing industry, terminals are used in all CPU-based systems, whether at the software- and hardware-development stages, in the end-user system, or both. Thus, software and hardware designers must have a thorough understanding of how their CPU-based equipment - be it a host computer, process controller or consumer-oriented intelligent home-entertainment system - might interface with various terminals. Although this article cannot furnish descriptions of the detailed interfacing aspects of all available terminals (more comprehensive information is included in the book *A Designer's Guide to CRT Terminals*, scheduled for publication during 1983 by Reston Publishing Company, Inc., Reston, VA), it illustrates, through specific examples, typical capabilities of representative terminals; the article aims at helping the software designer learn in general what to expect and know what questions to ask of terminal vendors. (In this article, the terminals under consideration include standard typewriter keyboards, a CRT display and a standard electrical interface.)

Keep in mind, too, that interfacibility is important whether or not a terminal is envisioned in the end product. For even if a product is to interface ultimately with a simple ten-button custom keypad or a complex voice-input/output module (rather than a standard CRT terminal and a built-in QUERTY keyboard), a standard terminal probably will be used to mimic the pushbutton keypad or voice-I/O module early in the design stage. Furthermore, such interfacing ease can prove invaluable for future software upgrades, in which the ability to plug a product's RS-232 or RS-449 interface into a software-development system's intelligent terminal can greatly simplify program revision and testing. Moreover, when products will interface with terminals in the end-user system, a thorough understanding of general interfacing requirements, and knowledge of the differences between the control requirements of various manufacturers' terminal types, will do more than facilitate selection of the optimal terminal for a particular application; this knowledge will also aid in modifying the equipment to operate with terminals *other* than the one selected as standard for the product, so the seller can be more responsive to customer requests for compatibility with specific terminals.

This adaptability can take several forms: it can require system modification via PROM changeout, or it can be nothing more than a customer flipping a switch or changing a few jumpers, if the ability to stock large quantitites of a single unit adaptable to many different terminals is important. In this case, a higher price must be exchanged for greater flexibility.

Before looking at methods of increasing the adaptability of CPU-based products, consider a terminal's function vis-à-vis a computer. A terminal basically accepts input one character at a time, usually displaying this information on a CRT. It then transmits this information over some standard interface (usually RS-232C) to a computer, either a character at a time, a line at a time, or a screen or more at a time. Terminals permitting transmission of more than one character at a time require on-board RAM storage sufficient to hold the maximum batch-size of data; practically all terminals include enough RAM for storage of at least one screen of data, whether or not they're intended for use in non-interactive applications. (Terminals without enough on-board RAM to store the number of characters that their screens can display can make use of an external CPU's main memory, which must in such situations include a section for storing displayed data. The terminal can then access this data through such techniques as direct memory access; the cost is additional hardware complexity, as well as increased microprocessor overhead - instruction cycles dedicated to display control rather than the main data-processing task.

Providing an example of on-board memory capability, Zentec's Zephyr/Zms-35 terminal includes 3920 bytes of displayable RAM - enough to handle two screens full of information, at 80 characters per line by 24 lines per screen, plus one message line.

In addition to its transmission ability, a terminal accepts information from a computer over the same interface and at the same rate for display on the CRT. Indeed, a terminal's transmission, reception and display capabilities usually must work together in a typical system's full-duplex single-character-transmission mode. In this type of operation, each character entered on the terminal's keyboard is immediately transmitted by the terminal to the host computer, which in turn retransmits the character back to the terminal for display on the CRT. (Note that no direct connection exists in this mode between the terminal's keyboard and display.) Although such single-character-transmission techniques drastically reduce terminal memory requirements, the host computer must be involved in processing each keystroke; thus, such techniques are often described as interactive. In contrast, terminal page and batch modes permit simultaneous transmission of one or more screens of data at a time, allowing an operator to review, edit and correct data entry before commanding host-computer time. (Such data transmission is, of course, actually serial in most cases - it typically takes place over an RS-232C port - rather than parallel as implied by the term "simultaneous." Nevertheless, most terminal users' manuals employ the term "simultaneous" to describe any data transmissions taking place at speeds dictated by the computer/terminal-interface performance - typically 300 to 9600 baud - rather than by human operator response.)

Available terminal configurations range from dumb terminals, which simply accept and transmit operator inputs and accept and display computer outputs, to intelligent terminals, which to some extent process operator inputs and computer outputs via on-board microprocessors - the Zentec Zephyr,

(continued next page)

for example, includes an Intel 8085A processor. Such terminals thus are able to "help out" the host computer – they support distributed-processing tasks. Whatever the terminal type, however, the software designer must be aware of each terminal's setup and control features and requirements to assure optimum interfacing with computer equipment. These aspects range from rear-panel- and printed-circuit-board-mounted switches that permit control of various terminal functions, to requirements for special remote-control codes that permit computer control of terminal operations.

Don't overlook the mode-control capabilities of many terminals that let an operator select terminal conditions via the unit's keyboard. The terminal section of Heath/Zenith's H/Z-89, for example, lets an operator select such functions as reverse video (ESCp), return to normal video (ESCq), keypad shifting (ESCt), in which shifted (upper-case) characters can be entered without pressing the shift key (in this mode, the shift key is required to generate lower-case characters), and automatic line feed on receipt of a carriage return (EXCx9).

In addition to using such keyboard commands, you can adapt main terminal features to your application through nothing more complicated than the proper resetting of rear-panel- or printed-circuit-board-mounted switches, obviating the need for any software modifications in your product. Don't for example, automatically include specific software routines which allow for connection to terminals furnishing only odd or even parity checks – the various terminals that you might reasonably expect your end product to ultimately interface with probably allow you to select either scheme via switches, or indeed, they might allow you to select mark or space alternatives to the parity bit. (In the former, the parity is always one, regardless of the data; in the latter, the parity bit is always zero.)

As an indication of the flexibility that such simple switch selections can provide, consider, for example, the switch-selectable functions of Applied Digital Data Systems Inc.'s Regent 20 terminal. The standard video presentation of that unit is dark characters on a light background, for which a rear-panel switch designated Switch 1 is set at 0. By setting that switch to 1, you can reverse the presentation.

Switch 2 determines whether the keyboard operates in full- or half-duplex mode. In the full-duplex mode, in which Switch 2 is positioned at 0, the keyboard simply acts as a code generator, sending data to the communication line. It is not directly connected to the display electronics, although echoed data received via the communication line does go directly to the display. Thus, for the operator to see the data from the keyboard in the full-duplex mode, the communication line must echo each character, and the host computer must operate in the interactive mode previously discussed, processing each character when entered by the terminal operator. In the half-duplex mode, with Switch 2 set at 0, keyed data is displayed on the screen as it is transmitted to the communication line. Here, the host computer must still process each character when entered but does not echo characters to the display.

Rear-panel Switches 3 and 4 select the type of parity check employed by the terminal – odd, even, marking or spacing. If odd or even parity is chosen, the terminal appends the appropriate parity bit (bit 8, whose state is based on the parity of the seven ASCII data bits) to outgoing data. (For even parity, the parity bit is set to make the total number of "1" bits in the byte under test [the seven-bit ASCII character plus the parity bit] an even number; the total number of "1" bits in the byte becomes an odd number for odd parity.) The terminal makes use of this feature to check the parity of all incoming data. If the unit's parity-check feature is enabled (via printed-circuit-board-mounted switches to be discussed later), an error in incoming data causes an asterisk to replace the defective character on the screen. If, on the other hand, marking or spacing parity is chosen, the parity bit on outgoing data is always 1 or always 0, respectively. In these modes, enabling the parity-check function will cause an asterisk to be displayed instead of any character with the parity bit in the wrong state – that is, whose parity bit violates the "always 1" or "always 0" rule. Although such error detection is far from ideal – it can only verify the integrity of the final bit, which is, after all, not even part of the data of interest – the appearance of an asterisk can indicate severe line degradation.

Switch 5 controls Auto Line Feed mode; in position 1, it's enabled, in position 0, disabled. When the Auto Line Feed mode is selected, receipt of a carriage-return code or depression of the New Line key generates an internal line feed within the terminal logic. A carriage-return code thus, for example, causes the display to scroll when the cursor has reached the bottom line and both Auto Line Feed and Auto Scroll are enabled. Note that an internal line feed is completed when the cursor advances beyond the end of a line. Such capabilities eliminate the need for such command combinations in host-computer programs.

Baud-rate settings (110, 150, 300, 1200, 1800, 2400, 4800 or 9600 for the Regent 20) are also back-panel selectable on the Regent 20; this adjustment is accomplished through Switches 6, 7 and 8. A baud rate of 300, for instance, is chosen when Switches 6 and 8 are set at 0 and Switch 7 is at 1.

Users can also select Regent 20 parameters via switches on printed-circuit cards. Parity check, for example, can be enabled via Switch 2 of circuit-board-mounted Switch Block A3. When Switch 2 is on, the terminal indicates detected parity errors on the CRT as an asterisk. When the switch is off, the terminal ignores those errors, and data is displayed just as it is received, regardless of how it was originally entered or transmitted.

Display of the cursor is controlled by Switch 3 of Switch Block A3 and Switch 4 of Switch Block A5. When it's on, the former causes the display of a blinking cursor; in the off position, it provides a steady cursor. Switch 4 of Switch Block A5 determines whether the cursor is a rectangular-block or underline type.

When Switch 4 of Switch Block A3 is off and the keyboard receives or generates the control code termed ESC,5, the keyboard locks and remains locked until the unlock command, ESC,6, is received or generated at the keyboard. BREAK is the only key that can transmit during keyboard lock. It will not, however, unlock the keyboard. When Switch 4 of Switch Block A3 is on, receipt of an EOT code causes the keyboard to lock, and receipt of an STX code, or depression of the BREAK key, unlocks it.

To select among character sets, an operator uses Switches 5, 6, 7 and 8 of Switch Block A3. Switches 5, 6 and 7 determine selection of one of seven international sets, and Switch 8 controls choice of lower- and upper-case characters. The Regent normally powers up in upper-case/shift-to-lower mode. Turning Switch 8 on, however, puts the terminal in the lower-case/shift-to-upper condition suited to such applications as word processing.

The final user-selectable function that the Regent provides via printed-circuit card switches is Auto Scroll mode. When that mode is enabled, data scrolls upward if the cursor is in line 24 (the bottom line) and if the display receives a Line Feed (CONTROL-J) code or if one is generated from the keyboard. If the cursor is at the bottom line and a CR is received, data will also scroll, as it will when the NEW LINE key is depressed and Auto Line Feed is enabled. However, the display does not scroll when the Auto Scroll option is disabled. Instead, a command that attempts to move the cursor down from the bottom line (Cursor Down or Line Feed) places the cursor in the top line.

In addition to allowing control via keyboard commands and switch settings, many terminals permit remote control via a host computer. Zentec's Zephyr, for example, includes 69 remote control codes that handle functions ranging from keyboard locking to insertion of data at the cursor position on the unit's CRT. In such modes, the host typically signals an upcoming remote command via a lead-in code, usually an ASCII character that the terminal won't display; so you must be sure that you need never display the lead-in character required by any terminals you select. The lead-in character is usually the ASCII ESC character (decimal 27, hexadecimal 1B); hence the remote-command codes are often termed escape codes. Some terminals, however, offer you a choice of lead-in characters.

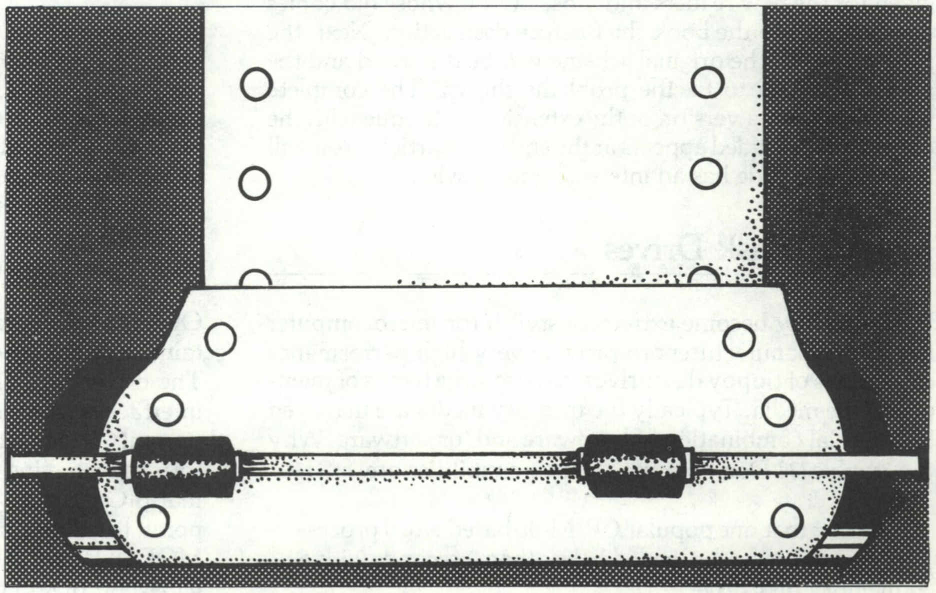
Representative of terminals permitting remote control, Hazeltine's Executive 80 Model 20 allows switch selection of an ASCII – (equivalent to a decimal 126 or hexadecimal 7E) or an ASCII ESC (decimal 27, hexadecimal 1B) as the lead-in code. A valid command parameter must immediately follow the lead-in code; otherwise, the lead-in code is ignored. And not all remote commands require the lead-in code, or the presence or absence of a lead-in code might determine the remote command to be executed.

Among the more important functions of remote-command control are screen clearing and cursor positioning. Hazeltine's Model 20, for instance, clears its screen (or an area determined by selectable roll-up limits) of characters and attributes (such as high/low intensity and forward/reverse video, which are also controllable via remote commands) and moves its cursor to the home (upper left-hand corner as seen by the operator) on receiving a lead-in code followed by a decimal 28 (ASCII FS, hex 1C). Similarly, a decimal 15 clears characters and attributes from the cursor position to the end of the cursor's line, and a decimal 24 clears characters and attributes from the cursor position to the end of the screen. Should you wish to insert characters rather than delete them, your host system can furnish a lead-in code followed by a decimal 26. This sequence moves down by one line all lines from the line including the cursor to the bottom of the screen; the bottom line is lost.

Among other Model 20 remote commands, a lead-in code followed by a decimal 11 or 12 will move the cursor down or up, respectively. A decimal 16 not preceded by a lead-in code moves the cursor one space to the right or, if the cursor is already in the right-hand column, to the left column of the next line. Similarly, an ASCII BS (back space) moves the cursor to the left one space (without destroying any characters already entered) or, if it is in the left column, to the right-hand column of the preceding line.

The Model 20's back-space remote command is illustrative of remote commands not requiring lead-in codes. The unit also furnishes an example of different interpretation of remote-command codes depending on the presence or absence of a lead-in code. In its Auto New Line mode, the Model 20 stores a carriage return preceded by a lead-in code; if the lead-in code is not present, the terminal moves the cursor to the beginning of the next line without storing the carriage-return character.

The remote-command features discussed so far represent only some of the more common functions; other remote-controllable features on Hazeltine's Model 20 range from selecting double-width and double-height characters or 132 character columns (rather than the standard 80) to controlling the unit's auxiliary-port parameters, including baud rate. And Zentec's Zephyr permits remote selection of such functions as full- or half-duplex transmission as well as character replacement, insertion and deletion. The key to successful design is knowing that such functions are available and how to use them. Indeed, the remote-control codes can be as important a design tool as your microprocessor's instruction set.



Feature

A Memory Resident Floppy Disk Program

Michael J. Karas

Several months ago *Lifelines/The Software Magazine* presented my two part article series concerning "A Dynamic BIOS Extension Technique" (January and February 1982); it gave the reader a taste of how to extend the number of logical disk drives connected to an already existing CP/M-80 system. The technique set forth the means of extending the BIOS capabilities with additional disk software drivers - at the expense of stealing some of the available TPA and leaving the CCP resident. My experience has shown me that the technique given has a particularly valuable place in the CP/M-80 systems implementation business, and as such the "ADD-ON MODULE" has been implemented in an amazing variety of configurations from the initial type (presented in the article series) of single density non-deblocking BIOS to 5 1/4 inch Winchester Disk drives with deblocking in the BIOS. The ADD-ON MODULE run time package, the .COM file, can be even auto-loaded at CP/M-80 cold boot time to permit the automatic startup of the additional disk drivers. (See the July 1982 *Lifelines/The Software Magazine* article by Kelly Smith on auto-load techniques).

These additional implementations of ADD-ON MODULES have uncovered a number of strange and unique problems. These problems have been corrected and the new ADD-ON MODULE format has been fully tested on at least five distinctly different computer hardware setups, each with a different CP/M-80 and BIOS implementation. Since the original article series resulted in a great deal of reader interest, I feel that the fully-fixed ADD-ON MODULE format should be placed at the disposal of *Lifelines/The Software Magazine* readers.

First a brief review of the ADD-ON MODULE scheme will be given for the new readers and those of you whose old copies don't make it to the book shelf before destruction. Next, the problems with the original scheme will be discussed and the approach taken to fix the problems shown. The complete source code for a version of the extension technique with the corrections included appears at the end of the article. You will find that the code has an interesting new twist.

Memory Disk Drives

It has recently become extremely stylish for microcomputer hardware manufacturers to provide very high performance emulations of floppy disk drives that use large banks of memory as the media. Typically the memory media are managed by a special combination of hardware and/or software. Why the trouble??? Don't ask me. Some possibilities are:

The fact that one popular CP/M-80 based word processor package can be loaded in less than 3 seconds with a memory disk drive . . .

Or a systems programmer desires an assembly, link, and load process to run 10 to 40 times faster than with a typical floppy disk subsystem . . .

Or the S100 processor board manufacturer feels it justifies the existence of a unique processor board with both 8-bit and 16-bit microprocessors on one board . . .

Or just possibly an S100 memory board manufacturer found that concurrent support of a memory drive software product with the board product did wonderful things for memory board sales.

Whatever the reason, I wondered "How?", "How much faster?", and "Could I do it?". This prompted the neat new twist of an ADD-ON MODULE, one that would make a system-to-system transportable memory drive demonstration program to answer the above questions. Now in this article I would like to share my findings about memory drives in conjunction with the ADD-ON MODULE fixes discussed above.

The ADD-ON MODULE Philosophy

The concept of the ADD-ON MODULE is to extend the normal capabilities of an existing CP/M-80 BIOS so additional drivers can be included for whatever type of hardware the user wishes to add to his/her system. This is effected through a tricky scheme of moving a software module into RAM (part of the normal TPA) just below the CCP that contains the additional disk drivers. Figure 1 shows the CP/M-80 system memory map for a typical system both before and after an ADD-ON MODULE is installed.

The movement of the code image to the space below the CCP is handled in almost exactly the same way that Digital Research made DDT, SID, or DESPOOL relocate themselves to the space below the BDOS. In the case of the ADD-ON MODULE, the relocated module modifies the jump address at location 5 in RAM to point to the base of the ADD-ON MODULE. This permits transient programs to know the size of the available program and data area without fear of overwriting the operating system.

Operational software within the ADD-ON MODULE contains code to "drive" the new hardware in the user's system. The drivers are BIOS level type in that the coding format, interface conventions and functions are designed around the normal CP/M-80 BIOS structure. Entry to the routines is performed by having the ADD-ON MODULE trade out the normal BIOS Jump Vector Table with a new set of jumps that point to within the ADD-ON MODULE. The ADD-ON MODULE then checks BIOS entry point calls from either the CP/M-80 BDOS or from direct BIOS calls within the transient program for functions to be performed by the ADD-ON module. The functions to be done within the module are thus handled locally, whereas all others are passed onto the normal BIOS. The ADD-ON MODULE keeps a copy of the previously swapped out BIOS Vector Table entries so that control can be transferred to the normal BIOS when necessary.

A unique feature of the ADD-ON MODULE is the ability to have several different modules, each with a different function, all resident within stolen TPA space at the same time. The code of the ADD-ON MODULE interrupts BDOS calls from the modified jump at location 5 in page zero to check for "already installed" so that the system user does not attempt to install an ADD-MODULE more than once. Having an "address" byte identifying a particular type of BIOS extension allows the TPA to contain several different types of ADD-ON MODULES. Each one relocates itself below the previous and modifies the TPA size pointer via the address at locations 5, 6, and 7. Each additional ADD-ON MODULE also swaps the BIOS Vector Table. The limit of the number of modules is based upon the amount of TPA space available.

Further implementation details of the ADD-ON MODULE philosophy may be gleaned from reading the previously referenced *Lifelines* article series or through detailed study of the program listing at the end of this article. Note that the source code for the relocation module that moves the ADD-ON MODULE to its execution position is included as the first listing within this article.

ADD-ON MODULE Construction

An ADD-ON MODULE is designed for use under single user CP/M-80 2.2. The next several paragraphs, from the previous article, show the procedure to get a version up and running.

The source file, being very similar to a BIOS for CP/M-80 2.2, requires the extension disk definitions to have disk parameter tables, check vectors and allocation vectors. In addition, a directory buffer must be allocated. An ADD-ON MODULE generates the appropriate tables through the use of the macro library DISKDEF.LIB provided by Digital Research on the CP/M-80 2.2 distribution diskette. This macro capability requires a macro assembler to properly process the DISKDEF macros.

An additional implementation requirement is the availability of the assembler output as a .REL file. The Digital Research Relocating Macro Assembler RMAC is just the ticket to generate the appropriate .REL file and process the DISKDEF.LIB macro includes. The command structure below shows how to assemble an ADD-ON MODULE program like the RAM disk demo listing at the end of the article. The source is assumed to be on drive B: and the assembler output .REL desired to be placed on A:. The print file is sent to the printer while generation of the symbol table is inhibited.

```
A>RMAC RAMDISK $AB RA PP SZ<cr>
```

The resulting .REL file has to be converted to a page relocatable format. After agonizing over the problem of easily making a "BIT MAP", I found that the Digital Research Link program distributed with RMAC and PL/I-80 can generate page relocatable files (.PRL type) which are compatible with MP/M-80. After reviewing the LINK output format of a .PRL file, I designed the previously described PRLMOVE program to relocate the LINK output files. The command structure below shows how to produce the .PRL file from the RMAC output on Drive A: entitled RAMDISK.REL.

```
A>LINK RAMDISK[OP]<cr>
```

where [OP] directs LINK to generate the desired .PRL file. The program PRLMOVE and the page relocatable RAMDISK.PRL must be combined into a single executable command file as follows:

```
A>DDT<cr>
DDT Vers. 2.2
-IRAMDISK.PRL<cr>
-R<cr>                       <- Load of .PRL file to
                                RAM to address
                                0100H with code
                                image starting at
                                address 0200H
NEXT PC
nnmm 0000                    <- Convert nn to
                                decimal and
                                remember value
-IPRLMOVE.HEX<cr>
-R<cr>                       <- Read PRLMOVE
                                program in over
                                the .PRL file at
                                load address of
                                0100H.
-G0<cr>                      <- Exit DDT to
                                CP/M-80

A>SAVE dd RAMDISK.COM        <- Save dd pages of
                                memory to get
                                command file.
                                dd = converted nn
                                from above!
```

This results in the .COM command file necessary to make an executable module out of the ADD-ON MODULE.

The New ADD-ON MODULE Changes

The latest ADD-ON MODULE scheme contains repairs and features over and above the original version. This section will briefly describe the problems with the original version and discuss how they were fixed.

1) Warm boot override - the new version doesn't play the dangerous game of changing the value of the warm boot vector address at 1 & 2 in memory. Some people have utilities that index to parameters in the BIOS off this address. The warm boot override prevented the normal "run-of-the-mill" Sysgen, Copy and Format programs from running on a system. The ADD-ON MODULE code now does not modify the jump address at 1 & 2 and instead changes the target address "in place" at the real warm boot vector just below the normal BIOS.

(continued next page)

2) Latest fixes installed to allow two or more ADD-ON MODULES to be present in memory at the same time – the version in the article didn't function fully as advertised! This problem was related to the modification of the address at 1 & 2 as discussed in the previous paragraph. Modification as in the original version made the first installed ADD-ON MODULE transparent and subsequently only the second installed would alter the normal BIOS capabilities.

3) The new example given in this article uses sector deblocking for 256 byte host disk sectors. A neat feature of the included example is the fact that it demonstrates a RAM floppy disk drive within the host TPA memory space.

4) The Warm boot override function now re-enters the CCP by subtracting an offset from the value of the entry stack pointer and going back in at the start up "defeat auto-load" entry point at CCP+3. This fixes a major problem in the first single density scheme from the first *Lifelines* article. It would blow up CCP many times upon typing control C at the CCP command level while an ADD-ON MODULE module is present. The constant reentry to CCP, via the exit stack pointer reference, without reload from the disk, caused problems. Entering a command and then reusing the CCP seems to set some internal flags that screw up operation for subsequent CTL-C usage. The present design of the ADD-ON MODULE with the negative offset pointer calculation reentry to CCP requires that the BDOS and CCP combination be genuine (REAL) Digital Research CP/M-80 Ver 2.2 of total length 01600H bytes. If your system uses a modified BDOS or CP/M-80 system look-alike, then the negative offset to the stack pointer will require a different value. Also, for non-genuine CP/M-80 systems, if the CCP is a different size (not 0800H) bytes, the program PRLMOVE will have to be modified to set down the relocation load address calculation by an amount equal to the difference in the alien CCP size.

Additional Information

I think you will find the new ADD-ON MODULE scheme informative. The greatest area of difficulty I've had in checking out deblocking type ADD-ON MODULES has been deciding when to post the host buffer to disk with respect to operation of the HOME routine. Digital Research's BDOS calls the home routine each time the directory is to be updated. I found during checkout of the latest ADD-ON MODULES, that the last physical sector (256 bytes) of each 16K extent of a file larger than 16K bytes would be written to track 0 sector 1 of the logical unit and not to within the file allocation group as expected.

Digital Research's CP/M-80 BIOS rules state an intent to clear the host active flag on a HOME operation. Unfortunately there seems to be an incompatibility between the timing and relative sequencing of posting, intended to achieve writing the file sectors to the proper place on the disk. The repair for the problem in my latest I/O routines has been to place the home routine in a passive mode. That is, clear the flags like Digital Research says but don't perform a drive restore. See that the driver for the read and write routines never moves the heads out from under them. Implied seek at the driver level may have some reflection on the problem too.

A RAM Disk ADD-ON MODULE

The new ADD-ON MODULE technique discussed within the first portion of this article, and shown as a programming example in listing #2, is an implementation that fits a very small sized floppy disk drive into RAM at the top of the TPA. The characteristics of this "drive" are set up via the DISKDEF macro to be 256 byte host sectors, at eight physical sectors per track. The total allocated space in 1K byte groups is 20K on 10 "tracks". The "directory" contains space for 32 entries.

ADD-ON MODULE startup "formats" the whole RAM drive to 0E5H data pattern so that the drives look empty. As shown, the enclosed program should run on any standard CP/M-80 2.2 system without modification, provided the system has more than a 24K TPA with the CCP resident. A larger amount of memory allows for more TPA to be still be available after you install this ADD-ON MODULE.

Please note that this software is not presented as a practical example of how to make a memory-based floppy disk for your system. I put it out as a public domain "way" of seeing how it can be done without modifying your existing system BIOS and operating system implementation. The enclosed software, if you desire, may very easily be set up to implement a practical RAM floppy in bank-switched memory. My suggestions about the optimal hardware set up for this purpose are:

1) Arrange to have the ADD-ON MODULE resident in RAM above 32K (8000H) when it is relocated below the CCP. A 56K system, or larger to 64K, should be plenty large.

2) Place the bank-switched RAM at address 00H of your 8080/Z80 CPU so that it is switched in banks of 32K. When access to the RAM drive is desired, make the ADD-ON MODULE switch out the normal CP/M-80 lower 32K in favor of additional banks to contain the "disk drive image".

3) Lastly, do not "format" all of the RAM memory within the start-up of the ADD-ON MODULE. Instead, write a short utility program entitled "RAMFRMT" that does this function. This allows rebooting your CP/M-80 system without loss of RAM disk data in case the running program crashes.

Concluding Comments

I find ADD-ON MODULES to be quick, easy to debug, and a reliable way of adding disk I/O for more drives to a CP/M-80 computer set up. If you have any additional thoughts or find problems/solutions that I have not uncovered yet, please call me so we can discuss them. I want the ADD-ON MODULE

(continued on page 48)

Opinion
Talking About You

Jane V. Mellin

What About You?

For many months now I've been reading your responses to the surveys we mail out with renewal notices. The most recent surveys are currently being processed, and you'll hear about the statistics as soon as they're available. However, I've found that by reading every survey I get a strong "gut" feeling about you. Of course, the statistics can only offer the coldest (though most accurate!?) portrait. Some of your concerns and desires just won't appear in the final, numerical analysis.

So I want to share a few of the major trends I've detected. To begin with, your responses show originality of expression and a real involvement with the magazine and its goals. This is of course what every editor hopes to find in a readership – it indicates that *Lifelines/The Software Magazine* has stimulated you and made you think.

Most of you look to *Lifelines* for objective, in-depth and comparative reviews, and that's what you want continued. BASIC comparisons, screen editor evaluations and data base management system reviews are among our most popular review series. Currently, we find that our new communications series and our survey of applications development tools are drawing your letters and calls.

You might be interested in how our "group" reviews develop. I'm referring to those series where several authors are involved in writing the reviews. The process is a study in the cooperative exchange of ideas. Generally, each writer presents his or her input on the form a review series should take. Different types of software have different aspects and criteria for study; *Lifelines* has certain broad categories for rating products, but we find the four point, four grade method to be a *reductio ad absurdum*. So we go beyond the documentation, error handling, etc.) to discuss the product's unique qualities: its sophistication and flexibility (or the users to whom the system is and isn't suited), the tasks the software seems to aim specifically at accomplishing and how well it performs them.

In short, our reviewers first agree on the spectrum of software a series will address, then on the general areas of performance relevant to that group of products, and finally, the handling of individual programs. We think the results reflect that objectivity you are looking for, while going into some depth. Our reviewers compare notes at length on different products, adding a scope of experience to our series.

Not surprisingly, you're making a strong demand right now for more information about the new operating systems that have been growing up overnight, like mushrooms. We're investigating them for you, but bear in mind that articles like Ron Fowler's essay on TURBODOS, and Mike Karas' evaluation of SB-80 require months of using the system under review. What you get at the end of this process is a clear idea of exactly what you can expect from these products.

Ward Christensen's tutorials have brought a strong and vocal response: praise, inquiries, suggested topics, requests for more. Your letters and survey comments have helped in the planning of future tutorials, and Kim DeWindt's new Z80 series is already becoming a star. These two authors are experienced in teaching programmers; they know which approaches are clearest to the novice and can anticipate what some of your questions will be. But they rely on *you* telling them what topics should be covered.

Practical programming articles, like those written by Kelly Smith and Michael J. Karas, have stimulated a hearty response from those among you who find *Lifelines/The Software Magazine* to be an effective professional aid. Dave Hardy's introduction to typesetting interfaces was a sleeper; the calls are *still* coming in.

In The Works

Some specific products we'll be reviewing include: Electric Blackboard, Mr. EDit, JANUS Ada, VolksWriter, Fast Figure, ASCOM, Programmer's Apprentice, EasyFiler, EasyWriter II, LOGON, BASIC/Z, The Archivist, RBTE-80, TERM2, Timin FORTH, and many many more.

We've requested a lot of program development tools and systems for review, and we're anticipating a very thorough survey and examination of these products – covering every category, from screen development utilities to full-blown applications systems.

As you've noted, communications is one of our primary concerns. Jim Mills is going to tell you in detail how to set up your own communications system. He'll discuss your options in terms of CPMUG software (see Volume 84, in this issue), and investigate the various levels of sophistication you can reach in creating bulletin boards and networking systems. In addition, Allison Phillips will offer tutorials which explain the basics of CPMUG software, including MODEM76.

One aspect of the communications revolution is the "electronic cottage". Our MicroMoneyMaker's column, which will resume next month, promises to encompass the "electronic cottage industry", a dream I think many of you have. Charles Sherman came up with the brilliant idea that we all should share our ideas about this creative and growing field. Of course, we're offering prizes to encourage your responses, but we feel that the whole endeavor can only benefit all you micro wheeler dealers. And some of you who aren't entrepreneurs may have suggestions on how the future captains of the electronic cottage industry can accommodate *your* needs.

In the coming months, I'll be talking more about you. Correct me if I'm wrong, and help me fill out this portrait I've been painting. As I think you've gathered, your calls, letters and survey responses are our lifeline, our vital link. Nobody will go unheard.

T.I.M., Part 2

Davis A. Foulger

When new developments open fantastic opportunities for introducing new products (and, let's face it, getting rich off the spoils) the rush to beat the pack to market can carry both risks and rewards. The reward, if your product is the first that satisfies a *bona fide* need, can be instant recognition, market leadership and lots of bucks. The risk is that the product won't be fully enough developed to take advantage of the new technology.

Although there are arguments over whether or not IBM really broke any new ground with its Personal Computer, there seems to be little question about the fact that it opened up great opportunities for software developers. With predictions of IBM PC sales in the hundreds of thousands, the bandwagon of software developers ready to write or adapt software quickly grew to overflowing. Today, with those predictions fulfilled, software seems to be appearing for the PC at a fantastic rate.

There are, for instance, at least fifteen word processing packages available for the PC, many of which are completely new and as yet unavailable on any other computer. That number is in part a reaction to the poor quality of the initial IBM PC word processing entry, a cumbersome and difficult to use version of Easywriter, but is also a reflection of the advanced features of the machine (see this month's review of Volkswriter).

The developers of T.I.M. (Total Information Management), a database management system, distinguished themselves in the race to bring new products to the IBM Personal Computer market in two ways, both of which were mentioned in the July *Lifelines/The Software Magazine* preliminary article on the database package. As it was the first business software package for the IBM PC offered through a source other than IBM's software central, it had an excellent lead in arriving in ComputerLand stores. Indeed, many customers were able to obtain T.I.M. before they could obtain the IBM-sponsored applications packages.

Because, moreover, it was the first data base management software package available on the IBM Personal Computer, it has sold more than a few copies to IBM PC users who needed a database management *immediately*. As a result, T.I.M. has a good-sized owner base among IBM PC users. Although there are no doubt large numbers of T.I.M. owners (both among IBM PC users and CP/M-80 users) who still use the package, there are almost certainly those who have moved on to more flexible packages like dBASE II and Condor (see the April, 1981 issue of *Lifelines* for a review of Condor and the August, 1981 and May, 1982 for reviews of dBASE II).

An Overview of T.I.M.

If my experience is any measure, T.I.M. will first delight users, then disappoint them, then make a comeback to respectability. It will not be all things to everyone, but will give most a fairly powerful database.

In its heart of hearts, T.I.M. is a database management system for novices. It is designed for people who need a moderately talented database system and who have no desire to program. Except for the installation procedure that you must follow when you first buy the package (yes, you will have to read at least some of the manual) and some advanced procedures that many may never have cause to worry about, everything in T.I.M. is menu-driven. You just pop the diskettes in the right drives and turn your machine on. After that, T.I.M. does everything but make your decisions and enter your data.

In this, the philosophy of T.I.M. is pretty well in line with the new kind of philosophy we can expect to proliferate as advanced 16-bit *supermicrocomputers* like the IBM Personal Computer become widespread. One of the best uses of extended memory like that of the Radio Shack Model 16 (500 Kbytes) and the IBM PC (up to 1 Megabyte, depending on the operating system) is the improvement of the user/software interface while the range of the software and its operation are enhanced. Unfortunately, this is not exactly what happened with T.I.M.

While it is true that T.I.M. is both easy to use and moderately versatile, the package is troubled by three shortcomings – slow operation, limited capacity, and a deficient underlying model of the data. Of the three, the slowness is most likely to put off less experienced users.

"One Moment While The Program Is Loading"

T.I.M. is written in Microsoft BASIC. This fact enabled Innovative Software, Inc. to bring out T.I.M. quickly after the introduction of the IBM PC, which, as it happens, offers users a good Microsoft BASIC in the $40 disk operating system (PC DOS) that comes standard with the computer. Since T.I.M. was already implemented in Microsoft BASIC for CP/M-80 machines, it was a rather simple matter to adapt the package to the IBM PC.

Microsoft BASIC is a rather powerful implementation, particularly in the advanced version available on the IBM PC. It is convenient for applications development, being interpreted rather than compiled; the price of this ease of use is in speed. No one would call Microsoft BASIC a speed demon. True to the language it is written in, T.I.M. is a slow operating database system, particularly in the small, repetitive tasks that really need to be speed-transparent to the user. I read several books (in pieces) while waiting for T.I.M. to perform various tasks, and although I am a reasonably fast reader, those waits were frustrating.

T.I.M. may be too slow for some practical applications, and the time it requires could actually decrease productivity. It should, in general, excel in real time "interview" applications, where human give and take will tend to constrain the speed of data entry more than the package does. It will be least satisfactory in high-speed "transcription" applications – mail order data entry, for instance – where it will tend to slow the data entry process. The user can solve this problem by writing a BASIC program for data entry and then converting the completed data file from ASCII to T.I.M.

The solution is a better data entry routine, either in BASIC (even the BASIC routine seems to be much slower than it could be) or, preferably, in assembly language. It currently takes T.I.M. between five and ten seconds, depending on the complexity of the data file and the screen, to draw a new form for data to be entered on. That wait is frustrating and, in the opinion of this reviewer, unnecessary.

Although T.I.M. is slow in other places, that slowness can usually be controlled with effective time management. Sorts and other long-winded utilities can be scheduled for lunch hours, meetings, breaks and other business that takes the user away from the computer. Five and ten second interruptions in the data entry sequence, on the other hand, can be difficult to utilize.

Forty Fields: An Unbreakable "Record"

The overall capacity of the database system also leaves something to be desired, at least in advanced applications. Although T.I.M. is able to handle up to 32,767 records in each file, those records are limited in size. Records are limited to forty fields of up to sixty characters each. Sixty characters are, of course, sufficient to fill the width of a page. This is, however, less than half the maximum field length available in many other data-base systems.

Field length is a small annoyance compared with the limitation to forty fields per data file. There may be very few applications that will not fit within the constraints of those forty fields, but almost any advanced user will have at least one. Plainly, T.I.M. will not do everything some advanced users would like a database package to do. However, the novice users most likely to enjoy T.I.M. will probably not find this forty field restriction to be a problem.

Some Card Tricks: The Underlying Data Model

A third limitation of T.I.M. is found in its *underlying data model*. While it is true that a broad range of database applications (including almost any application that might be required by many users) can be implemented in a single file, increasingly complex applications stretch the abilities of the single file database. Take, for example, a mailing list of family and friends. The mailing list is relatively simple so long as we stick to simple *relationships*, like the *one-to-one* relationship between a family, its mailing address and the wedding anniversary of the parents (useful when sending out anniversary cards) or the *many-to-one* relationships between various families and their religious convictions (useful when sending out holiday cards).

Processing becomes more complex, however, when we try to keep track of the birthdays of the different members of those various families (useful for sending out birthday cards). Keeping track of the *one-to-many* relationship between individual family members, their birthdays, and the address the card should be sent to creates a problem. This problem can be solved by creating several fields within each family field for family member names and their birthdays in the database record of each family, but this solution creates search problems (several fields must now be searched for birthdays) and ultimately wastes field space within the family record (particularly in families of one that coexist in a database with a family of twelve), a serious problem in the limited field space of T.I.M.

The problem can also be solved by creating a record for each family member. This, however, creates other nuisances. Even more space is wasted (a family of twelve will have its address listed twelve times in a single database). Problems, moreover, can occur in sending out the one-to-one relationship cards. A family of twelve, for instance, might wind up with twelve copies of the same Christmas card, each addressed to a different family member, but all sent to the same address. Such problems grow even worse in divorce cases where parents have joint custody. Children, in these cases will be members of two families, each with a diferent address. The problems multiply when we have to worry about this kind of *many-to-many* relationship between child, family, address and birthday.

If things can get this complicated in just building a mailing list to send birthday, holiday and anniversary cards, imagine how complex they can get in business applications, where buyers make multiple purchases of different kinds of merchandise and make multiple payments; suppliers make multiple shipments of different kinds of parts and supplies from a variety of different warehouses, and to a variety of different facilities; employees with a variety of different skills work in a variety of different departments.

Advanced applications require a database to draw relationships between a variety of *domains* of differing objects (family members, families, and the different groups that families are associated with, for instance). The true power of a database system isn't realized until the database can treat these different domains of objects as completely separate and independent data files (sometimes called *relations*). It is generally conceded that there are three well-formed models for dealing with distinct, but related, domains of objects – the hierarchical, network and relational models of data.

More Card Games: Differences Among Models

Those who are gaining a full understanding of the differences between these rather esoteric sounding models are encouraged to read C. J. Date's *An Introduction to Database Systems* (1977; Addison-Wesley Publishing Company, Reading, MA), an excellent exploration of good database system design. For now, let us briefly examine the difference in these models by considering a deck of cards. Cards are constructed from two domains. The first, which is called the suit, contains four attributes: hearts, diamonds, clubs and spades. The second, referred to as ranks, contains thirteen attributes: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King.

A card game like *Go Fish* uses only one of these domains. Only the ranks of the cards matter and an underlying data model is unimportant. Go Fish is, of course, a novice's game, just as T.I.M. is, for the most part, a novice's data base.

(continued next page)

Conditions change in a game like *Hearts*, however. Here, one domain (suit) clearly dominates the other (rank). There is, if you will, a hierarchy of domains in Hearts; ranks only matter to the extent that they fall within particular suits. When databases are organized this way, with the access to family members provided only through various families, the database is called *hierarchical*. The biggest problem with hierarchical databases is the same problem you should have if cards were always treated as hierarchical – you wouldn't be able to play Go Fish or poker, two games which have a decided disregard for this kind of hierarchy of domains.

An understanding of the *network* model might be best understood by introducing a third domain of objects, card players, and a third game, Bridge. In each hand of a game of Bridge, each player is linked, by his or her cards, to the domains of suits and ranks, and by agreement, to another player. Once the cards are dealt, players bid what they think they can win, either within a dominant suit, or with no dominant suit.

Clearly, a hierarchical model won't do for representing bridge or any similarly complex database. None of these three domains clearly dominate any of the others (although suit still generally dominates rank). The network model gets around this problem by providing for flexible links that can be changed at will (with each shuffle of the cards). There is a problem, however, with the maintenance of those links in the face of repeated change; repeated changes are required in the complex range of links associated with all the files (suits, ranks, players), a time-consuming task.

A Touch of Reality: The Relational Advantage

A *relational* database system reduces this complexity by building new database files that lay out the relationships between members of these various domains. The relational database contains more than the three data we have discussed so far – a data file of suits and their characteristics, a data file of ranks and their characteristics, and a data file of players and their characteristics. It also contains a data file card explicitly stating each of the relationships between members of the suit and rank, and a data file (for each hand) that states the relationships between each player and the deck of cards.

It may seen that these "relational" data files add complexity, but appearances deceive. These files actually reduce the amount of storage needed for the entire database (compared to a network model) by eliminating the need to tag each attribute of the suit, rank, and data files with their links to the other data files. They also speed up the process of changing those links, as changes need only be made to one file when links change. Finally, the files increase the flexibility with which we can use the attributes of various domains.

If, for instance, we want to play a game of *Mate*, a card game which only uses twenty cards out of the standard deck, we can easily set up a Mate data file (relation) that specifies the twenty relations between suits and faces that exist in the game and the characteristics of each (in practice we simply remove the cards from the deck and specify the rules). Mate could, of course, be specified under a network model, although the process would be much more complex. Since, however, the rules of hierarchy in Mate can turn upside down with each play of the cards, it would be impossible to specify adequately under a hierarchical model.

Here we see the real value of the relational model, which most closely emulates the way we operate in real life and the way domains of information relate to one another. We use a deck of cards that relates suits to ranks (two data files of attributes and information we keep in our heads) instead of dealing a deck of suits and a deck of ranks. Similarly, we use a billing data file that relates items sold (a data file) with the addresses of customers (another data file) instead of sending out a mailing to all customers, including those that have no outstanding balances, informing them that bills are ready and can be picked up and paid. The relational model is both simple and real. Thus, almost any database suffers when it does not adopt the relational model.

T.I.M.'s Underlying Data Model

The underlying model of the data in T.I.M. seems confused. The software seems to have the flexibility of a network model, the reporting characteristics of a hierarchical model, and some of the reformatting capability of a relational model – without really being any of them. Two files can be related in T.I.M.'s report procedure in a full range of relationships (one-to-one, one-to-many, many-to-one, many-to-many), opening the door for the two family and family member data files of our greeting card example.

The suit and rank data files of our playing card example might also be implemented and related within T.I.M., but it would be difficult to relate the player data file with either in a flexible way. In both the greeting card and playing card examples, the implementation of the files would follow what would seem to be a network model, but the use of those files would, although retaining the flexibility of a network model, entail hierarchical procedures. Oddly, however, the reports produced could easily be used to form a third "deck of cards" data file. The new data file could not, however, be used in the way it would be used in a relational database.

A Tour Of T.I.M.

Now that we have examined the general strengths and weaknesses of T.I.M., it is time to look at the specifics of the package. Many of these specifics are outlined, in detail, in tables 1 through 3, and I will not touch on all of them in this discussion. A user's tour of T.I.M. begins with the user's manual, which is a combination of reference manual (fourteen blue tabbed sections) and tutorial (four white tabbed sections). The documentation is quite complete and fairly easy to read, but clearly shows the rush with which the IBM PC version of T.I.M. was brought to market.

The manual seems little changed from the CP/M-80 T.I.M. manual. Although T.I.M. generally takes full advantage of the IBM PC's function keys, these keys are never mentioned in the manual in more than a footnote or two. The keys are never shown in the manual's tables. It is also clear that the BASIC language statements printed in the manuals appendix are the ones used in the CP/M-80 version. When I encountered error messages (I don't think that these errors represented bugs so much as miscalculations), they almost never had anything to do with the

BASIC language statements that supposedly were being represented. (The code was protected from listing and I couldn't confirm this impression.)

Despite the manual's CP/M-80 underpinnings, it remains an excellent introduction to, and reference for, T.I.M. Many users will have very little use for the manual, however, as the menus which guide the user through every step of the program are excellent. Indeed, after reading Chapter 0 (Preliminary), which gives installation instructions and an overview of the package, I used T.I.M. for about a week with little need for the manual.

Out Of The Frying Pan And Into The Fire

With T.I.M. installed (a simple procedure), the user only must put floppy disks in the right places and turn the machine on. The program moves through several displays and one question (where is the data?) before showing the user its main menu. The main menu is the center of all that happens in T.I.M. The user is offered twelve options on this menu (which correspond to twelve of the chapter headings in the manual). One of these, the word processing interface, wasn't operational in my copy of T.I.M., an event which was something of a disappointment to me. The other eleven commands were all intact, however.

These eleven commands can be broadly grouped into three broad areas - file oriented commands, output oriented commands, and user oriented commands. Three user oriented commands allow the user to look at a directory of the T.I.M. data files available on various disks, examine the specifications of the various files within that database, and get help.

Of the three, only the help command, which if used correctly gives the user a capsule overview of the actions taken by each of the twelve commands, leaves anything to be desired. Simply typing the indicated H command results in a message that says that Help is not implemented, a rather misleading remark. The user has to turn to the manual to discover that help requires two letters, an H followed by the command the user wants to know about. It strikes this reviewer that the fix for this problem is simple. It ought to be fixed in the next release.

A fourth exit (X) command, which is supposed to exit T.I.M. and return the user to the SB-86 (MS-DOS, PC-DOS) disk operating system also needs a fix. It currently returns users to BASIC instead of the DOS. The fix, however, is as simple as ending the T.I.M. to DOS command with a *system* command.

The remaining seven commands will have to be described more carefully. For now, it should be noted that that four of these commands are data file oriented commands and three of them are output oriented commands.

Playing God: Creating A New Data file

After reading over the menu for the first time, a user might select the first of several file oriented commands and attempt to *create a new file*. Entering file creation mode is a simple matter of pressing the *C* key. But the plot quickly thickens. In the IBM version of T.I.M., one discovers that all three T.I.M. disks do indeed have a purpose. Here, the program asks the user to insert T.I.M. disk 2.

Written in BASIC, T.I.M. is a fairly substantial and complicated program. It requires about 295K bytes of disk storage, far more than is provided on the 5.25", single sided, double density disk drives initially offered by IBM. The capacity of these drives was conservatively set at about 160K bytes.

This capacity is generally conceded to be the biggest mistake IBM made with the machine, especially now that 5.25" disk drives can be bought with capacities of half a megabyte and more. IBM has recently moved to ease this problem somewhat with the introduction of double-sided drives that double the capacity of most disks, but the low end capacity of the first drives has forced software providers to adapt large software packages to small disk capacities as best they can.

Innovative Software's solution was to repackage T.I.M. on three disks, with a series of less-used utilities called in from the second and third disks as needed. If the packaging had been more carefully planned, the job probably could have been done with only two disks, but the three disk format gives Innovative Software some needed mobility in updating their product. In any case, T.I.M. should almost certainly fit on a single disk in the new IBM double sided drives. It should be noted, however, that although the multiple disk format can be time-consuming and somewhat annoying, it is not really a problem. Disk changes are infrequent and tend to be tied to even more time-consuming tasks.

With the second T.I.M. disk placed in drive A, we enter the *create a new file* menu that includes four options. The simplest of these options lets you create a file that matches another file (an operation that can be particularly useful when two data files will have duplicate structures or when one is the subset of another). When selected, this operation takes about thirty seconds to create the duplicate file specification under a new file name. Overall, the operation is simple, powerful and convenient.

The operation would be handier if it incorporated format modification routines that allowed the user to delete unnecessary fields from the old data format, add new ones, and otherwise restructure the "model" data file structure in the copy. It is often the case that an old data file is very similar to what is needed in a new data file, but not exactly the same. It doesn't seem like the implementation and use of such an option would be terribly troublesome or complicated at this stage.

A second option, implicit to the limitations of the first, is the ability to create a new data file format. Although one of the more complicated operations in T.I.M. from the standpoint of the number of decisions that a user has to make, this option remains very user friendly, with a full set of menus and prompts to guide the user through the task of building a data file structure. This option would profit from a display of the fields that the user has already entered (probably possible in two or three lines of BASIC) and the ability to change the order and structure of the fields in the data file as it is being built. These two features would make the use of the file creation facility easier for novices who will often fail to think far enough ahead to plan a data structure before entering the file creation facility.

A third option allows the user to specify a custom screen for forms entry. Use of this facility is a good idea, as the

(continued next page)

standard screens implemented in T.I.M. are somewhat painful to work with. Unfortunately, the facility for creating custom screens is also troublesome. In my notes I have described the facility as showing "no thought given to the keyboard" and "poor programming". In truth, if I were Innovative Software, I'd be embarrassed by this one.

Why, for instance, are the F1, F2, F3 and F4 function keys on the IBM keyboard used for cursor movement when the keyboard already has a full set of equally addressable cursor movement keys? And why is the movement of the cursor so slow? Microsoft BASIC is slow, but you can certainly move the cursor around faster than that. I could go on, but the point is made. Somebody didn't think when this part of the program was revised for the IBM PC.

The custom screen also disappoints me in its failure to allow users to put titles and other relevant menu and prompting information on these custom screens. The screens would be far more useful if such provisions were made.

Data Entry and File Modification

With a file created and, perhaps, given a custom screen, the user is faced with a fourth menu item, exit, which takes the user back to the main menu via a prompted disk change. Now we can start to put data into the our newly specified data file and, hopefully, begin using that data. Facing the main menu once again, we are confronted with three commands that will allow us to turn our data file structure into a data file.

The first of these is the *Add/Inspect/Update a record* menu option. This well-designed group of routines allows the flexible entry, modification and updating of data in the data file. Records can be searched for strings (a name, for instance), key-fields can be modified, and the user generally has free movement within the data file.

The routines are not without problems, however. These problems, already discussed, involve the slow speed with which screens are drawn for data entry. One suspects that a revision of the BASIC routines that draw the screens would speed it up tremendously, at least on the IBM PC, but the speed increment can certainly be obtained with assembly language routines.

In the absence of such routines, some users may want to write their own data entry programs (or use VisiCalc, for instance) and then take advantage of one of the options available under the main menu's *utility commands* listing. Specifying these utilities results in a prompt for T.I.M. disk 3. The utility in question, which converts *ASCII to T.I.M.*, allows just about any ordered set of data to be imported into a T.I.M. data file.

Other utilities available under this menu selection include a routine for transferring T.I.M. files from one disk to another (a program that must be used in any such transfer operation), a routine for transfer of information from one T.I.M. data file to another (allowing files to be restructured as needs change), and a routine permitting large files (on a Winchester hard disk, for instance) to be cut up into a series of small files (on floppy disks). This last operation can also be reversed, with a series of small files built into a single large one.

Another series of utilities (this set stored on T.I.M. disk 2) is accessed through the *file maintenance* option. One set of routines provides for renaming field titles or a T.I.M. file. Another is for the erasure of key-fields, duplicate or unwanted records, or entire T.I.M. files. Another utility allows data files to be sorted into a new sequence along a set of key-fields.

The availability of the utilities gives T.I.M. a great deal of power in dealing with a wide range of database management problems. Many are rather slow, however, especially when dealing with large data sets, and should be scheduled judiciously at times (lunch hours, breaks, meetings, etc.) where they won't interfere with getting other work done on the microcomputer.

Turning The Data File To Good Use

The remaining three options have similar menus, reflecting their similar purposes – list generation, report generation, and record selection. The general purpose of each of these utilities is to provide a usable output.

The list generator, for instance, can be used to produce mailing labels and, if desired, bills or checks. The report generator turns out a summary of information contain within the data bases, and allows the totaling of fields across cases with two levels of subtotals. The record selection routine, moreover, lets the user create a new data file that is a subset of another, particularly useful when billing customers or sending out birthday greetings to friends and family born in the same month.

These routines are not without limitations. The records selector, for instance, sorely misses a NOT operator to join its reasonably decent range of selection options. The report generator, moreover, would probably be improved by a better underlying data model. And the list generator would profit from the availability in the IBM PC version (it appears to exist in the CP/M-80 version) of a facility for updating label formats stored in the list generator's format library.

The biggest problem faced by these routines is, however, the way such format libraries are implemented in all of them. Each utility has an associated library of formats, established by the user, that can be used over and over again in similar applications. These libraries are, unfortunately, located on the T.I.M. program disks. It is unfortunate because the IBM PC format versions of the T.I.M. disks are already crowded. There really isn't any room for much of a library. Users will quickly find that they have run out of space.

The solution is to relocate these libraries on the data disks in conjunction with the appropriate data sets. It is difficult to tell how difficult that move would be, but it should be fairly simple.

Summary and Recommendations

T.I.M. is a reasonably powerful and useful database system, distinguished from other database systems mainly by its ease of use. This ease of use is achieved, however, at the price of a good bit of the flexibility that advanced users expect of database packages. While T.I.M. can link two files for some operations and is capable of a wide range of data sorts and restructurings, it cannot easily be used for many advanced database applications.

The package is also distinguished, unfavorably, by its speed, which can be frustrating and may lead to productivity declines in some applications. The package stands to be significantly improved here and elsewhere, and it is hoped that its writers will take on the task of making those improvements.

It should also be noted that although T.I.M. is implemented on a machine (the IBM PC) that has considerable advanced capabilities, it fails to take advantage of them. It is hoped that a future version will implement a new and more clearly relational database system, employing the extended addressing space of the IBM PC and other supermicrocomputers to increase speed and processing power, while maintaining the high standards of user friendliness and ease of use that distinguish T.I.M. from its competition.

T.I.M. is recommended as a database language for non-programmers and undemanding applications. Its menu driven structure and wide range of utilities give it power without sacrificing ease of use. T.I.M. is not, however recommended for demanding applications and users who don't mind getting their hands a little dirty in learning how to use a flexible database. These applications and users will find Condor or dBASE II far more satisfying.

It should be noted, however, that T.I.M. does give its users something of a glimpse of the future, when large supermicrocomputer addressing spaces will have resulted in very user friendly application packages, consumer sales will make for opportunities such that software developers may be able to become very rich off the spoils.

TABLE 1
Facts & Figures

Package: T.I.M. (Total Information Management) version 3.11
Price: $495 for IBM PC ($695 for CP/M-80)
Systems Available For: IBM Personal Computer, CP/M-80
Required Supporting Software: Microsoft BASIC
Memory Requirements: requires, and uses no more than, 56K RAM
Diskette Capacity Required: Two Disk Drives
Utility Programs Provided: ASCII to TIM conversion (Word Processor Interface)
Record Size & Type Limits: Internal storage is fixed length ASCII stored as a single record (no line feeds or carriage returns). Up to forty fixed length fields of up to sixty characters each can be stored in up to 32,767 records per file. External data can be produced by the list and report utilities with record lengths of up to 131 characters (delimited by line feeds and carriage returns). Multiple lines can be produced for each record. Variable length external data can be converted to TIM if fields are delimited by commas.

Ed. Note:

Innovative Software has just announced a compiled version of T.I.M. to be released shortly in 8086 code for the IBM PC (and hopefully soon thereafter in 8080 code for CP/M-80 machines). The run-time improvement is claimed to be in the range of 10-40 times faster.

It is not immediately clear which, if any, of the other issues raised in this review are resolved in the new version.

TABLE 1
Facts & Figures (cont.)

User Skill Level Required: A novice should be able to master most of the system in a matter of an hour or so without so much as looking at much more than the preliminary information in the manual (which is really only needed for the backup diskette initialization procedures). Many professionals would be better served by a faster or more flexible package like dBASE II.
System Upgrade Policy: Upgrades, when available, will be issued at unspecified prices.

TABLE 2
Qualitative Factors

	Rating
Documentation	
organization for learning	6
organization for reference	6
readability	5
includes all needed information	7
Ease of Use	
initial start up	7
conversion of external data	4
application implementation	3
operator use	5
Error recovery	
from input error	6
restart from interruption	4
from data media damage	5
Support	
for initial start up	4
for system improvement	4

* Ratings in this table will be in a 1-7 scale where:
1 = clearly unacceptable for normal use
4 = good enough to serve for most situations
7 = excellent, powerful, or very easy depending on the category

(continued next page)

TABLE 3
Data Management Capabilities

A. Underlying Data Model

1. *Data Types*
Alphanumeric (including special inverted name fields and data fields), Numeric (up to four decimal places with special dollar and sequential fields), and Calculated (including totals).

2. *Relationships*
There doesn't seem to be any clear model underlying the data. The reports feature does, however, support a full range of relationships – one to one, one to many, many to one, many to many – between any two related files.

B. Functions Provided

1.a. *Data dictionary maintenance:*
There is no central data dictionary and data is not file independent. Naming is file dependent. Names can, however, be changed, and where fields match is length and domain, two files can be linked in report specifications.

b. *Data reorganization and conversion:*
Data can be converted into TIM from ASCII using a conversion utility. Lists and reports can, moreover, be written to disk for use by other programs. Short files with common formats can be concatenated into longer ones. Long files (stored on a Winchester hard disk, perhaps) can be split into smaller ones (e.g. – for storing a floppy disk backup of a long Winchester hard disk file). Files can be completely restructured with fields added and deleted, fields from two or more files combined into a new file (this requires some care), duplicate and deleted records removed, and files sorted on key fields.

2.a. *Data Entry and Editing:*
Users can specify custom screen formats of up to two pages to match forms. Custom screen generation procedures are counter-intuitive and slow, however. Records are easily updated with simple error recovery and data modification procedures. Data entry can be unbearably slow, however, as it takes several seconds (4.6 with 5 fields; 6.8 seconds with 12 fields) for each screen (record) to be called up and drawn.

b. *Report Generation:*
Reports can be generated using up to two files (main and detail). Subtotals can be computed with two breakpoints. Grand totals are also computed. Reports can be written to either print or disk, providing a means for generating sub-total data for movement into new data files.

TABLE 3
Data Management Capabilities (cont.)

3.a. *Data selection by predicate:*
Eight relational operators give users a relatively strong data selection capability. "And" and "or" operations are supported, but "not" operations are not.

b. *Data Joining and Relating Multiple Data Sets:*
Up to two data sets can be related in the report procedure. Data joining will require effort and planning, but can be done within the package.

c. *Calculations on Data:*
Different fields can be totaled and built into new fields through standard calculations (+,-,*,/) within records. Totals and two levels of subtotal are supported in the report generator.

4.a. *Data Independent Application Interface:*
None. The menu promises a word processing interface in future releases, however.

A Review of Pascal/Z

James Gagne

Pascal/Z is one of the earliest implementations of Pascal for the CP/M-80 operating system. First released in 1978, it is now in version 4 and has been used to implement itself for some time. In this review, I will evaluate the features and performance of Pascal/Z and compare it with the other two major CP/M-80 implementations, Pascal/M and Pascal/MT+ (particularly Pascal/MT+).

I was invited to review Pascal/Z in response to an earlier review of Pascal/MT+ (*Lifelines*, November, 1981). In that article I noted that although MT+ was an ambitious undertaking, it had a number of problems. With a one-line program, it took two minutes to go from editor to then compile, link, run, and return to the editor, versus 15 seconds for the same program running under UCSD Pascal on the same computer (Version II.0; compare with 35 seconds for Apple Pascal and about a minute for Pascal/M). I was not impressed with MT+'s ease of use or suitability for the development of large programs. Error messages were clear but untimely. To discover many of your problems you had to wait through a three-pass compile, with a basic compile speed of about 200 lines per minute in addition to a 60-second fixed overhead. (All times are from my Z80-based S-100 system which uses double-density, single-sided, 8-inch disks.)

The MT+ compiler is heavily laced with features, particularly those supporting: a) 99% of the ANSI standard, b) a high degree of compatibility with UCSD Pascal syntax, and c) anything you'd need in order to do bit diddling and other low-level programming. MT+ features an optional Speed-Programming Package that sounds wonderful but was in fact not particularly fast or useful.

Ithaca InterSystems responded to this review, extolling the advantages of Pascal/Z, with an invitation to review it and their assurance that I would like it a lot better than MT+. To support their claim, they included reports of benchmarks, which showed that for the programs used, Pascal/Z created generally smaller code files than MT+ and ran them more quickly (with both compilers set up to generate the quickest code). It was noted that Pascal/Z code was fully re-entrant, which MT+ was not. (I did not attempt to verify these claims.)

During the twenty hours I spent reviewing Pascal/Z, I became disenchanted because of a number of real problems with Pascal/Z; these greatly limit its usefulness for the commercial or contract programmer. But Pascal/Z is not a total loss. If you are willing to put up with its poor user interface and other idiosyncrasies, it is possible to write substantial programs.

Reviewer Biases

I started out with CP/M-80 when I added a floppy disk system to my IMSAI in 1977 and used CP/M-80 extensively until I finally got UCSD Pascal running a year or so later. CP/M-80 sure beat trying to load programs from a cassette tape recorder, and I became a reasonably competent Z80 assembly language programmer and a fan of the Electric Pencil. But eventually, UCSD Pascal won out.

I love the UCSD system because it is so fast and easy to use; in contrast, CP/M-80 seems klutzy. Pascal is appealing because of its clarity and elegance, and because of the ease with which software tools can be built and maintained. I began one of the first users' libraries for UCSD Pascal and am now chairman of the Software Library of the UCSD p-System Users' Society. Several of the software tools I wrote and then donated to that library have been incorporated into The Incredible Text Printer, a 12,000-line text formatting program that runs on the p-System and combines power with clarity of operation and ease of use. (*Editor's Note*: See last month's brief under New Products.)

Pascal excels at complex programming tasks that involve the manipulation of information (as opposed to number crunching or hardware-dedicated tasks), particularly when your program is large and you would like program and data flow to be orderly and well structured. It is one of the better standardized languages available for microprocessors, so that a well-written program will port easily from system to system. Because the majority of Pascal programs today are written with the UCSD system, the extensions to the language created by UCSD Pascal have formed a quasi-standard which is likely to be in force until some more official set of extensions is created by international agreement. Thus, as a reviewer, I expect a Pascal compiler to support the creation or modification of large programs that may have originally been developed on the UCSD system, once the obvious changes have been made to accommodate the new operating system.

One of my most important review criteria is how much the programming system improves my productivity. Unfortunately, this position is not widely shared. One computer company executive (whose programmers all use assembly language) told me, "I really don't care how easy it is for my programmers to program; that's their job. What counts is the ease of use of the end product." I disagree. Programming resources must be carefully respected.

There is an inherent limit to the number of programmers who can work together effectively to complete a project. (Imagine hiring thirty composers to help you write a symphony faster.) If you have an especially large undertaking, requiring several programmers, each should have a well-defined portion of it which is uniquely his/hers.

It typically takes from one to four years to write a serious, commercial-grade program. From my experience with the UCSD Pascal community, programmers using high-productivity development systems don't just do the same old thing in

(continued next page)

less time. They produce a much more refined product in the same one to four years. So if you want your product to be alive in two or three years, you'd better make the most of your time. And here is where the majority of the CP/M-80-based products fall apart, including Pascal/Z. Pascal/M is the only one of them that truly helps you out, and it's still too slow.

An Overview of Pascal/Z

The Pascal/Z system consists of a compiler, relocating macroassembler, and linker, along with a debugger you can link in, a separate peep-hole optimizer (in case you want to go over your code once again), and complete source text to the machine-language run-time library. There is a utility (which works!) to convert from UCSD-format disks to CP/M-80 format, as well as an impressive collection of demo programs.

Pascal/Z is unique among CP/M-80 implementations, because its one-pass, recursive-descent compiler produces "TDL"-like assembly language (i.e., Z80 mnemonics that use extensions of 8080 mnemonics rather than those of Zilog). The resulting text file is then assembled, along with an outer shell common to all Pascal programs (common initialization code, constants, etc.). Finally, the relocatable machine language is linked together with the needed routines from the run-time library.

(In contrast, MT+ produces native machine code directly, creating several temporary intermediate files on the disk during its three passes. Pascal/M produces p-code directly in one pass, requiring no intermediate steps; but there is considerable overhead in then interpreting the p-code at run time.)

One would think that it would be an incredible nuisance to have to assemble something you've compiled in order to link and run it. But actually, the compiler is reasonably quick, and you can find about all your errors in one pass. You can think of the assembler as the second and third pass of the compiler. It *almost* works that way...

With a 56K CP/M-80 system, I needed the smaller, overlay version of the Pascal/Z compiler, which fits in a 48K TPA but runs half as fast as the 56K version. Its fixed overhead (i.e., with a one-line program) on my system was 24 seconds, almost all of it manipulating disks. Time to compile a 800-line program was 5 minutes, working out to about 160 lines per minute. There is reason to believe that this time would be considerably reduced if something smaller than an assembly-language file were produced: the product of the 800-line program was a text file occupying 84K of disk!

As you might imagine, the 2-pass assembler took a while to digest this file, which was assembled together with the 10K-long program shell routines called MAIN.SRC. Assembly time was six minutes, fourteen seconds. Add linking and the total time from Pascal source to ready-to-run machine language was 14 minutes.

Not exactly high productivity, considering the modest size of the source file (800 lines, roughly 30K bytes). But since there was so much disk access, and floppy disks are slow, this time might be three to five minutes with a hard disk.

Pascal/Z would be a lot faster during development if there were a way to disable the listing file or the assembly-language output. Alas, you must always send them somewhere, either to a disk or to the console or printer. (One way to turn off the listing is to use the $L- directive within the program source, which allows only error lines to go to the listing. It was not mentioned in the documentation, but if you put the $L- directive before the "PROGRAM <name>;" declaration, it is ignored.)

There were more problems. The console display during compilation consists of each procedure name followed by one hyphen every ten lines or so. If an error is found during a ten-line section, an "E" appears instead. There is no way to determine *where* an error has occurred during compilation without looking at the program listing (although you may direct the listing to the console and try to spot the bugs as lines go whizzing by). Even in the program listing, errors are marked with an uparrow under the offending symbol and the standard Jensen and Wirth error number, without further explanation. There is no way to disable the compilation when an error is encountered; you can't even turn off the assembly language or listing outputs. Heaven help you if you make a major goof (maybe a BEGIN without an end or an unmatched parenthesis) and have to watch helplessly as error message after error message was sent to the listing file and the program continued to crank out listing and assembly garbage.

My 800-line program ran reasonably well under UCSD Pascal and under MT+. However, an extensive effort was required to modify the program to fit in the MT+ compiler and linker. In contrast, the program compiled and linked almost immediately with Pascal/Z. Alas, when it came time to try it out, there were a succession of value range errors. Oops...if you don't set the $E compiler option in your source (extending the size of the resulting file slightly), you get the notice of the value range error without any hint of where it occurred in the program. You also need a listing, containing "statement numbers", or the count from the beginning of the program of Pascal statements that are code-producing rather than declarations...a handy reference so long as you have a current listing. (Changing around a few lines and then recompiling requires a new listing if you're going to find out where you are, since the line count is global.) Procedure names or numbers would be soooo nice!

However, once I found where the problem lay, from a review of the source it appeared impossible that the program error should have occurred. I made sure my loop variables were local and removed some set comparisons ("...IF i IN [1, 3..5] THEN..."), only to fall through to the next error after another round of compile-assemble-link 14 minutes later. Arrghh!

I tried adding the debugger to the code (which requires another 14-minute compile-assemble-link), only to discover that it wouldn't fit in memory with the linker. As it says in the manual under the discussion of "CODE OVERWRITES TABLES", the error message I received: "You lose. This is a fatal error."

I never did get my program to run.

Just for fun, I tried the peephole optimizer on my 800-line program to see what it would do. Nearly six minutes later, it reported to me that it had saved 85 bytes from the 23,000 byte code file. Sigh. But most of Pascal/Z's optimization is done within the compiler, and its code size and run-time speed are about equal to that of Pascal/MT+ and the other CP/M-80 compilers.

Some miscellaneous comments on the operation of the compiler: the listing was not helpful in determining how big each procedure was. The compiler could not tell me where any code would actually wind up; this would require an assembly listing with added Pascal source statement comments (available as an option). Actually determining the memory location of a procedure or function would require both an assembly listing and a linker map. The compiler does not accept a source line longer than 80 characters. Finally, I tried a trick to insert compiler options: nested include files (explicitly allowed). I compiled the file:

```
        {$L-,E+,P-}
        {$IMYFILE.PAS }
```

and discovered that the compiler ignored the INCLUDE directive, perusing only the two lines and giving me a "premature end-of-file" error.

Features and Faults

In contrast to MT+, which is loaded with features, the Pascal/Z compiler is fairly small. There is room to compile moderate-sized programs without running out of symbol table. Several Jensen and Wirth features have been omitted, including PACK, UNPACK and DISPOSE. These deletions reflect the fact that there are no packed variables under Pascal/Z (a serious problem if you're trying to manage disk space or diddle bits). Set sizes are fixed at 32 bytes or 256 elements, obviating the use of sets as a bit-manipulation technique. Heap management is a la UCSD Pascal prior to version IV.0 (i.e., garbage collection is up to you).

Limited constant expressions (e.g., MaxArry = 100; MaxAryM1 = MaxArry - 1) are allowed, a nice touch. There is an ELSE clause within CASE statements, also useful.

GET and PUT have also been deleted from Pascal/Z; all I/O is performed with READ and WRITE. I have no reason to doubt their claim that all you lose is access to the file window variable "f↑", having to read/write directly from some other declared variable. Still, it's nonstandard.

Random access to structured files has been implemented with particularly elegant syntax.

All reads from the console are done via the CP/M-80 line buffer, meaning that you cannot read a character without following it with a <RETURN> (or writing your own assembly-language BIOS calling facility). There is no excuse for this small but significant oversight.

There is a major problem which overshadows all of the other I/O features. There is no means whatever to tell if any disk access (opening a file, closing a file, creating a new file, reading or writing) was successful. Pascal/Z goes on its merry way, acting as if all was well - no error message, no IORESULT or other predeclared variable, NOTHING! Forget about serious applications programming, unless you're willing to dig around in the run-time library (full source *is* included) and create the fix.

There are useful two I/O features worth mentioning. At your option, whenever the user enters a Control-C, the program will abort and return to CP/M-80 (compare with Pascal/M, where if you're using the debugger, Control-\ restores control to the debugger, no matter what is happening, and tells you where you just were). Also, Pascal/Z is the only major microcomputer Pascal that prints out "TRUE" and "FALSE" when you go to writing boolean variables. In fact, at your option you can extend this ability to *read and write directly* any enumerated variable. Reading is totally case insensitive; writing always produces upper case.

You can chain from one Pascal/Z program to another (but not to other CP/M-80 programs); you must link the chain routines into both programs before running.

Complex boolean expressions are evaluated only so far as necessary for them to be satisfied. So, if you say "A or B" and A is true, B is not even looked at, since its value is irrelevant to the outcome. Similarly, if your statement is "A and B" and A turns out false, B is ignored. This feature allows you to avoid state variables or complex "IF THEN ELSE" statements. For example, if you are searching array "ARRY" for value "A", you can simply say

```
WHILE (Index <= ArrayLimit) AND (ARRY[Index] <> A) DO
   Index := Index + 1;
```

without having to protect ARRY[Index] from range error in case A is not in ARRY and Index becomes greater than ArrayLimit.

Compatibility with UCSD Pascal syntax is poor, particularly in the case of strings (Pascal/Z string functions are rudimentary).

There is a symbolic debugger that requires a limited number of cryptic two-letter commands to run it. It should not be terribly difficult to learn. Its use adds 12K to the resulting program, which will be a problem primarily when linking. As noted previously, I couldn't get it to fit in memory with my 800-line test program.

Pascal/Z and Pascal/MT+ share a common problem with exception handling. There is no convenient means to exit an arbitrary nest of procedures if an error or exception occurs. UCSD Pascal uses the EXIT statement (which allows virtually any procedure name as an argument); standard Pascal allows a GOTO to just about any place in the program. Large programs virtually always require some sort of mechanism to handle exceptions. The alternative to an effective EXIT or GOTO is to continually check a state variable indicating that no error has occurred. If an error or exception occurs, one must trickle program flow down a trail of exception detectors in each called procedure on the way to the exception handler.

Separate Compilation

I am enormously fond of creating procedures and functions that perform a simple task well (say, reading an integer from

(continued next page)

the console in an uncrashable manner) and that can be used anywhere, over and over, building upon each other. These "software tools" (see the excellent text of the same name by Kernighan and Plauger) must make no assumptions about their environment and must in turn be able to be maintained independently from host programs which use them, so that their mutual integrity is assured.

With version I.5, UCSD Pascal introduced an extraordinary mechanism for developing software tools, the UNIT. A UNIT is a separately compiled module intended for use by other programs, which contains an explicit, public INTERFACE section followed by an IMPLEMENTATION portion that is private and can be modified without disturbing the linkage to the outside world. When a UNIT is declared in a host program, the compiler automatically reads in the INTERFACE and treats it as if it were part of the host program declaration section, thus insuring accuracy, saving oodles of typing, and providing automatic type checking. Thus, not only can UNITs be maintained independently of their host programs, they can be sold to others in object form as software tools. (In fact, a booming business of this sort has already begun.)

Under the new IV.0 version of UCSD Pascal, UNITs have been expanded dramatically, with impressive results. Because UNITs and other program segments are loaded automatically and discarded after use by the operating system if there is insufficient room, true machine-independent memory management is now a reality. Virtually unlimited quantities of UNITs may be utilized by a program, and UNITs may use other UNITs to perform their functions. Machine-dependent operations such as the interface to the CRT console and modem drivers have been isolated into standard UNITs, providing further machine independence.

Like the other CP/M-80-based Pascals, Pascal/Z has a primitive external compilation facility. First, there is the ability to pull in external procedures and functions from a library or other externally assembled or compiled source. The name and parameter list of any such procedure or function is listed and declared as EXTERNAL before it is used in the program, in much the same way that you declare a procedure or function FORWARD. Accuracy and type checking are entirely up to you.

Second, you can break up your program into up to sixteen modules, with the host module containing at a minimum all global variable and type declarations plus the name and parameter list (again, declared as EXTERNAL) of any procedure or function you wish to access outside of its module (thus preserving a measure of independence of one module from another). When the main module is compiled, a list of its global types, variables, and procedure/functions is written to the disk, where it is available to the other modules. (This means that all the other modules must be recompiled whenever the main module changes.) The secondary modules may be modified and recompiled independently of each other, and then bound together with the linker. No type checking is done, however, between routines declared EXTERNAL within the host and their actual implementations. This facility is a moderately handy means of breaking up a large project into modules. It is useless, however, to implement software tools, since the creation and compilation of the tool should precede, not follow, that of the host.

In an extension of the ability to create modules, Pascal/Z also supports program overlays, with memory management performed by a program (OVLYGEN.COM) that analyzes each file and creates a SUBMIT file to drive the linker so the correct memory allocation is produced. The overlay must explicitly be brought into memory by the programmer, and there is only one overlay area allowed.

Some Suggestions

Improve the compile-time interface by allowing commandline compiler options and by printing lines containing errors (along with a descriptive message) on the console, with the optional ability either to abort compilation or to turn off the assembly language output or the listing if desired. You should be able to disable any and all outputs, just to fly through compilation and turn up errors. The listing should give some indication of procedure size, preferably with each line displaying the offset in bytes from the start of the procedure. The linker should generate a listing of where each procedure is located in memory.

Provide a means of detecting disk errors. Correct single character reads so that they are complete with one keypress.

I suggest that the compiler and assembler interact with an assembly file consisting of tokens, so that it is unnecessary to write and read twenty characters of ASCII per byte of output code. This change should shrink the size of the resulting assembly file by a factor of ten, much in the manner of Microsoft BASIC's .BAS files. Since eliminating the need to produce ASCII assembly language should shrink the size of the compiler, I'd like the option of using a stripped down version of the compiler in order to get more symbol table space. (In fact, I would be willing to use a separate utility to convert the token file into an ASCII assembly language file, since I'd rarely want to muck around with the assembly language output anyway.) A compressed assembly source should speed up both assembly and compilation dramatically.

Summary

It appears that Pascal/Z suffers from the same problem that plagues most software: incest (the people who write the program do most of the testing). The problems of poor user interface and marginal reliability should have been revealed through thorough check-out by a wide variety of programmers. Pascal/Z needs work. However, I feel that it would not be difficult to turn it into a truly useful development tool.

REPLY

Dear Ms. Mellin,

We would like to thank *Lifelines* for the opportunity to respond to Mr. Gagne's review of our Pascal/Z compiler prior to publication, and to have our comments printed in the same issue as the review. We appreciate the professionalism of *Lifelines*.

In general, the facts in Mr. Gagne's review are correct. There are a few exceptions, however.

(1) The large version of Pascal/Z requires 54K, not 56K.

(2) It is not impossible to halt compilation; the compiler can be stopped at any point during compilation by typing Control C. This will halt compilation and preserve all files output to that point, so the listing file will be available for information on errors.

(3) We do not simply give Jensen & Wirth error codes for all errors; the most common ones are given in English. There are also full descriptions of our error messages in the Pascal/Z User's Manual. We have implemented many of our error-checking features as compiler options in order to minimize code size as much as possible. The options can be enabled at any time to provide full diagnostics. With the E option (extended error messages enabled, not only does our compiler give the statement numbers in which the error occurred, but if using separate compilation, it gives the module number as well. We wish to thank Mr. Gagne for bringing the problem of $L- before the program heading to our attention. We will test this and document it accordingly.

(4) In reference to Mr. Gagne's problems with INCLUDE files, the problem is that he did not have a program heading at the beginning of the file.

(5) Almost all variables in Pascal/Z are automatically PACKed, therefore we have not implemented PACK and UNPACK. DISPOSE has been replaced by MARK and RELEASE in Pascal/Z.

A few things to which Mr. Gagne objected have been modified in version 4.1, to be released this summer. These include:

(1) Overlays may now have user-defined base addresses, allowing more than one overlay area.

(2) Better error messages are now given for operations involving disk I/O.

We object to certain inconsistencies in Mr. Gagne's review, as he seems to be largely biased towards U.C.S.D. Pascal. While he objects to the few nonstandard features of our compiler, he extols many nonstandard constructs of U.C.S.D. – indeed, most of the features Mr. Gagne finds most attractive in U.C.S.D. are totally nonstandard. U.C.S.D. is NOT the standard; in fact a standard for Pascal does not yet exist – there is just a draft proposal. Although we have deviated in a few instances, we have tried to adhere closely to the Jensen & Wirth definition and to maintain the spirit of Pascal. When a standard is adopted, we will make every effort to conform to it, while maintaining compatibility with previous versions of Pascal/Z.

We believe our compiler to be the highest quality Pascal compiler available for Z-80s, and many of our customers agree. Peter Grogono, author of the well-known PROGRAMMING IN PASCAL, states that:

"To the best of my knowledge it is the highest quality Pascal compiler available to users of microcomputers."

Other users have written:

". . . a superb software product! I've NEVER purchased software of this quality before."

"An excellent system that is getting better."

"All in all I think it's an excellent package – keep up the good work."

Several large companies are using our Pascal/Z for their applications programming, including Xerox Corporation, Jet Propulsion Laboratories, Naval Ocean Systems Center, and the National Bureau of Standards. Unlike Mr. Gagne, they seem to feel that Pascal/Z is ideally suited for the development of professional software products. We have benchmarks available for Pascal/Z and Pascal/MT+ testing for both execution speed and final code size; these are available to the public by writing to InterSystems.

At InterSystems we are dedicated to the support and improvement of all our software products, and appreciated feedback from our users. Therefore, although we disagree with some of his observations, we would like to thank Mr. Gagne for his time and effort.

We also thank *Lifelines* for your fairness in allowing us to comment.

Sincerely,

Laurie Hanselman Moskow
Software Products Manager
Ithaca InterSystems, Inc.
1650 Hanshaw Rd.
P.O. Box 91
Ithaca, N.Y. 14850

A Call For Manuscripts

Maybe you've written for publication before? Or perhaps you've always wanted to write? It could be that you've got some ideas you'd like to share with us. We're interested in hearing about your experiences. We like to publish both longer essays and those short gems which can hold so much important information. We pay competitively and our authors will tell you that writing for **Lifelines/The Software Magazine** is satisfying in many ways. We're particularly interested in material on using and modifying CP/M-80, CP/M-86, PC DOS, MS DOS, UNIX, XENIX and UCSD Pascal.

Send us a brief resume of your software experience, and samples of your previous writing, if you have any. (Don't be shy if you're **not** an experienced writer.) Then we can talk about your work and about payment for your efforts. Write or call: Editorial Dept., Lifelines Publishing Corporation, 1651 Third Ave., New York, N.Y. 10028. Tel.: (212) 722-1700.

A Query-Driven Erase Function

Thomas N. Hill

In the course of learning and using Digital Research's CP/M-80 operating system, many people become aware of minor inconveniences and annoyances which prevent CP/M-80 from being the perfect operating system. This article and the accompanying program listings are an attempt to relieve one of these annoyances, namely the problem of ERAsing files which cannot be easily referenced through the use of 'wildcard' characters. The program presented here was written after seeing a similar program in operation upon a system running under MP/M-80.

The MP/M-80 ERAQ program very considerately queries the user during erase operations concerning the advisability of erasing that particular file. My ERAQ performs the same function, but in addition it will also indicate whether the file in question has been marked as $SYStem or R/O. If the file is set to R/O and the user indicates that s/he wishes to erase it, the program will automatically remove the R/O attribute before erasing the file.

About the Program

The program was written for assembly using Microsoft's M80 Relocating Macro-Assembler, but should translate to other assemblers easily. I am not using any fancy macros and the only special feature which is used is the INCLUDE pseudo-op, which will cause the Macro-Assembler to read a .LIB file from the default disk. This makes life easier for lazy programmers (me included), since we only need to prepare commonly used assembly code once. Listing 2 is the text of the INCLUDEd file, EQUATES.LIB. Note that if you use one of Digital Research's assemblers (ASM or MAC), you must place an 'ORG 0100H' statement before the actual program code. The program performs all disk and console operations through standard BDOS function calls, and should therefore perform upon any CP/M-80 2.x system.

Listing 1

```
        TITLE ERAQ
        SUBTTL PROGRAM TO PROVIDE QUERY DRIVEN ERASE FUNCTION

        .COMMENT \

        WRITTEN BY: THOMAS N. HILL
                    ALASKA MICRO SYSTEMS
                    200 OKLAHOMA
                    ANCHORAGE, AK.   99504
                    (907) 337-1984  [0900 - 1700 (AST)]
        DATE WRITTEN: NOVEMBER, 1981
        MODIFICATION AND UPDATE LIST:
                11/12/81        VERSION 1.0 FINALIZED.
                11/20/81        UPPERCASE BUG FIXED ON ERASE QUERY.
                11/24/81        DEFAULT TO *.* ON NULL COMMAND LINE.
                12/08/81        SYS AND R/O DETECT ADDED.
                04/22/82        ATTRIBUTE BIT CLEARED BEFORE OUTPUT,
                                FOR VB-3, ETC.

        This program is patterned after the ERAQ function
        provided with Digital Research´s MP/M system.
        It provides the user with a query driven erase function which
        accepts an ambiguous file reference from the command line and
        scans the specified disk for matches to the file reference.
        All matches are displayed to the console and the operator is
        queried as to whether or not the file is to be erased.  Those
        files requested for erasure are then erased from the disk
        directory.  Files tagged as ´system´ or ´read-only´ are flagged
        as such when queried for erasure. \

        INCLUDE EQUATES.LIB

        ; additional equates

        SETATR  EQU     1EH     ; change file attribute bits

        ; program commences

        ;CSEG is default

ERAQ:   LDA     FCB1+1
        CPI     ´ ´             ; no file reference on command line?
        CZ      ALLFILE         ; no file reference, default to *.*
        LXI     D,DMABUF
        MVI     C,SETDMA
        CALL    BDOS
        LXI     D,FCB1
        MVI     C,SRCHF         ; find first occurrence of the requested file
        CALL    BDOS
        INR     A               ; return 0ffh if no match
        JZ      FIN
ERAQ0:  DCR     A
        ADD     A               ; offset to buffer address
        ADD     A
        ADD     A               ; there must be an easier way!
        ADD     A
        ADD     A
        LXI     H,DMABUF
        MVI     D,0
        MOV     E,A
        DAD     D               ; calculate offset
        XCHG
        LHLD    DIRPTR          ; get current table pointer
        MVI     C,32            ; move 32 bytes of directory entry
LP1:    LDAX    D               ; get a byte
        MOV     M,A
        INX     H
        INX     D
        DCR     C
        JNZ     LP1
        SHLD    DIRPTR          ; save updated table pointer
        LDA     DIRCNT          ; count entries found
        INR     A
        STA     DIRCNT
        LXI     D,FCB1          ; find the next entry
        MVI     C,SRCHN
        CALL    BDOS
        INR     A               ; done yet?
        JNZ     ERAQ0
        ; table now contains all selected directory entries.
        ; display them and request erase permission.

        LXI     H,BUFFER
        SHLD    DIRPTR          ; reset pointer to start of table
DISP0:  MVI     A,8             ; eight chars in name
        PUSH    H               ; save pointer to start of name
DISP1:  INX     H               ; over user byte
        CALL    SENDSTRG        ; print the string at (HL) for (A) bytes
        PUSH    H
        MVI     E,´.´           ; print a period
        CALL    OUTPUT
        POP     H
        MVI     A,3             ; three chars in type
LP2:    CALL    SENDSTRG

        ; now check attribute bytes

        MVI     E,´•´
        CALL    OUTPUT          ; print 2 spaces
        MVI     E,´ ´
        CALL    OUTPUT
        POP     H               ; start of name
        LXI     D,9
        DAD     D               ; first attribute byte
        MOV     A,M             ; get char
        RLC                     ; put attribute bit in carry
        JNC     DISP2
        MVI     A,TRUE          ; flag R/O file
        STA     ROFLG
DISP2:  INX     H               ; system attribute
        MOV     A,M
        RLC
        JNC     DISP3
        MVI     A,TRUE
        STA     SYSFLG          ; set system file flag
DISP3:  LDA     ROFLG           ; now see if any attributes are set
        MOV     B,A
        LDA     SYSFLG
        ORA     B
        JNZ     DISP7
        MVI     E,TAB
        CALL    OUTPUT
        JMP     DISP6
```

```
DISP7:  MVI     E,´(´
        CALL    OUTPUT
        LDA     ROFLG               ; print R/O symbol?
        ORA     A
        JZ      DISP4               ; nope.
        LXI     D,RO
        CALL    STROUT              ; print read/only symbol after file name
DISP4:  LDA     ROFLG               ; if printed R/O, then print comma
        ORA     A
        JZ      DISP8
        MVI     E,´,´
        CALL    OUTPUT
DISP8:  LDA     SYSFLG
        ORA     A
        JZ      DISP5               ; print system symbol?
        LXI     D,SYS
        CALL    STROUT              ; yes, print it after file name
DISP5:  MVI     E,´)´               ; close symbol string
        CALL    OUTPUT
DISP6:  LXI     D,MSG1              ; ask about erasing file
        CALL    STROUT
        MVI     C,CONIN             ; get answer
        CALL    BDOS
        PUSH    PSW                 ; save answer
        CALL    CRLF                ; print a crlf
        POP     PSW
        ANI     5FH                 ; make upper case
        CPI     03                  ; check for control-c abort
        JZ      FIN
        CPI     ´Y´                 ; if answer is yes, then erase
        CZ      ERASE
        MVI     A,FALSE
        STA     ROFLG               ; else clear flags and look for next match
        STA     SYSFLG
        LDA     DIRCNT              ; finished yet?
        DCR     A
        STA     DIRCNT
        JZ      FIN
        LHLD    DIRPTR              ; adjust table pointer
        LXI     D,32
        DAD     D
        SHLD    DIRPTR
        JMP     DISP0

        ; erase the file pointed to by the address stored in DIRPTR

ERASE:  LHLD    DIRPTR
        LDA     FCB1
        MOV     M,A
        LDA     ROFLG               ; check for R/O status
        ORA     A
        JZ      ERAS1
        PUSH    H                   ; save file pointer
        LXI     D,9
        DAD     D
        MOV     A,M                 ; change R/O attribute bit
        ANI     7FH
        MOV     M,A
        POP     H
        PUSH    H                   ; recover, resave pointer
        XCHG
        MVI     C,SETATR            ; alter file attributes
        CALL    BDOS
        POP     H                   ; recover saved pointer
ERAS1:  XCHG
        MVI     C,DELETF            ; erase the specified file
        CALL    BDOS
        LDA     ERACNT              ; count files erased
        INR     A
        STA     ERACNT
        RET

FIN:    CPI     03                  ; was it an abort?
        CZ      FIN9
FIN0:   LDA     ERACNT              ; how many did we erase?
        MVI     H,0
        MOV     L,A                 ; convert to decimal value
        LXI     B,-100
        CALL    CNVRT
        MOV     A,E
        CPI     0                   ; check for leading zeros
        JZ      FIN1
        CALL    PRTDIG              ; print the digit
FIN1:   LXI     B,-10
        CALL    CNVRT
        MOV     A,E                 ; check for leading zeros
        CPI     0
        JZ      FIN2
        CALL    PRTDIG
FIN2:   MOV     A,L                 ; one digit
        CALL    PRTDIG              ; print it, even if it is zero
        LXI     D,MSG2
        MVI     C,PRTBUF
        CALL    BDOS                ; print the message after the value
        JMP     CPM                 ; return to operating system

FIN9:   LXI     D,MSG3
        MVI     C,PRTBUF
        CALL    BDOS                ; indicate an abort
        RET

        ; print the string at (HL) for (A) bytes to the console

SENDSTRG:
        PUSH    PSW
        PUSH    H
        MOV     A,M                 ; pick up char
        ANI     7FH                 ; clear MSB
        MOV     E,A
        CALL    OUTPUT              ; print it
        POP     H
        POP     PSW
        INX     H
        DCR     A                   ; done yet?
        JNZ     SENDSTRG
        RET


CNVRT:  MVI     E,-1
CNVRT1: INR     E
        DAD     B                   ; subtract power of ten from binary value
        JC      CNVRT1              ; do it until we go negative
        MOV     A,B
        CMA                         ; now add back last value subtracted
        MOV     B,A
        MOV     A,C
        CMA
        MOV     C,A
        INX     B                   ; make sure we two´s complement it
        DAD     B                   ; add it back in
        RET

PRTDIG: ADI     ´0´                 ; convert to ASCII
        PUSH    H
        MOV     E,A
        CALL    OUTPUT              ; print it
        POP     H
        RET

ALLFILE:
        LXI     D,MSG4
        MVI     C,PRTBUF
        CALL    BDOS                ; check for go-ahead
        MVI     C,CONIN
        CALL    BDOS
        PUSH    PSW
        CALL    CRLF
        POP     PSW
        ANI     5FH                 ; make upper case
        CPI     ´Y´                 ; if Y then proceed, else abort
        JNZ     FIN9
        LXI     H,FCB1+1            ; leave disk number intact
        MVI     B,11
ALL1:   MVI     M,´?´               ; make FCB all qmarks
        INX     H
        DCR     B
        JNZ     ALL1
        MVI     B,24
ALL2:   MVI     M,0                 ; and rest with zeros
        INX     H
        DCR     B
        JNZ     ALL2
        RET

CRLF:   MVI     E,CR
        CALL    OUTPUT
        MVI     E,LF
        CALL    OUTPUT
        RET

OUTPUT: MVI     C,CONOUT
        CALL    BDOS                ; print file name
        RET

STROUT: PUSH    H
        PUSH    D
        MVI     C,PRTBUF
        CALL    BDOS
        POP     D
        POP     H
        RET

        ; messages

MSG1:   DB      TAB,´Erase this file? $´
MSG2:   DB      ´ files erased.´,CR,LF,´$´
MSG3:   DB      ´Program aborted.´,CR,LF,´$´
MSG4:   DB      ´Use *.* as file match? $´
RO:     DB      ´R/O$´
SYS:    DB      ´SYS$´

        ; data

DIRPTR: DW      BUFFER
DIRCNT: DB      0                   ; count of selected directory entries
ERACNT: DB      0                   ; count of erased files
ROFLG:  DB      0                   ; set initial false
SYSFLG: DB      0
DMABUF: DS      130
BUFFER  EQU     $                   ; start of directory buffer.

        END
```

Listing 2

```
; SYSTEM EQUATES

CPM     EQU     0
TPA     EQU     CPM+100H
BDOS    EQU     0005H    ; BDOS ENTRY POINT
FCB1    EQU     005CH    ; CP/M FILE CONTROL BLOCK
FCB2    EQU     006CH    ; SECOND FILE CONTROL BLOCK
CBUF    EQU     0080H    ; DEFAULT COMMAND BUFFER

; 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
SIZEF   EQU     35       ; COMPUTE FILE SIZE
SRCHF   EQU     17       ; search for first ambiguous file
```

(continued next page)

```
SRCHN   EQU     18      ; serach for next ambiguous file

; 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 OFFH INDICATES FAILURE.

; character equates

CR      EQU     ODH         ; carriage return
LF      EQU     OAH         ; 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
;
FALSE   EQU     00H
TRUE    EQU     OFFH
;
```

Notice

The August issue was placed into the mail on July 27th. If you had any problem with the timeliness of this issue, please call our Subscription Department at (212) 722-1700, or write to **Lifelines/The Software Magazine** Subscription Department, 1651 Third Ave., New York, N.Y. 10028. We expect to place this issue, dated September 1982, into the mail around August 25th. Each month we will print the date of the previous issue's mailing and would appreciate your help in tracking the deliveries.

Volume 84, Catalogue and Abstracts

CP/M Users Group

Catalogue

DESCRIPTION: MODEM version 7.65
XMODEM version 5.0

NUMBER	SIZE	NAME	COMMENTS
		-CATALOG.084	CONTENTS OF CPMUG VOL. 84
		ABSTRACT.084	Abstracts of files on this disk
		CRCK.COM	Program for checking CRC's of files
		CRCKLIST.084	CRCK's of files on this disk
084.1	16K	MODEM7.DOC	Documentation for MODEM program [from CPMUG Vol. 79]
084.2	14K	MODEM76.LIB	Macro library used with MODEM 7.65
084.3	6K	MODEM76.SET	Instructions for "hot-patching" MODEM
084.4	63K	MODEM765.ASM	Macro assembler source code for MODEM 7.65
084.5	10K	MODEM765.COM	Object code of MODEM 7.65
084.6	18K	SEQIO22.LIB	Macro library used with XMODEM 5.0
084.7	3 K	XMODEM47.DOC	Documentation for XMODEM program
084.8	49K	XMODEM50.ASM	Assembler source code for XMODEM 5.0
084.9	1K	XMODEM51.FIX	Notes on a bug fix for XMODEM

Note: MODEM program requires assembly with Digital Research MAC macro-assembler. XMODEM may be assembled with ASM.COM if the "logging" feature is disabled, otherwise MAC is required for assembly.

Abstracts

084.1	**16K**	**MODEM7.DOC**
084.2	**14K**	**MODEM76.LIB**
084.3	**6K**	**MODEM76.SET**
084.4	**63K**	**MODEM765.ASM**
084.5	**10K**	**MODEM765.COM**

These files comprise the MODEM package, latest revision. Digital Research's MAC macro-assembler is required for assembly. This version incorporates many bug fixes over the version released in CPMUG Vol. 79. Most users of CP/M-80 will be familiar with the MODEM program, originally written by Ward Christensen, now greatly modified and enhanced by a variety of programmers. For those not familiar with MODEM: this program, once customized to your particular system hardware, will allow your system to act as a computer terminal over the telephone lines. In addition, MODEM uses a checksum/CRC block transfer method of sending and receiving disk files over the phone lines, with autoretry on error, error reporting, counting, etc. It is a widely endorsed and used program which soon becomes an essential tool for the serious microcomputer user. [Note: Modem hardware *is* required].

084.7	**18K**	**SEQIO22.LIB**
084.8	**3K**	**XMODEM47.DOC**
084.9	**49K**	**XMODEM50.ASM**
084.10	**1K**	**XMODEM51.FIX**

These files comprise the XMODEM package, a utility found on Remote CP/M-80 (RCPM) systems. XMODEM uses the same protocol as MODEM, but is designed for sending and receiving files only, with greatly simplified commands. Version 5.0 implements a "logging" feature which "logs" the files sent and received, the date and time (assuming your RCPM uses a clock/calendar of some sort), and the name of the user sending or receiving (user data will be expected in the file "LASTCALR" which is created by both the RBBS and SIGNON systems). This feature, if enabled, will require assembly with Digital Research's MAC macro-assembler. If this feature is disabled, ASM.COM may be used. If the LASTCALR file is not found, no logging will take place.

Jim Mills, CPMUG Reviewer

Feature

A Preview of Supersoft's Ada Compiler

Steve Patchen

Ada Release 1.00a
by Dave Norris

distributed by;
Supersoft, Inc.
P.O. BOX 1628
Champaign, Il 61820

The first release of this compiler is a small subset of the Department of Defense's definition for Ada. Supersoft says they plan eventually to implement the full compiler. The current release lacks a number of features essential for many serious applications. Instead of attempting a full review of a product likely to change considerably over the next several months, I will demonstrate some of those features of Ada which *are* present and discuss related missing features. To this end I have written a simple screen editor which incorporates some of the structures I will discuss.

There are two basic areas I have addressed with this program (see source listing): the console terminal device and string manipulations. In this and possible future articles I will also be keeping an eye open for weaknesses in Ada (other than those pointed out in July's article) for use in business applications.

Discussion Of Features

The sample program first defines four console control strings. These strings are first defined as strings of blanks and later loaded by the procedure screen_init with values for the appropriate terminal. An enhancement would be to load these parameters from a terminal configuration file. Next, several console keyboard function keys are defined. The rest of the data definitions define the screen configuration used and provide program variables. The declared procedures supply cursor manipulation for screen editing.

Strings in Ada are fixed length one dimensional character arrays similar to Pascal strings. Ada, however, also defines a SLICE, which is a one dimensional array denoting a sequence of consecutive components of a one dimensional array (slice ::=array_name(discrete_range)). The discrete range must be either a null range or possible index values for the named array. This slice feature facilitates sub-string manipulations like searches and matching and partial substitutions. Since Supersoft's current release doesn't include this feature, character-by-character manipulations must be performed. This makes such operations noticeably slower. For instance, the for loop in the screen_init routine was first tried with a nested for loop which did character-by-character loading of spaces to the array. This produced a significant delay in loading and starting the program.

Since all strings are fixed in length, routines like put(string) send all the characters in the array to the terminal. If only a few of the characters in the string are significant, the programmer has to provide his or her own procedures for displaying and manipulating the significant part of strings. I have done this with the terminal control strings. The routine putscreen() stops sending characters to the terminal if it encounters the null character in the string. The attributes defined by Ada to allow a program to determine the length of a string are missing from this release also. This means all string manipulation procedures have to know the length of the strings they will handle and cannot determine this on their own. There is also a problem with mixing string arrays and character arrays because this version does not allow assignment of string literals to character arrays. Assignments must be made character-by-character. String concatenation has also been omitted from the first release, forcing character-by-character building of resultant strings. Indices, however, are limited only by integer size (-32768..32767) and memory availability.

The character() function is complimented by the integer() function for converting integers to characters and vice versa. I attempted to provide a feature for printing the screen on the printer, but ran into a bug in the bdos() function. This bug has been corrected on the version currently being distributed.

Another inconvenience I ran into was that all the files had to be on the same device: the source file intermediate files and the resultant .COM file. There are three other limitations which restrict the application of this compiler. Only seven digit single precision numbers are provided in addition to integers; file reading and writing can only be done sequentially, and there is no trace or debugging facility provided for locating errors discovered while running programs. The compiler does have good compiler error messages though.

The compiler comes with several application examples, most of which are games. There is also a primitive calculator program. It has a poor terminal interface, but the strategy used in this editor could be used to make the calculator operational on more terminals.

In Summary

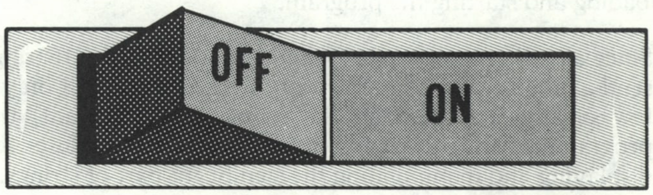
Although release 1 of Ada has very few features, it can perform simple applications like the calculator and simple screen editors. The lack of random access will limit its use for data entry and report generation.

Source Code For Sample Program

```
-- 07/18/82
-- copyright 1982 by Steve Patchen
-- A test routine for screen entry
pragma print(on);

procedure typer is


-- terminal dependent features

    clear_screen : string(0..7) := "        ";
    home_cursor : string(0..7) := "        ";
    erase_end_line : string(0..7) := "        ";
    cursor_xy : string(0..7) := "        ";
    offset : constant integer := 32;

-- keyboard command functions
```

(continued next page)

```
    cr : constant character := character(13);
    back_space : constant character := character(8);
    backspace : constant character := character(19);
    rubout : constant character := character(127);
    print_cmd : constant character := character(16);
    esc : constant character := character(27);
    erase_line : constant character := character(25);
    insert_line : constant character := character(14);
    forspace : constant character := character(4);
    tab : constant character := character(9);
    backtab : constant character := character(1);
    up_line : constant character := character(5);
    down_line : constant character := character(24);

-- bdos equates

        lst : constant integer := 5;

-- program constants

    zero_col : constant integer := 0;
    top_row : constant integer := 1;
    length : constant integer := 79;

    input : character;
    col_pos,row_pos,DE : integer := 0;
    error : boolean := false;

    screen : array(1..23) of string(0..79);
    title : string(0..79)        := "  ------------------- THIS IS A TEST TITLE LINE ----------------
    null_string : string(0..79) := "

-- this initialization routine contains terminal dependent constants

    procedure screen_init is
        begin
        clear_screen(0) := character(27);
        clear_screen(1) := ´H´;
        clear_screen(2) := character(27);
        clear_screen(3) := ´J´;
        clear_screen(4) := character(0);
        home_cursor(0) := character(27);
        home_cursor(1) := ´H´;
        home_cursor(2) := character(0);
        erase_end_line(0) := character(27);
        erase_end_line(1) := ´K´;
        erase_end_line(2) := character(0);
        cursor_xy(0) := esc;
        cursor_xy(1) := ´Y´;
        cursor_xy(2) := ´ ´;
        cursor_xy(3) := ´ ´;
        cursor_xy(4) := character(0);
        for i in 1..23 loop
                    screen(i) := null_string;
        end loop;
        end screen_init;

-- This routine prints strings terminated by a null to the terminal

    procedure putscreen(c : string) is
        n : integer := 0;
        begin
          while n <= 7 and c(n) > character(0) loop
                put(c(n));
                n := n + 1;
          end loop;
        end putscreen;

    procedure put_cursor(y, x : integer) is
        begin
                cursor_xy(0) := character(27);
                cursor_xy(1) := ´Y´;
                cursor_xy(2) := character(y + offset);
                cursor_xy(3) := character(x + offset);
                cursor_xy(5) := character(0);
                putscreen(cursor_xy);
        end put_cursor;

    procedure next_pos is
        begin
                col_pos := col_pos + 1;
                if col_pos > 79  then
                        col_pos := 0;
                        row_pos := row_pos + 1;
                        if row_pos > 23 then
                                row_pos := 1;
                        end if;
                end if;
        end next_pos;

    procedure back_pos is
        begin
           col_pos := col_pos - 1;
           if col_pos < 0 then
                col_pos := 79;
                row_pos := row_pos - 1;
                if row_pos < 1 then
                        row_pos := 1;
                        col_pos := 0;
                end if;
           end if;
        end back_pos;

    procedure clear_line(x : integer) is
        begin
                col_pos := 0; row_pos := x;
                put_cursor(row_pos,col_pos);
                putscreen(erase_end_line);
                screen(x) := null_string;
        end clear_line;

    procedure move_lines( x : integer) is
        begin
                for i in reverse 22..x loop
                        screen(i+1) := screen(i);
                end loop;
        end move_lines;

    procedure display( r : integer) is
        begin
                putscreen(home_cursor);
                for i in r..23 loop
                    put_cursor(i,zero_col);
                    putscreen(erase_end_line); put(screen(i));
                end loop;
        end display;


    begin
      screen_init;
      putscreen(clear_screen);
      put(title);newline;
      row_pos := 1; col_pos := 0;
      put_cursor(row_pos,col_pos);

      loop
        -- get an input character
        get(input);
        case input is

            when back_space | backspace =>
                back_pos;

            when forspace =>
                next_pos;

            when backtab =>
                if col_pos < 8 then
                        col_pos := 0;
                        back_pos;
                else
                        if col_pos rem 8 = 0 then
                                col_pos := col_pos - 8;
                        else
                                col_pos := (col_pos/8)*8;
                        end if;
                end if;
            when tab =>
                if col_pos > 71 then
                        col_pos :=79;
                        next_pos;
                else
                        col_pos := (col_pos/8)*8+8;

              end if;
            when cr =>
              row_pos := row_pos + 1;
              col_pos := 0;
              if row_pos > 23 then
                      row_pos := 1;
              end if;

            when down_line =>
              if row_pos = 23 then
                      row_pos := top_row;
              else
                      row_pos := row_pos + 1;
              end if;

            when up_line =>
              if row_pos = 1 then
                      row_pos := 23;
              else
                      row_pos := row_pos - 1;
              end if;

            when esc =>
              bdos(0);

            when erase_line =>
              clear_line(row_pos);
            when insert_line =>
              move_lines(row_pos);
              clear_line(row_pos);
              display(row_pos);

            when print_cmd =>
              for i in 1..23 loop
                      put(screen(i));
                      for j in 0..79 loop
                              DE := integer(screen(i)(j));
                              -- bdos(lst,DE); bdos has a bug in first version
                      end loop;
                      newline;
              end loop;
              putscreen(clear_screen);
              put(title);
              display(top_row);
              put_cursor(row_pos,col_pos);

            when others =>
              screen(row_pos)(col_pos) := input;
              next_pos;

        end case;
        putscreen(home_cursor);put("      ");
        putscreen(home_cursor);put(row_pos);put(´-´);put(col_pos);
        put_cursor(row_pos,col_pos);

      end loop;
    end typer;
```



Feature

8080 Assembler Programming Tutorial: Subroutines, Part 4

Ward Christensen

Disk I/O

This month's tutorial will deal with CP/M-80 disk input and output, including directory searches, making, opening, reading, writing, closing, and erasing files.

Background

Rather than just saying "this is how you open and read a disk file", I'll start with some background, working up from disk layout, to directory layout and file allocation – all the background you should need to be comfortable working with files.

Disk Layout

Floppy and hard disks consist of magnetic material. The read/write head of the disk drive can move to access one of many concentric circles on the disk, called *tracks*. Each track is further divided into several segments, called *sectors*.

On a standard 8" single density CP/M-80 floppy disk, there are 77 tracks with 26 128-byte sectors per track. While there are exceptions, double density disks usually have 26 256-byte sectors, 15 or 16 512-byte sectors, or 8 1024-byte sectors. Double sided drives double this disk capacity.

By convention, tracks are numbered starting at 0, and sectors starting at 1. I have heard a plausible explanation for this: early IBM disks had a special record called "record 0" on each track, containing system information. Thus the first actual user data record was number 1.

There is virtually no limit to the allowable layout for a disk which CP/M-80 can access. There is an upper limit on the total size of a single disk – 8 megabytes (8,338,608) which CP/M-80 2.x can handle. However, larger disks may be broken up into smaller *logical* disks, each of which is then limited to 8 MB. For example, I broke my hard disk into many logical disks, ranging in size from 248K to 2.2 MB. One reason this was that a DIR of an 8 MB disk would simply be overwhelming.

Allocation

The first part of a disk is usually reserved for CP/M-80 use. In a standard 8" single density disk, track 0 sector 1 is the "boot sector", i.e. the one that the hardware reads in. The rest of track 0 and part of track 1 contain CP/M-80 itself: CCP, BDOS, and BIOS. The boot sector reads this in. The directory starts on track 2.

For data storage, CP/M-80 divides the remainder of the disk into logical blocks, typically 1K (8 sectors) in single density, or 2K in double density. Hard disks (or even double sided floppies) might have 4K or larger blocks.

These blocks are numbered starting at the directory, with the first block of the directory being numbered 00. On single density 8" disks, the directory occupies 16 sectors, or two blocks. Thus the first file on an empty disk is placed in block 02. If the file is longer than 8 sectors, it overflows into block 03. See FCB bytes 16-31 below, to see where this information is kept for a file.

On single density disks (or in general any disk containing fewer than 256 blocks) a single byte pointer is used. If a disk has more than 255 blocks, two bytes are used.

SCRAMBLING: To obtain relatively good speed accessing files, CP/M-80 usually "scrambles" the sectors on a track. If it didn't, you would read one sector, but when you go back to read the next, you might have just missed it, meaning you have to wait another disk revolution (1/6 second on 8" floppies) to read it. The standard scrambling for CP/M-80 is every 6 sectors. Thus, rather than reading an 8" floppy sectors 1, 2, 3... you read sectors 1, 7, 13, 19, 25, 5, 11, 17, 23, 3, 9, 15, 21, 2, 8, 14, 20, 26, 6, 12, 18, 24, 4, 10, 16, and 22.

You don't need to know this, unless some time you want to figure out *exactly* what sectors a particular file occupies.

BIOS

CP/M-80's Basic Input/Output System provides the actual disk I/O by performing the functions of disk select, track select, sector select, setting the address, and then either reading or writing.

The BIOS is hardware-dependent upon the particular machine in which CP/M-80 is running.

In CP/M-80 2.x, the BIOS contains tables describing the disk format. In CP/M-80 1.4, this information was contained in BDOS, and thus not readily accessible for changing.

BDOS

Recall from last month, that the CP/M-80 Basic Disk Operating System does the "logical" I/O to the console, disks, printer, reader and punch. BDOS deals with the disk in terms of named files.

CP/M-80 Files

In most circumstances (and specifically in this tutorial) you need not be concerned about the disk layout. Instead, you

(continued next page)

will deal with the disk through CP/M-80's BDOS, as *named files*.

DIRECTORY: A file is identified by an 8-byte file name and a 3-byte file type, stored in the *directory*. Each directory entry takes 32 bytes, so there are four per sector. Single density typically has 64 directory entries, double has 128, and double density double sided may have 256. Hard disks – well there is no practical limit.

In single density, each directory entry keeps track of up to 16K of a file (128 sectors). In some larger disk formats, particularly hard disks, a directory entry may keep track of more than 16K.

EXTENTS: If a file is more than 16K, it is necessary to use *another* directory entry for it. Each subsequent entry is called an *extent*. CP/M-80 automatically maintains the extent as an additional byte in the directory immediately following the filename-filetype. Generally, you need only initialize the extent to 00H before working with a file.

SECTOR I/O: BDOS calls provide I/O at the *sector level*, i.e. there is no built-in reading of "fields" or "characters". However, this month's sample program will do some byte-at-a-time reading of a disk file, and next month I'll cover some more generalized subroutines for reading a byte and writing a byte.

File Control Blocks (FCB)

To perform disk I/O, CP/M-80 must know the name of the file you are using. It does so when you point the DE register pair to a file control block. The layout of the FCB is covered in the *CP/M Interface Guide*, should you ever need to quickly find it.

When you type a command to CP/M-80, any name you place on the command is automatically formatted into a *system FCB* at location 5CH. If you typed a second name, that name is formatted into location 6CH.

Most commands can use the FCB at 5CH as filled in by CP/M-80. If you have given a second name (or perhaps options for the program), CP/M-80 will have placed it at 6CH, so you will have to move them from 6CH (or at least process them) before actually using the FCB at 5CH, since once used, the system FCB occupies 5C-7C (or -7F if using CP/M-80 random file commands).

An FCB must be 33 bytes long, or 35 if doing random reads using the new CP/M-80 2.x random I/O facilities. I will not discuss this ability for the time being, as it is infrequently used.

Here is the FCB layout, as it should be in memory for use in accessing files. A number (e.g. 0) or number range (e.g. 1-8) specifies the displacement within the FCB for the given byte(s). The addresses following give the standard addresses for the system FCB which starts at 5CH – more about that later.

0 (5CH) identifies the disk you are referring to. If it is a 0, then you are referring to the *currently logged in disk*, i.e. the one given in the prompt (such as A>). However, since there is a BDOS call to change the logged in disk, it may be that the logged in disk is not the same as the one printed in the most recent prompt. To refer to specific disks, specify "A:" or "B:" (etc.) on the filename(s) given in a command, or use 01 for A, 02 for B, etc. to explicitly refer to a disk.

1-8 (5DH-64H) is the file name in upper case letters, left justified and padded with spaces.

9-11 (65H-67H) is the file type in upper case letters, left justified and padded with spaces.

12 (68H) is the extent. Usually, just set it to 00H. On rare occasions, you might want to *not* open the file at the beginning, and in this case select the appropriate extent: 00 if the sector wanted is in the first 128 sectors, 01 if in the second 128, etc.

13-14 (69H-6AH) are reserved bytes; set them to 00H.

15 (6BH) is the file size in sectors, if this is a one-extent file. If the file is longer than one extent, then it is the length represented by *this* extent. A full extent is typically 128 (80H).

16-31 (6CH-7BH) is 16 bytes where CP/M-80 keeps track of where the file is. These 16 bytes are the numbers of the blocks, as mentioned above under *ALLOCATION*. You need not worry about these bytes.

An aside on CP/M-80 disk allocation: CP/M-80 knows where it can place new files through a bit map in memory, with each bit corresponding to a block. The map is built when you "log in" a disk. CP/M-80 sets up the bit map showing all blocks available, then "zaps" off the bits corresponding to the directory. It then reads the directory, looking at the block pointers for each file, and turns off the bits used by files. Then when another file is opened for writing, CP/M-80 finds an "open bit" in memory. That way, there is no record on disk of free space which could get screwed up, i.e. be inconsistent with the actual files. The bit map is dynamically determined each time a disk is logged in. More aside: this bit map is what STAT looks at to determine how full the disk is. For example, if it finds 27 open bits in the bit map, and the disk is double density using 2K blocks, STAT reports 54K free on the disk.

32 (7CH) is the current sector number to read or write. Start it at 00.

Now, let's look at some typical FCBs as they appear after a command to the CP/M-80 A> prompt has been entered. To do so, I'll use a program I wrote called Q, which simply dumps memory from 00 to FF, so I can see what FCBs look like. Also, whatever you typed as part of the command (minus the command name itself) is stored character-by-character at 80H, so Q helps me see what this "raw buffer" looks like, too. Incidentally, if the operand of Q starts with an "@", then the operand is taken to be an address to dump from. Thus "Q @200" dumps memory starting at 200. Q.COM itself takes 2 sectors, from 100H to 200H, when it runs.

I have included a listing of Q.HEX for you to type in and LOAD, to learn more about FCBs (see *Listing 1*). Edit a file called Q.HEX, type in the lines exactly as they appear, end the edit, then type "LOAD Q". This will create Q.COM. Since the HEX file contains a checksum to validate the data, the LOAD program will abort if you have made any typing mistakes. The only likely error it wouldn't catch would be an entire missing line.

Here are some sample executions, showing only enough of 5CH's contents so that you can see what was stored in the buffer starting at 80H. Note that "garbage" exists in this buffer, but that address 80H contains the length of *valid* data. I did these at different times and in different orders, so don't try to make sense of the "garbage".

Let's look at one with no operands:

```
A>q
      00       04       08       0C

0050                             00202020  |            .   |
0060  20202020 20202020 00000002 00202020  |        .....   |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  00008601 3E2ECD36 01237DE6 0FC27901  |....>..6.#}...y.|
```

5CH is 00, showing that we specified no disk. The file names stored at 5DH and 6DH are blank. (I put the 00, 04, 08, 0C in just this once, to help you realize how the columns are numbered.)

Let's put just a disk specifier in the command:

```
A>q c:

0050                             03202020  |            .   |
0060  20202020 20202020 00000002 00202020  |        ....    |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  0320433A 002E0D0A 1A000000 00000000  |. C:............|
```

5CH contains the 3, meaning we explicitly requested drive C.

Let's try another, with two file names.

```
A>q test.file sample.out

0050                             00544553  |            .TES|
0060  54202020 2046494C 00000002 0053414D  |T    FIL.....SAM|
0070  504C4520 204F5554 00000000 00DB03C4  |PLE  OUT........|
0080  15205445 53542E46 494C4520 53414D50  |. TEST.FILE SAMP|
0090  4C452E4F 55540073 2C206F72 8D0A7175  |LE.OUT.s, or..qu|
```

5CH is zero, showing again no disk specified. The first name is formatted into 5CH, the second into 6CH. Note how the filename was left justified, and the filetype was placed after the 8-byte filename.

Now, two names, one without an explicit disk, the other with. Just for fun, I'll log in drive B:

```
A>b:
B>a:q foo.zot b:holy.cow
CB06

0050                             00464F4F  |........T....FOO|
0060  20202020 205A4F54 00000002 02484F4C  |     ZOT.....HOL|
0070  59202020 20434F57 00000000 00DB03C4  |Y    COW........|
0080  1320464F 4F2E5A4F 5420423A 484F4C59  |. FOO.ZOT B:HOLY|
0090  2E434F57 00636F77 0D0A1A00 00000000  |.COW.cow........|
```

The first name, with no explicit disk, resulted in 00H at 5CH. The second, with its explicit disk, resulted in 02H at 6CH. Nothing was affected by the fact that I had logged in B first, nor by my executing Q.COM off the A: disk. Byte 5C was still a 00, meaning "the currently logged in disk".

What if a name is too long? It is truncated:

```
A>q long.filetype

0050                             004C4F4E  |            .LON|
0060  47202020 2046494C 00000002 00202020  |G    FIL.....   |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  0E204C4F 4E472E46 494C4554 59504500  |. LONG.FILETYPE.|
```

Note how "filetype" was truncated and placed in the FCB as "FIL".

Now for an oddity: Placing certain special characters in a name has interesting results. Just as "." tells it to stop scanning the filename, and start looking for the filetype, several other special characters tell it to stop scanning all together.

```
A>q this=that

0050                             00544849  |            .THI|
0060  53202020 20202020 00000002 00202020  |S       .....   |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  0A205448 49533D54 48415400 53414D50  |. THIS=THAT.SAMP|
```

Only "THIS" is stored in the FCB. The "=" told CP/M-80 to stop handling the filename. Other characters that do this are: "—", ":", "<", ">", and ";". The latter is a comment symbol. Special characters, such as tabs, are rejected if you try to make them part of a filename.

If a special character such as a tab *is* necessary as part of a command, it must be *after* what CP/M-80 would treat as the two possible filenames.

For example, I have a simple variable-length record sort, to which you specify delimiter characters for the sort field, following the name of the file to sort. If I wanted to skip over a % sign in each record, I'd just say: sv filename @ %

The "@" means "at", then "%" specifies "at which character" the sort is to begin in each line of the file.

I used my Q.COM to learn how the arguments to SV would look. Let's go back and "look over my shoulder" at what I tried:

```
A>q filename @%

0050                             0046494C  |            .FIL|
0060  454E414D 45202020 00000002 00402520  |ENAME   .....@% |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  0C204649 4C454E41 4D452040 25007375  |. FILENAME @%.su|
```

SV scans for the "@" by looking at 80, to see that the number of characters typed is not zero, then scans from 81 on to look for an @ sign. If a 00 byte is found before finding an @, SV knew there was no @. I could have just looked in the second FCB (at 6D) for the @, but characters which don't apply to a filename might have been entered, such as ".", or ":", which then wouldn't appear in the filename. Thus using the "raw data buffer" at 80 is the best technique. (Also, SV allows specifying an output filename in case you don't want the file sorted and written back upon the input file. This output file would be in the second FCB, so I *must* use the technique of scanning the buffer at 80.)

Now, let's try passing a tab as the delimiter: @-sign tab:

```
                    A>q filename @
                    filename @      ?
```

CP/M-80 rejected it, typing back the name, the @, and the tab, followed by a "?".

I got around that by simply specifying a single "." as the second filename. Then the "@-sign tab" may be typed without CP/M-80 trying to make a filename out of it:

```
A>q filename . @

0050  D3F8E3E3 DBFD07DA 5400D500 0046494C  |........T....FIL|
0060  454E414D 45202020 00000002 00202020  |ENAME   .....   |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  0E204649 4C454E41 4D45202E 20400900  |. FILENAME . @..|
```

That was easy. I can just scan 81H on for my "@", and I'll find my "@-sign tab" at 8DH.

How about "*" or "?" in filenames? Well, "?" gets passed on as is, but "*" gets expanded into enough "?" to fill the remainder of the filename or filetype:

```
A>q a*.com

0050                             00413F3F  |........T....A??|
0060  3F3F3F3F 3F434F4D 00000002 00202020  |?????COM.....   |
0070  20202020 20202020 00000000 00DB03C4  |        ........|
0080  06202A2E 434F4D00 30303030 30303030  |. *.COM.00000000|
```

(continued next page)

It is considered good programming to check a filename for question marks before accepting it and going on with the program. Few early CP/M Users Group programs do this, which can have bad effects. The IDUMP program which I'll introduce in this tutorial, will screw up a CP/M-80 1.4 directory entry if something like "IDUMP *.ASM" is typed. The "????????.ASM" will match the first ".ASM" file, but if the file is over 16K, CP/M-80 will go back to the directory, overlaying the *real* name with "????????"! (CP/M-80 2.x doesn't update the directory if it detects that only *reads* have been done).

Enough on FCBs. Let's get to the meat of CP/M-80 file I/O. But first, we'll discuss the EQUates to be used as a consistent way to reference things.

BDOS Equates

Of the table of equates I presented in the tutorial in July *Lifelines*, this one will deal with:

```
OPEN     EQU     15        ;Open a file, FCB in (DE)
CLOSE    EQU     16        ;Close a file, FCB in (DE)
SRCHF    EQU     17        ;Search for first file, FCB in (DE)
SRCHN    EQU     18        ;Search for next file, FCB in (DE)
ERASE    EQU     19        ;Erase a file, FCB in (DE)
READ     EQU     20        ;Read a file sector, FCB in (DE)
WRITE    EQU     21        ;Write a file sector, FCB in (DE)
MAKE     EQU     22        ;Make a new file, FCB in (DE)
REN      EQU     23        ;Rename a file
SETDMA   EQU     26        ;Set the DMA (Disk I/O) addr. (DE)
;
;Here are a few other equates which you will find useful:
;
BDOS     EQU     5         ;entry point to BDOS
FCB      EQU     5CH       ;Main file control block address
FCB2     EQU     6CH       ;Second file control block address
FCBEXT   EQU     FCB+12    ;Extent byte of first file
FCBRNO   EQU     FCB+32    ;Record number in first file
```

Recall these BDOS calls work by placing the function in C, and the parameter (address, character, etc) in DE. For CP/M-80 file I/O, DE must point to an FCB. It need not be the one at 5CH.

Searching For Files

You can check on whether a file exists by searching for it. CP/M-80 supplies two search types: "first", and "next". Why? Because you may be searching for, say, "*.COM". Doing a "search first" returns the first ".COM" file found. Performing a "search next" returns the next. Repeated calls to "search next" return the subsequent ones, until no more can be found.

Here is how you find out whether a file exists: (1) place its name in an FCB somewhere. This may be as simple as having the filename in the command line, so the default FCB at 5CH is filled; (2) point DE to the FCB; (3) load C with the "search first" function; (4) Call BDOS; (5) test the accumulator to see if it has 0FFH in it indicating no file found:

```
lxi      d,fcb            ;point to FCB
mvi      c,srchf          ;request search
call     bdos             ;go do it
inr      a                ;set zero if it was ff
jz       nofile
```

Search next is done similarly, simply changing the function passed in C.

How do directory programs work? Well, they start typically with an FCB containing all "?", so any file will match. (Specifically, if no filename was entered, the directory program fills the filename and filetype with all "?". Then, they do the "search first" and a bunch of "search next"s. On both search types, the value passed back in A helps find the file. If A is not 0FFH (indicating no match), then A indicates the relative position of the file in a directory sector.

CP/M-80 1.4 passed back the actual directory position, i.e. 00 for the first file, up to 3F for the 64th. This doesn't work for CP/M-80 2.x, because it supports much larger disks. A disk with 500 directory entries would have one which was number 255, i.e. 0FFH, so you would think you had not found it in a directory search. And, there is of course no way to indicate a number larger than 255 in A, which is only 8 bits.

The solution was to pass back only the position of the directory entry within its sector, rather than within the entire directory. Thus under CP/M-80 2.x, A contains either 00, 01, 02, or 03 (or 0FFH) on a search first or search next. A routine which will work for both CP/M-80 1.4 and CP/M-80 2.x needs only "ANI 3" after the search (after testing for 0FFH of course), to make sure a value between 00 and 03 is used.

The 128-byte buffer at 80H is used for search first and search next, so you can use the 00-03 value of A to find the file. Here's how:

```
         mvi      c,srchf          ;request search
next     lxi      d,fcb            ;point to FCB
         call     bdos             ;go do it
         inr      a                ;set zero if it was ff
         jz       nofile
         dcr      a                ;un-do the inr
         ani      3                ;make it 00-03
         add      a                ;double it
         add      a                ;again
         add      a                ;again
         add      a                ;again
         add      a                ;again
         adi      80h              ;now x 32 + 80h
         mov      l,a              ;set up hl to point
         mvi      h,0              ;to the entry
         inx      h                ;point to filename
```

The logic was simple: we have the relative position in the directory (00-03), but each entry is 32 bytes long. Just multiply the 00-03 by 32 (by doubling it 5 times), then add 80 to this. If you want to get the next file after processing the first, just:

```
.
.
mvi      c,srchn          ;search next
jmp      next             ;look for it in directory
```

This will loop back, and use the same code as search first.

Making Files

For a file to exist on disk, it must be in the directory. To place a file in the directory with an assembler program, you *make* the file with BDOS function 22. *CAUTION:* you must be sure not to *make* a file that already exists. If you do, you will have a useless duplicate directory entry. Standard practice is to either *erase* the file first, or *search* for it, and if you find it, ask the user if it may be erased.

Let's take the first case, where we erase any possible old file, then make a new one:

```
         lxi      d,fcb            ;point DE to FCB.
         mvi      c,erase          ;load function to C
         call     bdos             ;erase the file
;
         lxi      d,fcb            ;point DE to FCB.
         mvi      c,make           ;load function to C
         call     bdos             ;make new file
```

If the directory should happen to be full, CP/M-80 returns

0FFH in the accumulator, so the easiest test for this problem is to INR A to make it 00H, and JZ to an error routine:

```
inr     a               ;no dir. space?
jz      dirfull         ;yes, go to error.
```

This placed an *empty* entry in the directory – empty meaning only the filename and type, but 00 for length and disk block locations.

We didn't need to set up this FCB if it was the system FCB filled by CP/M-80 on a command. As you can see from the Q.COM examples above, CP/M-80 filled the extent byte and record count with 00H.

(Aside: It is actually the Console Command Processor, CCP, that is filling the FCB. There is no general facility anywhere else in CP/M-80 itself, to fill a file control block; if you want to fill one under program control, such as from a reply to a message given in your program, quite a bit of code is required. You must translate the name to upper case, scan for a possible disk [e.g. A:], place the filename in the FCB, left justified and padded with spaces to 8 characters. Similarly the filetype [if any] must be placed in the proper FCB location.) If you are working with an FCB other than one the system filled, you would have to zero the extent reserved bytes and sector number before making the file.

Opening Files

To do reads or writes to a file, the file must be *open*. The make function opens a file, so if you have just executed a make, you don't have to do an open. If you *did* do an open, it would merely obligingly find the same empty directory entry the make function had just put in the directory.

To read a file, don't perform the make function on it, since some previous program did that. Just open it. Similar to the make function, 0FFH returned in A signals an error, in this case, that the file was not found:

```
lxi     d,fcb           ;point DE to FCB.
mvi     c,open          ;load function to C
call    bdos            ;open for reading
inr     a               ;no such file?
jz      nofile          ;yes, go to error.
```

Reading and Writing Files

Reading or writing sectors to the file is very simple. Unless you tell CP/M-80 otherwise, the 128 bytes at 80H are used for the sector read or written. I keep saying "sector", because reads and writes are done at the *sector* level. You will need more complex subroutines if you want to read or write something other than a *sector*.

Read and write return a 00H in A if it was successful. Read returns an 01H if end of file is reached. Write returns an 01H if there was an error.

```
        lxi     d,fcb           ;point DE to FCB.
        mvi     c,read          ;load function to C
        call    bdos            ;read a sector
        ora     a               ;error or end of file?
        jnz     ckeof           ;yes, go check it.
        .
        .
;
;got read error - test for end of file
;
ckeof   dcr     a               ;was it eof? (=1)
        jnz     rderr
;
;eof.
;
        .
        .
```

You'll note I didn't do anything with sector numbers or extents. CP/M-80 is handling these for me. If I read past sector 128, CP/M would automatically increment the extent number, and open the next extent.

CP/M-80 works similarly for writing, making new extents for you after you have made the first. Thus it is possible for write to return an 0FFH, meaning it couldn't find room in the directory for the additional extent of the file.

Closing Files

If you have written a file, the FCB and its disk block pointers are only in memory, and not on disk, so if you were to end your program, CP/M-80 wouldn't know where the file was. Therefore, *close* the file when you have finished writing it. Again, 0FFH is an error, in this case, "unable to find matching directory entry"; it shouldn't occur unless you forgot to originally make the file:

```
lxi         d,fcb               ;point DE to FCB.
mvi         c,close             ;load function to C
call        bdos                ;close the file
inr         a                   ;close ok?
jz          badclos             ;no, go to error.
```

In CP/M-80 1.4 and 2.x, you need not close a file which you have only read. However, MP/M, with its multi-programming ability, keeps closer track of open files, and *does* require you to close a file so it knows it is again available. Thus it is not a bad idea to close even files you have read.

Erasing Files

To delete a file under program control, point DE to the FCB containing the filename (or specification, with "?" allowed), load C with the erase function (19), and call BDOS.

```
lxi     d,fcb
mvi     c,erase
call    bdos
```

So much for "bits and pieces". Next month, let's put this together into a sample program.

Listing 1
Q.HEX

```
:10010000210000392260C02316C023A5D00FE40CAC7
:10011000AB012A060025CD3201CD9C01CD9C0121E9
:100120000000000CD3201CD60012425CA22012A6C02D3
:10013000F9C97CCD4C017DCD4C01CD3D013E20C5A2
:10014000D5E55F0E02CD0500E1D1C1C9F51F1F1F26
:100150001FCD5501F1E60FC69027CE4027C33F01C2
:10016000E57ECD4C017DE6033CFE04CC3D017DE601
:1001700073CFE08CC3D01237DE60FC26101CDA600
:1001800001E17EFE20DA8D01FE7FDA8F013E2ECD69
:100190003F01237DE60FC28201CDA6013E0DCD3F7A
:1001A000013E0AC33F013E7CC33F01115E002100B6
:1001B000001A13FE20CACA0129292929FE41DAC3DF
:1001C00001D607D630856FC3B101CD3201CD6001B4
:1001D000CDDE01CA2D017CB5C2CA01C32D01E50ED9
:1001E0000BCD0500E1B7CA0602E50E01CD05000EF4
:1001F00001FE13CC0500E1FE03CA2D01FE0DCA2D40
:0802000001FE20C024C93CC925
:0000000000
```

Software Notes

INCLUDE In BASIC-80

Bob Kowitt

One of the complaints against BASIC as a computer language has been the constraint of line numbers. New versions of the different dialects of BASIC have been created over the years to remove some of these constraints. For example, the ability to renumber programs, still lacking as an integral part of many BASICs, is almost universal. In BASIC-80, the ability to renumber selectively from any point to the end by any increment desired has permitted programmers to open up a program, and make changes while keeping subroutines located in one section and the main thread of the program in another.

There is a feature of Microsoft's BASIC compiler that makes it easier to consider routines by name instead of number. Of course, the phrase used was *easier to consider*. One still cannot, as in COBOL, Pascal or PL/I:

DO procedure

where 'procedure' is a name or label attached to the procedure to be implemented.

This technique is a modification of a scheme I have been using for some time with the BASIC-80 interpreter, written up in S-100 *Microsystems* magazine this spring.

While using the interpreter, I have been storing subroutines and restricting them to line numbers between 200 and 950. There can be no duplication of line numbers in these subroutines, so when I use the BASIC80 command MERGE, the desired subroutine can be pulled in without danger of overwriting any other. By maintaining a standard jump table in the location 100 - 199, the program can GOSUB to this location and branch from there to the subroutine needed. Lines 1 -100 are reserved for introductory remarks and housekeeping.

The space between lines 950 and 1000 is used as an area for jump vectors when chaining between modules in a program; the chain must enter at various points of this module. Thus, if the module must be renumbered, lines 1 - 1000 can remain untouched and the jump destination can be renumbered without altering the re-entry point.

Since I now use BASCOM-80, this same library of subroutines is written on a disk with an editor such as PMATE or WORDMASTER. However, for this purpose all line numbers are eliminated. This form allows the ASCII coded subroutines at compile time to be called in with %INCLUDE. One of the immediately perceived advantages of this method is shorter source code.

In the example given in figure 2, the subroutine takes a name in the form LASTNAME, COMMA, FIRSTNAME and converts it to FIRSTNAME, SPACE, LASTNAME for proper printout or display. This subroutine is saved on the disk as SWAPNAME.BAS, with the 'A' toggle to save it in ASCII rather than the Microsoft token form:

```
SAVE "SWAPNAME",A
```

According to the restrictions of BASIC, both interpreted or compiled, my code must GOSUB to a line number. In this application, however, it is not necessary to write out all the code at that line number location. Programming with the editor, a dummy line number is written with a REM and a description of the function, following it on the next physical line, with %INCLUDE subroutine. On the first line of the %INCLUDEd subroutine is a REMark stating the name of the subroutine for self-documentation purposes.

Figure 1

```
240 GOSUB 510
    .
    .
510  REM    switches last name,first name
             % INCLUDE SWAPNAME
```

This results in compiled code that works as if it had been the same as figure 2, even though it is not in the original source code.

Figure 2

```
240 GOSUB 510
      .
      .
      .
510  REM   switch last name,first name
     ´  SWAPNAME - name set to E$ & has comma between name
     X=INSTR(E$,",") : IF X=0 THEN RETURN
     E2$=LEFT$(E$,X-1):E1$=RIGHT$(E$,LEN(E$)-X):E$=E1$+" "+E2$
         RETURN
```

The subroutine must have been saved in ASCII with no line numbers. It is very important that it be written in a self-contained format, with no branches via GOTO or GOSUB. Following the techniques of structured programming, be sure to allow only one exit from the subroutine, the final RETURN. With judicious use of FOR... NEXT loops and the WHILE...WEND structures, it is possible to avoid branching with the subroutine. Any branches via GOTO or GOSUB to another line number would make the subroutine dependent on the location within the program and would defeat the primary purpose of this method.

BASCOM-80 now must compile the program using the /C compile switch, which tells the compiler to ignore line number constraints. Line numbers can be in any order, or the program may have none at all except as addresses for branching. I employ so many switches in my compiling that I use a submit file to do the work. (See Fig. 3.) Actually, the program could have branches within my subroutine, but I would run the risk of duplicating a number used elsewhere in the program. I haven't tried it but I imagine that the compiler would give me a *DUPLICATE LINE NUMBER ERROR* under these conditions.

Figure 3

```
PIP A:=B:$1.BAS[V]    ;move the source file from B: to A:
BASCOM                ;invoke the compiler
=$1/C/Z/X/E/N         ;Z = Z80,X & E for on error coding,
                      ;          N inhibits symbolic generation
L80 $1, $1/N/E        ;N saves the COM, E exits to CP/M
PIP B:=A:$1.COM[VO]   ;move the object code back to B
ERA $1.*              ;erase REL file from A
```

Of course, working this way precludes testing the program with the interpreter, a restriction programmers in other languages have had to live with. BASIC programmers, at this level of sophistication, will have to trade off one advantage for another.

Obviously there is some time saved with this means of coding. However, with a single module program, the time saved is minimal. Where the technique shines is in a multi-module program. The command %INCLUDE has another advantage. Often a line of code is used in each module. These modules are called in at runtime. The line of code can be %INCLUDEd in each module when it is compiled. Should it be necessary to alter a line in one of subroutines, it saves time and trouble to recompile the appropriate modules after only one modification to a LIB file.

I use a file that I call my system control data file. In this file are the clear screen code, cursor addressing strings, end-of-line code and any other terminal dependent data that was filed at setup time. While it is true that this data can be CHAINed with COMMON, different sub-groups of modules within the large program may not require the same COMMON definition. It is easier to re-read the system data from a disk file each time a COMMON definition is changed.

In addition, many times during the creative process, the programmer may decide to change the system control file fields. After modifying the read line once, each module can be re-compiled without re-editing the source code in all modules.

Figure 4

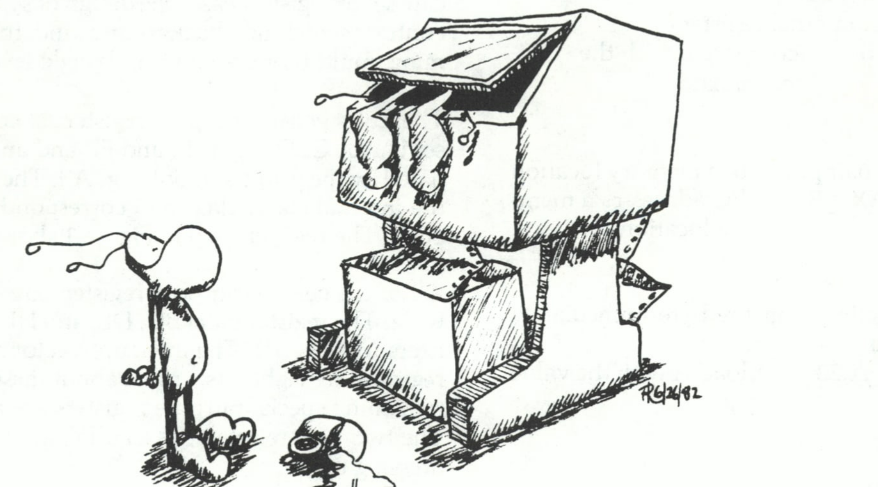
```
OPEN "I",#1,"SYSCTRL.DAT"
INPUT #1,CLSCREEN$,HOME$,EOLINE$,EOSCREEN$,LEADIN$,OFFSET,XY
```

Now after further work, should a decision be made to include, for example, screen size data, the file "SYSIN.LIB" can be modified with the additional fields, SCCOL, SCLIN.

Figure 5

```
OPEN "I",#1,"SYSCTRL.DAT"
INPUT #1,CLSCREEN$,HOME$,EOLINE$,EOSCREEN$,LEADIN$,OFFSET,
        XY,SCCOL,SCLIN
```

Then %INCLUDE "SYSIN.LIB" will read it where it belongs at compile time in each of the modules being compiled with SUPERSUB. Total writing time – five minutes instead of two hours to edit a total of ten modules.



"THAT NEW VACUUM PAPER FEEDER FOR OUR PRINTER IS MARVELOUS BUT I THINK IT'S A LITTLE TOO POWERFULL..."

Feature

Z80 Programming Tutorial, The Architectural Wonders of the Z80

Kim West DeWindt

Welcome to the wonderful world of silicon. This month, I am delving into the strange new world of the Z80's insides. Much of the Z80 looks just like the 8080. There are, however, a few key additions that can make your life, and your programming, easier. If you do not know about some of the Z80's internal quirks, some of your 8080 programs may produce bizarre results, if they run at all. The internal status of the Z80, represented by the state of the flag register, can vary from the state of the Program Status Word (PSW) of the 8080. Conditional tests and jumps that worked with an 8080 may send your Z80 into never-never land.

Let us start with the basic assumption that you have read either Ward Christensen's tutorial or Intel's data book and know something about the internal structure of the 8080. You should be familiar with the 8080's registers, and know the difference between an accumulator and a general purpose register. (You can't do arithmetic or logical operations in a general purpose register. Those functions are performed in and by the accumulator.) You should also know about the different types of address modes. Below is a summary of address modes. In the examples, the opcodes on the left are Z80 mnemonics, the underscored opcodes on the right are 8080 mnemonics.

register addressing
(also called implicit or implied):
the instruction implies an internal register
ex: CP B <u>CMP B</u> ;compare B with the accumulator

indirect addressing:
the contents of a register pair points to a memory location
ex: LD A,(BC) <u>LDAX B</u> ;BC addresses a memory location

immediate addressing:
data used by the instruction is in the byte immediately following the instruction
ex: LD A,23 <u>MVI A,23</u> j;load A with the value 23

absolute addressing:
the two bytes of data following the instruction are an absolute address pointing into memory
ex: JP 2354 <u>JMP 2354</u> ;jump to address 2354

relative addressing:
the byte of data following the instruction is an offset that is added to the program counter (PC) to provide the next program address. In the Z80, this is the Jump Relative command
ex: JR 56 <u>no 8080 opcode</u> ;56 is added to the PC

indexed addressing:
the byte of data immediately following the instruction is added to an index register, this composite address points to the data that is used by the instruction
ex: LD A,IX+5) <u>no 8080 opcode</u> ;add 5 to the index register (IX), get the data at that location and put it into A

bit addressing:
the opcode specifies which bit of the addressed byte is to be used for data.
ex: SET 3,B <u>no 8080 opcode</u> ;set bit 3 of register B equal to 1

Some of these modes may be new to 8080 users. Later on, I will go in the details of relative, indexed and bit addressing. Enough about assumptions, let's move on to some of the interesting features of the Z80.

The Prime Registers

The designers at Zilog were not content with the normal complement of registers found in the 8080. So they doubled the number of general purpose registers in the Z80. This duplicate set of registers eases the design of systems that use single-context switching, background and foreground programming (multi-user systems), and single-level interrupt systems.

The Z80's general purpose register set consists of a principal set (A, B, C, D, E, H, L, and F), and an alternate set (designated by the prime symbol, e.g. A'). The F register consists of the internal status flags and corresponds to the PSW of the 8080. The register map in Figure 1 shows their layout.

All of the general purpose registers are eight bits (one byte) wide. The register pairs BC, DE, and HL form sixteen bit registers (one word). The interrupt vector and memory refresh registers are eight bits - more about these two later. The four remaining special purpose registers are all sixteen bits wide. The two new registers, IX and IY, are used in the index addressing mode.

The prime registers (the alternate set) look great on paper. However, they are not true registers. They really consist of a small bank of internal RAM that is used to store one operating state of the Z80's main registers. By using this alternate set of registers, the programmer can save the internal context of a program and switch to another by swapping the current state of the main registers with the old values of the alternate registers. Unfortunately, the programmer can only 'see' the contents of the main register set. This ability to quickly swap one state of the Z80 with another is useful in simple interrupt schemes and timing loops, when the Z80 must leave the main stream program and process interrupts or real time functions. However, the programmer has to be careful when writing programs that handle multiple interrupts. The Z80 can only remember two states. If, in the process of handling one interrupt, another interrupt arrives, and the programmer gleefully swaps registers again, she or he is back in the realm of the main program. There go all those carefully saved register states, and nobody knows where the Z80 goes.

The Z80 has a special set of instructions which exchange the contents of the main register set with the contents of the alternate register set. EX AF exchanges the contents of the AF and A'F' registers. EXX exchanges the contents of the remaining registers (BC, DE, HL with B'C', D'E', and H',L'). In the programming section, I will go into more detail on how to use the exchange instructions, and how not to get lost in multiple interrupt systems.

The Flag Register

The F register is the collection the internal flags of the Z80. It is equivalent to the PSW of the 8080 – well, almost equivalent – the Z80 has one more flag than the 8080. There are six bits within the flag register, one more than the 8080 has. Four of them are testable (Carry, Zero, Negative-sign, and Parity/-overflow). The remaining two flag bits (Half-carry and Subtract) are for internal use and cannot be tested by the programmer. I will outline all of the flag functions here, then spend more time talking about their operation in the programming section.

The Carry flag (C) of the Z80 is similar to the Carry bit of the 8080. It contains the highest order bit of the A register after an add, subtract, shift, or rotate.

The Zero flag (Z) is set whenever the result of an accumulator operation is zero. If the result is not zero, the flag is cleared. Note that loading a zero into the A register does not set the zero flag. Only the arithmetic and (some of) the logical operations change the state of this flag.

The Negative-sign flag (N) is set if the result of an arithmetic or logical operation is negative. The seventh bit of the accumulator is the sign bit. If it is one (negative) the N flag is set. A zero in the accumulator (positive) clears the N flag.

The Parity/overflow flag (P/O) is a dual purpose flag. If a logical operation produces a result with even parity, the parity bit is set. If the result has odd parity, the bit is cleared. If an operation uses signed two's complement arithmetic, this bit is set when an overflow occurs. It is cleared if the operation did not overflow. This is an extension of the 8080's parity bit. If any of your old 8080 programs test this bit, you should make sure that the test occurs only after a logical operation (the parity check). Testing this bit after arithmetic operations may cause inexplicable effects.

The Half-carry flag (H) is set if there is a BCD (Binary Coded Decimal) carry or borrow from the lower nibble of the accumulator. The state of this flag is used internally when the Z80 is doing decimal adjust operations (the DAA instruction). This flag is an extension of the Auxiliary Carry bit of the 8080. The Aux Carry bit signals the presence of a carry out of bit three; it does not flag the borrow condition.

The Subtract flag (S) is used in conjunction with the H flag. It identifies whether the previous instruction produced a carry or a borrow. The states of the S and H flags determine the kind of correction that has to be made to a BCD integer. There is no comparable flag in the 8080.

The Interrupt Register

The I register or interrupt vector is used in interrupt mode 2. (A quick word on interrupt modes – the Z80 has three different interrupt responses. In the first mode [IM0], the Z80 expects the interrupting device to place a one byte instruction on the data bus, just like the 8080 interrupt response. In the second mode [IM1], the Z80 automatically jumps to location 38 hex [like a RST 7 for the 8080]. The third mode [IM2] uses the I register to help provide a jump address. Look for more details on interrupt responses later in this section.)

In interrupt mode 2, the Z80 looks for a byte of data from the interrupting device. It assumes that this is the low byte of an address. The I register provides the high byte of this interrupt vector address. Once the Z80 has formed this interrupt vector, it goes to that location in memory and treats the data stored there as an address. This is also known as Very Indirect Addressing. The equations below show how this works.

I register + data from intrpt device = Intrpt vector

Interrupt vector points to memory location IV hi and IV low.

The PC is pushed onto the stack and the address located at IV hi and IV low becomes the new PC.

high / low bytes
(I register)(data) = sixteen bit vector (IV) = >
= > (address low byte) = low byte of the program counter

sixteen bit vector + 1 (IV + 1) = >
= > (address high byte) = high byte of the program counter

One word of caution – the data byte that is sent by the interrupting device must be an even byte (bit A0 must be zero). The Z80 is generating a sixteen bit vector that points to a sixteen bit address which resides on an even boundary. Let me repeat that: the composite interrupt vector must be an even address, therefore its low byte must be even.

The interrupt register is loaded from the accumulator. The contents of the I register can be read by down-loading it back into the accumulator. The instructions look like this:

```
LD   I,A          ;load I with contents of A
LD   A,I          ;load A with contents of I
```

The Refresh Register

When the Z80 first came out on the market, dynamic RAM was expensive, and static RAM was very expensive. In order to simplify Z80's interface to dynamic RAM, Zilog added a refresh register and a special refresh cycle. The refresh register

(continued next page)

is a counter that is decremented after every M1 cycle. (An M1 cycle is an instruction fetch cycle. The Z80 differentiates between a normal memory read cycle, and an instruction fetch cycle. There is a special line, M1*, that goes low during an M1 cycle. There is no equivalent cycle in the 8080.) At the end of an M1 cycle, the contents of the refresh register are sent out on the lower address lines. At the same time, the RFSH* (refresh) and MREQ* (memory request) lines go low to indicate to external circuitry that a refresh cycle is in progress. This refresh address can be sent to the local dynamic RAM and used as a refresh address. Since an M1 cycle comes at least once every four cycles, the RAMs will be adequately refreshed.

This scheme works with the Z80 and Z80 A, which run at 2.5 and 4 MHz respectively. The new Z80 B runs at 6Mhz and the refresh pulse, provided by the Z80, is not long enough to refresh RAMs that are currently available. For slower systems, using slower RAMs, the refresh register and refresh cycle cut down on the number of parts needed to interface with dynamic RAMs.

The refresh register is loaded from the accumulator and its contents can be read back into the accumulator. These instructions are as follows:

```
LD  R,A       ;load R with contents of A
LD  A,R       ;load A with contents of R
```

Notice that if the R register is read randomly, or after a 'long' time, its contents will be unknown to the programmer (Eureka, a pseudo random number generator within the bowels of the Z80). If used carefully, the R register works admirably as a random number generator. If you want to try using the refresh register for this purpose, keep the following in mind:

Don't read R inside a small loop and expect it to be random. Remember, it is decremented after every instruction fetch. If there are only seven instructions in a loop, R will be smaller by seven every time around, this is not even considered quasi-random.

Actually, the above consideration holds true for large program loops. If your program requires a mathematically random number, read R at 'random' intervals. If your program branches here and there, gets periodic interrupts and waits for console I/O, the contents of R will appear quite random.

The Index Registers, IX and IY

The Z80 adds a new mode of addressing to those available in the 8080. To implement this mode, called index addressing, index registers were added. These registers are sixteen bits wide (one word) and they can be loaded, stored, incremented, decremented and exchanged with data in memory or with some of the other registers. Opcodes that specify index addressing are two bytes long, followed by a one byte offset (the offset may be zero, but that zero still takes up a memory location). This mode is not always space efficient, but now that memory is cheap, index registers make it easy to code and use translation tables, jump tables, and the like.

Until we get to the section on the Z80 instruction set, see Figure 2 for a partial list of 8080 commands that can now be extended to include the index registers. In the programming section, I will give some programming examples that use indexed addressing techniques.

Interrupt Modes

The Z80 handles interrupts in a slightly different fashion from the 8080. In the 8080, different input lines force the 8080 to jump, call, or restart. The location of the jump or restart is determined by the line that requested the interrupt. One of the lines, TRAP, cannot be ignored. The rest of the interrupt lines can be internally disabled (masked off) by software. The Z80 has only two interrupt input lines, INTR* and NMI*. NMI (Non-Maskable Interrupt) is like TRAP. It cannot be shut off by the software. A low level on the NMI* line forces the Z80 to jump to location 66 hex. INTR*, like INTR and the RST lines of the 8080, can be masked off. If INTR* is disabled (by the instruction DI), a low level on the interrupt line is ignored by the Z80. If interrupts are enabled (by the instruction EI), a low level on the interrupt line forces the Z80 into one of three interrupt modes. (Interrupts are recognized after the end of the current instruction cycle). Now, about those three interrupt modes.

Mode 0 (IM0):
This mode was designed to be compatible with the 8080 interrupt mode. After acknowledging an interrupt, the Z80 expects to see a one byte instruction (usually a Restart instruction) on the data bus. The Z80 then performs this instruction six times, just to make sure that it does it correctly, and continues on its merry way.

Mode 1 (IM1):
This mode is similar to the NMI response. The Z80 does not look for any data from the interrupting device. It jumps directly to address 38 hex. This mode is handy if the interrupt is coming from a passing clock pulse or other non-intelligent device.

Mode 2 (IM2):
This third mode is designed to be used with the family of Zilog peripherals. Actually, it works very well with any interrupt controller that can store address locations, and send selected data to the Z80. This mode takes advantage of the I register, using its contents as the upper half of an interrupt vector address. The interrupting device must provide the lower half of this address. Interrupt mode 2 gives the programmer a lot of flexibility. Your interrupt routines are not confined to certain pages or locations in memory. Interrupt handling routines can be anywhere. By manipulating the interrupt register, the same device can send the Z80 to any number of different interrupt routines. The final destination would depend on when the interrupt occurred, and what was in the I register.

In the programming section, there will be examples of interrupt vector tables, and how to use the variable contents of the I register to send your Z80 zipping throughout memory. In addition, I will spend some time explaining how to keep track of where your Z80 has zipped and how to get it back to your original location, in it's original state.

It is up to the programmer to set the interrupt modes of the Z80. Initially, the Z80 is in mode 0. There are three instructions that change the interrupt mode. These are simply:

```
IM0        ;set interrupt mode 0
IM1        ;set interrupt mode 1
IM2        ;set interrupt mode 2
```

The interrupt mode can be changed at any time. I strongly recommend that you disable all interrupts before changing the mode. If an interrupt arrived while the response mode was being changed, all hell could break loose.

Enough on the architecture on the Z80. Next time I will talk about Zilog's mnemonics. The instruction opcodes (i.e. the hex values of an instruction) are the same for the 8080 and the Z80, but the names of the opcodes have been changed. In this section I gave some examples of instructions in both Z80 and 8080 mnemonics. In the next section, I will give you a 8080/Z80 opcode translation table. Once you get used to reading Zilog mnemonics, I think that you will find them quite logical.

Because the Z80 instruction set is a superset of the 8080 instruction set, there will be some additional instructions to learn. In some cases, Zilog made use of the undefined one byte values in the 8080's instruction table. When they ran out of single byte instructions, they added a few double byte instructions (e.g. the block move instructions, and some of the instructions that use the index registers).

If anybody out there has any specific questions about the Z80 architecture, or would like to see more detailed information about its internal structure, drop a line to me care of *Lifelines/The Software Magazine*. If it appeals to the general reader, I will cover it in a future installment. If it is a question about a nitpicking detail, I will send a personal reply.

Z80 REGISTER CONFIGURATION GENERAL PURPOSE REGISTERS

main register set		alternate register set	
A (acc.)	F (flags)	A′ (acc.)	F′ (flags)
B	C	B′	C′
D	E	D′	E′
H	L	H′	L′

SPECIAL PURPOSE REGISTERS

Index register IX
Index register IY
Stack Pointer SP
Program Counter PC

Figure 1

NOTE: In all instances, IX can be replaced by IY

Z80 mnemonics		**8080 mnemonics**	
LD	IX,(nn)	LIXD	(like LHLD)
LD	(nn),IX	SIXD	(like SHLD)
LD	IX,NN	LXI	nn
LD	A,(IX + d)	LDAX	IX + d
LD	(IX + d),A	STAX	IX + d
INC	IX	INX	IX
DEC	IY	DCX	IX

Figure 2

Make your systems sales soar!

STOK PILOT™

A language for creators.

STOK PILOT can make your final product more saleable and keep it sold.

This unique langauge allows straightforward creation of a friendly interface between novice users and CP/M for any program application. Using STOK PILOT, a software designer can easily produce a hand-holding menu-driven front-end supervisor with built in tutorials to help guide the novice end-user through a complex application. The actual end-user does not have to know STOK PILOT.

STOK PILOT opens the door for companies to evolve structured approaches to meeting their own training needs.

STOK PILOT can simulate any application program for the purpose of tutoring the user. This allows training and guidance without the necessity of working on and possibly damaging live data. Then, when the user is ready, STOK PILOT can call the actual application program.

STOK PILOT is a superset of the PILOT CAI language. STOK PILOT was designed with the syntax and structure of PILOT because PILOT is an easy to use and easy to learn language.

STOK PILOT can be thought of as an interactive SUBMIT facility. One program written in STOK PILOT can control an entire session without ever entering cumbersome CP/M commands. The commands can be dynamically formed at run time by STOK PILOT.

Menus can be created that will allow the user to select various programs. STOK PILOT can chain to any program or utility written in any language, leaving the entire TPA available, and regain control when the other program ends.

STOK PILOT can check for the existence of a file before it actually calls the application that needs it. Let STOK PILOT see to it that the proper disks have been inserted by the user.

The package includes a well written and indexed 75 page manual. All the instructions are explained in full detail. Many useful programming examples are also included.

Complete 8 inch CP/M format disk and manual retails for $129.95. The manual is available alone for $14.95 and is deductible from a future order. NY residents please add sales tax.

Toll free order line: (800) 431-1953 ext 183
In NY (800) 942-1935 ext 183

C.O.D. - Mastercard - Visa

Stok Software Inc.
17 West 17th Street
New York, NY 10011
(212) 243-1444

Dealer inquiries invited
CP/M is TM of Digital Research

Opinion Letters

Some Comments On dBASE

July 10, 1982

Dear Mr. Olfe,

I have enjoyed reading your short articles about hints, bugs, etc. of dBASE II. I would like to share with you a rather serious bug that I discovered while using dBASE. It seems dBASE II can't multiply when the multiplier or multiplicand are fractions. Enclosed is a run showing the problem. [*Ed. - See below.*]

I discovered this bug while writing a command module to do simple interest calculations for a loan (you know, the one that goes interest equals principle time rate). Well, I have been playing (?) with these little fellas for a number of years (actually their big brothers) and after I got the command module to work and looked at a report that showed the interest, principle, etc. for a short five year loan, I checked some of the calculations using pencil and paper and lo and behold the numbers didn't quite match! After redoing the calculations on two other calculators I began to smell something rotten in dBASE.

As you can see from the listing, when the "?" command is used, and real numbers are used to do the calculation, the result comes out correct. But, when you try to use memory variables to do calculations, the precision is certainly less than desirable! The other thing I like is that a different result is returned depending if you multiply i*p or p*i.

When I discovered this problem, I gave Ashton-Tate a call and talked to them. They felt that it was indeed a problem. I had asked them to call me back after they discussed the problem with the "writer" to see if there was an alternate solution to performing these calculations. Three weeks and another phone call later, it seems there was little or no thought given to this problem. (Oh, well, you pay your money, you take your chances.)

Other problems I have come upon are the set alternate process not working as described in the manual or even SoftwareBanc's user manual. Ashton-Tate said it was a known problem and that the process shouldn't even be in dBASE. After a bit more coaxing, they also said there were problems with the COUNT command sometimes counting deleted records and the SET ESCAPE ON not always working with the INPUT and READ commands, and SET CARRY not operating properly while processing an empty file.

One last problem I encountered (which was actually the first one I discovered) was after I installed an invoice tracking system for a client. I had version 2.3B, he had version 2.3A. Things worked just fine on my machine, but the data base became corrupt after producing reports on his machine. After spending a couple of days looking for problems in my logic, I gave Ashton-Tate a call and they said that there was a known problem ,with the end-of-file marker being set properly, which would cause the problem I described to them. They said my client should send the distribution disk back and get the new version.

dBASE is a wonderful piece of software, especially after working with a ponderous monster like IBM's IMS data base on the mainframes. Hope you feel these notes worthy to share with your readers. Look forward to more hints of usage, and especially trying to implement some subroutine calls from dBASE.

Michael P. Kelly
Design Software
San Francisco, CA

```
b:
BO>
. clear
. store 22052.93 to princ
 22052.93
. store 0.0700000 to int
0.0700000
. store int/12.0 to monthint
0.0058333
. list memory
PRINC      (N)        22052.93
INT        (N)       0.0700000
MONTHINT   (N)       0.0058333
** TOTAL **       03 VARIABLES USED   00018 BYTES USED
. ? 0.0058333*22052.93
     128.641356500
. ? 22052.93*0.0058333
     128.641356500
. store princ*monthint * princ to bal
 128.642091500
. store monthint * princ to bal
 128.642091600
. list memory
PRINC      (N)        22052.93
INT        (N)       0.0700000
MONTHINT   (N)       0.0058333
BAL        (N)        128.642091600
** TOTAL **         04 VARIABLES USED   00024 BYTES USED
. set print off
```

Listing From Mr. Kelly's Letter

A Call For Help

July 17, 1982

Dear Sirs,

I am writing a tutorial for computers; and I need a program that intercepts a student's typed input to check it for typographical errors, tells him what his error is, and tells him to try again. It should work when he is in PIP, ED, and CP/M command modes.

Dean Dwyer
1534 Voorhees
Manhattan Beach, CA 90266

A Correction

July 14, 1982

Dear Mr. Bloch,

I recently received my July issue of *Lifelines*, and have to report an error on page 43 concerning the HP 125. At the end of that page, you state that Hewlett-Packard's CP/M-80 allows only 64 directory entries. While this was the case in the preliminary version of BIOS which you were using all of our customer shipped systems allow 128 directory entries, or extents, per disc. I realize your statements were based on your experience with the HP 125, and I can only apologize for the error on our part.

Sincerely,
Miles Kehoe
GSD On-Line Support
Hewlett-Packard

A Reply To Zoso

This letter is in response to the Opinion column written by Mr. Zoso in the July, 1982 issue of LIFELINES. I would like to take this opportunity to respond to Mr. Zoso's comments about products purchased from Priority One Electronics.

Mr. Zoso makes several comments about the manual he ordered with his Tandon 8" disk drives. The spec sheet which he received was the only information then available on the 8" Tandon disk drives. We have open purchase orders dating back to September of last year for the purchase of the actual Tandon manuals. Tandon has a new revision of this data sheet that does include more technical information, such as strapping options, which we are currently providing. Due to constant updates and revisions, Tandon has not published an offical technical manual.

We did receive a preliminary manual from Mr. Jay Tyzzer at Tandon only after assuring him that we would not reproduce it. Due to the many changes in the product, Mr. Tyzzer said the manual was not accurate and should not be redistributed. This must be the same manual he sold to Mr. Zoso for $25.00. In the preliminary manual, Tandon lists only question marks for the part numbers of the official manuals.

Mr. Tyzzer's comment that Tandon's new data sheets will have "...prominent copyright notices so nobody will be tempted to dupe it..." is an interesting one. Tandon does not make their manuals readily available and then they want to discourage those who sell their product from disseminating the information to use it. They will not sell us their manuals or allow us to copy them as indicated above. We have sold hundreds of Tandon drives; one would think that Mr. Tyzzer, only a half mile away, would see to it that we have current and complete information to distribute to our customers. The people at Tandon could learn a valuable lesson from William DeLorie of Qume. When Mr. DeLorie learned that the Qume OEM manual we were selling was not current, he sent us a copy of the current manual as well as a copy of the service manual and encouraged us to reproduce both. Now every customer who purchases a Qume drive from Priority One may also purchase the manuals without having to call the factory for assistance.

Mr. Zoso's second complaint deals with his friend's purchase of two Mitsubishi disk drives and their subsequent replacement. Contrary to Mr. Zoso's statements, we did offer to pay the freight on all defective Mitsubishi drives. We offered to issue UPS Call Tags to all customers who needed to return their drives. The delivery of these drives from Japan is beyond our control. We quote delivery in good faith based on the information the manufacturer provides for us. In the case of the Mitsubishi drives, they did arrive three weeks after their promised due date in April. We can only apologize for any inconvenience this delay may have caused Mr. Zoso's friend.

Mr. Zoso's final comments concerning the pricing on Qume disk drives were in error. During the months of February and March we were able to purchase a limited number of Qume disk drives at a reduced cost. These special purchase drives had a white face plate rather than the standard black face plate. Our lower cost was passed on in the form of special sale pricing to our customers. At the reduced prices, we quickly sold out of these drives with the white face plates. When these drives were gone, we resumed selling our normal stock of drives with black face plates at a price that reflected our normal cost. Contrary to Mr. Zoso's opinion, we did not raise the price $100.00, but rather we lowered our price $100.00 while we were able to.

Thank you in advance for printing our reply to Mr. Zoso's opinions.

Sincerely,
John C. Gunn
Customer Service Manager
Priority One Electronics

MicroPro Replies

Sirs:

Your April, 1982 issue contains some information which needs clarification.

Concerning Mr. Van Natta'a evaluation of SpellStar Version 1.0 (pp.25 thru 27):

The first problem mentioned was that there existed problems with the interface between SpellStar and Lifeboat's version (2.25b) of CP/M for the TRS 80 Model II. The situation relates to SpellStar 1.0 handling of memory-mapped video boards.

Second, the article says that SpellStar is slow. The version tested is not as fast as some others. Also noted is the fact that work files are left on the disk when not enough space is available for SpellStar to sort its files.

(continued next page)

All three of the above mentioned problems have been resolved in SpellStar version 1.2 which is available for shipment. It is as least twice as fast on small files as version 1.0, and also faster on larger files. It handles memory-mapped video boards as they should be, hence, you don't need to downgrade your WordStar by re-installing it as a serial terminal. Version 1.2 deletes its work files upon both a normal exit, and premature ones. It also no longer stops processing after 9500 words.

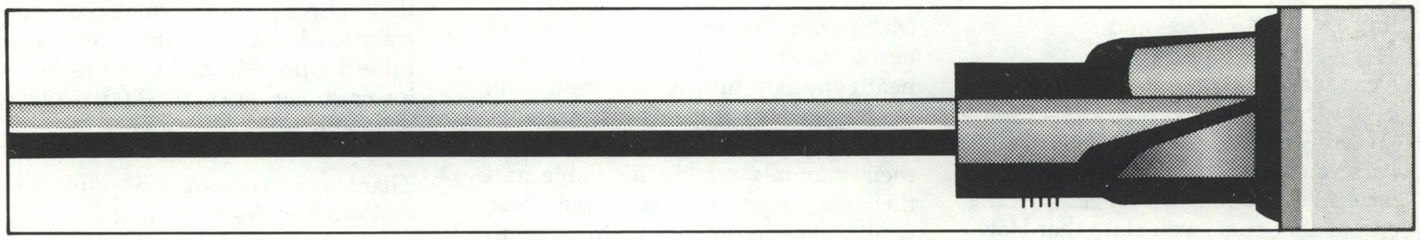
Next, under the heading of Software Notes, Page 46: A Patch for WordStar on the Superbrain. You should note that users of DOS 3.0 on the Superbrain only require the suggested modification if they bought their WordStar from Lifeboat Associates (Or their disk has a Lifeboat sticker on it). If they bought their disk from anyone else, the patches are already made. Also, the differences between the three patches published are three bytes (1 at address 02ED and two starting at 02F1.). This problem was caused by Lifeboat downloading a regular 8" WordStar to Superbrain format and selling that. Special changes were put into our version of WordStar on the Superbrain to allow it to use memory-mapped video.

For users of MicroPro's version of WordStar for the Superbrain, here are the areas which need to be patched for those who have later than 3.0 DOS.

This is not to imply that the patches documented will not work for all versions of WordStar on the Superbrain, but the above patches seem less frightening for those who have MicroPro's version.

Sincerely,
Joe Masters
Product Release and Evaluation Services
Technical Services Department
MicroPro International Corporation

Location	For 3.1 DOS Change to :	For 3.2 DOS Change to :
02ED	00	00
021F1	18	20
02F2	E7	E7



BASIC/Z . . .

the ultimate CP/M* compiler!

- Generates native code (8080/Z-80) for *fast* execution - 16 bit versions soon
- Sort verb is unmatched by stand-alones. 2000 elements in two seconds!
- Alpha-numeric labels, variable and function names of any length
- Chain program segments which share variables declared common
- Five data types - binary/BCD/string
- BCD floating point math - *never* a "round-off" error - precision is program definable from 6-18 digits
- Full function program editor tests syntax as you type
- Recursive, multi-line, multi-argument user defined functions
- **No royalties - No run-time charges**
- Dimension arrays dynamically (to an expression) and selectively erase
- Screen oriented editing of console input at run-time (cursor left/right/start/end, delete left/right/line, insert/change mode, and input masking available)
- Push/pop subroutine stack
- Trace and single-step debugging
- Multi-tiered error trapping even handles BDOS errors
- Cursor addressing, reverse and blinking video, erase and more are supported from source code level, with virtual hardware independence
- An extended library of over 200 "key-word" functions

For free brochure and mini-manual:

System/z, inc.
P.O. Box 11
Richton Park, IL 60471
(312) 481-8085

* a trademark of Digital Research

System/z, inc.

Product Status Reports

New Products

The products described below are available from their authors, computer stores, software publishers and distributors. Information has been derived from material supplied by the authors, and *Lifelines/The Software Magazine* can assume no responsibility for its veracity. Software of interest to our readers will be tested and reviewed in depth at a later date.

Advanced Legal Software
Advanced Legal Software

This law office information and management system includes client billing and trust accounting. In conformity with American Bar Association standards and the Code of Professional Responsibility, the package was developed by attorneys and uses legalistic jargon rather than "computerese".

Hourly, flat fee and contingency types of billing are supported, with fourteen hourly rates for each attorney. Client accounts may be open, inactive, closed or deleted. Professional statements can be produced in eight different user-defined formats.

The system runs under CP/M-80, CP/M-86 or MP/M-80.

APPLI-CARD
Personal Computer Products, Inc.

This CP/M-80 product furnishes Apple II users with the ability to execute WordStar and fully utilize its features.

It provides 64K of on-card memory and comes standard with a 4MHZ Z80A; a 6MHZ Z80B can be ordered separately. 2K or 8K of EPROM are permitted on the card and a real time clock is supported. An expansion interface port is available for added memory.

The card comes with CP/M-80 or SB-80 and allows all 96 ASCII characters to be input or output without hardware modifications, along with upper and lower case. 40 to 255-column horizontal scrolling and 70 column by 24 line screens are supported.

With the full implementation of WordStar, the APPLI-CARD is priced at $595. It will be available this fall.

Business Software Series
Microcomputer Consultants

The elements of this series include Inventory Control for Manufacturers, Job Cost Control, General Ledger, Accounts Receivable, Accounts Payable, and Retail/Wholesale Inventory. Designed for multi-user situations, record-locking allows users to simultaneously view but not update the same record. A record in the process of being changed may not be interfered with by a second user.

Digital Research's Access Manager is utilized for file handling. An index is maintained for all large data files. Portions of this index are buffered in memory.

The series requires MP/M-I or -II. Suggested retail prices for individual modules range from $600 to $995.

C/80 Compiler
The Software Toolworks

This C compiler supports major C language features, excepting floats, longs, and arguments to macros. It includes data initialization and all C storage classes. C/80 compiles all C arithmetic and logical operator and control statements, generating 8080 assembly language code for Macro-80, or for the included absolute assembler. The runtime profile facility prints execution time and frequency for each subroutine.

The suggested retail price is $49.95; C/80 requires CP/M-80 or HDOS.

COGEN
BYTEK

This ANSI COBOL '74 RM/COBOL program generator is intended for standard business applications: file maintenance, inquiries and reports. Modules create files, screens and programs for use in business application systems. The modules are designed for flexibility in program generation. Overlay and split screen features are supported.

Reports with optional headers, multiple detail lines, multiple control breaks, conditional printing, and data selection from reference files are part of COGEN. The user can document applications with hard copies of screen and report layouts; these layouts can be updated.

COGEN requires less than 10K of memory, with versions to run under MP/M-86, CP/M-80, CP/M-86, and OASIS. Contact the author for pricing.

footPRINT!
Sterrett Consulting, Inc.

This text formatting program takes standard ASCII files with preformatting commands as input, producing output to printer, console, file, and other devices. Modularly written in Pascal, it supports chained logical input devices, continuous status display, interrupt/resume processing, selective page processing and single sheets or continuous forms.

Page length and width, margins on four sides, and heading/footing notes (multi-line) can be controlled by the user. Right and left block indentations are allowed, along with automatic line break and line break override, conditional hyphens and word breaks, phantom spaces, variable line spacing, conditional page breaks and overrides, automatic widow/orphan line control, automatic footnote placement and continuation.

Centering, microjustification (left or right), broken/solid underscores, boldface, sub- and superscript, arabic or roman page numbering and footnote numbering are designed to make a more impressive presentation.

The author states that a single-user license for this product will be priced at under $200.

ITOZ/ZTOI
RD Software

These independent source program processors or translators run on Z80-based systems. ITOZ accepts a CP/M-80 file written in Intel 8080 mne-

(continued next page)

monics and creates Zilog mnemonics output. ZTOI translates from Zilog back to Intel. The translated files are fully formatted, containing all original comments. Command line options include conversion of asterisks to semicolons as comment delimiters and the addition of colons as label terminators. Errors are flagged by console messages as well as being written into the output file as comment lines.

Utility programs for each translator allow the user to add custom pseudo-ops or macro names to the translators.

The suggested retail price is $40.

LEADtrack
LEADtrack Corporation

LEADtrack is a multipurpose marketing tool for tracking trade advertising programs and inquiries from space ads, publicity releases and trade shows. It can also be used for client lists and records, sales information and medical files.

The three essential modules are: prospect/customer data management with sales territories calculated by zip code, literature fulfillment support (labels, personalized letters, sales lead notification), six analytic reports for tracking and tabulating sales leads (by magazine, product, ad campaign, territory). This third module also can produce analyses of prospects' purchase status within a territory.

This CP/M-80 compatible product is written in BASIC and can be purchased as a service bureau option, or outright.

MATHLIB
BRIDGE Computer Company

The FORTRAN programs included in this library perform mathematical, engineering and scientific computations, like vector and matrix operations, linear/non-linear equations and regressions, optimized least squares and eigenvalue/eigenvector problems. Several independent documented modules make up the product, and source code is provided in machine-readable and hard copy form.

The suggested retail price is $365.

MCDISPLAY
MasterComputing, Inc.

This software development utility includes a display development aid with user interface. The application programmer defines screen displays using a large system approach. The displays are saved as disk files and are used by the application program through a series of directives contained in the user interface. The interface also processes data entries, handles messages and prompts, provides error checking, and converts user entries into string, integer, single or double precision variables. Message and prompt overlays are supported.

The system currently operates under Microsoft BASIC-80 and BASCOM, but it is designed to be language independent.

A cursor-addressable terminal is required, along with CP/M-80. The suggested retail price is $175.

Plotpak
BRIDGE Computer Company

Plotpak is a two-module graphics package, consisting of a user interface and a hardware interface. A screen logical device and a plotter logical device share commands and execute graphs simultaneously if both enabled. The hardware interface communicates with the plotter logical device.

This product is designed for general scientific and engineering graphs, signal processing presentation results, computer-aided designs, and linear graphs for reports. It is written in FORTRAN-80, RATFOR and MACRO-80, with code provided in printed and machine-readable form.

This plotting library includes drivers for MicroAngelo 512, ADM + Reprographics, TEK 4010-compatible terminals, HP 7225B table plotter and HP 7470B multipen plotter, Houston Instruments DMP-4 multipen plotter and TRS-80 ball pen plotter. The user may write other drivers if necessary. CP/M-80 and FORTRAN-80 are required. The suggested retail price is $365.

Record Management System
Washington Computer Services

This relational database management system is written in assembler language and is designed to perform such record keeping tasks as inventory control, mailing lists, accounting functions. Records can be accessed by alpha or numeric keys; the maximum file size is 65535 records. Each record may consist of up to fifty fields and 1021 bytes. Record layout is dictionary-driven, and files can be merged or moved when a record format change is desired.

Included with the system is a form fill out, record look up, update and data entry system. A configure program helps the user customize RMS for his or her CRT and keyboard. The ASCII file format means that RMS files can be accessed via other utilities or from user-written BASIC or Pascal programs.

Other features are a sort and a report creation system. The report creation system allows arbitrarily complex record selection and user control of output format, which can be routed to the screen, printer or a disk file.

The suggested retail price of RMS is $395; it runs under CP/M-80 or IBM PC DOS (requiring 48K).

STRPAK
BRIDGE Computer Company

This collection of 135 subroutines and functions is written to enhance the FORTRAN-80 compiler. Directional I/O, string/character manipulation routines, and file control procedures are included. Source code in RATFOR and MACRO-80 is supplied with STRPAK.

FORTRAN-80 and CP/M-80 are required. The suggested retail price is $275.

XASMZ8
XASM75
Lifeboat Associates

These two new cross-assemblers generate machine code for the Zilog Z8 microprocessor and the NEC 7500 microprocessors respectively, accepting their standard instruction mnemonics and syntax.

Input is taken from a CP/M-80 text file, and output is an object code file, an assembly listing and an alphabetized listing of all symbols defined in the assembly. The object file may be in Intel HEX format (or Motorola MIKBUG format for XASM75) or omitted. Listings may be sent to the list device, console, or disk file.

The assemblers feature storage definition instructions, conditional assembly, control of listing format, multiple source files in an assembly. New mnemonics may be defined as synonyms for opcodes and pseudo-ops, minimizing the need for re-editing existing source programs.

XBASIC
Xitan Systems

This Z80 interpreter provides MP/M-80 compatibility, includes matrix arithmetic function, areas and code for the incorporation of user-supplied devices, record-locking, and test and write facilities. Multiple printer use and password protection of files are also supported.

Other features are a user key-definable line editor, timeout on input statements, full error trapping, a four level trace facility, user-selectable precision of any numeric variable (integer to 14 digits), extended variable names (up to 13 characters), binary coded decimal arithmetic, and a 24K size in the graphics version. A cross-reference utility is included with the interpreter.

The graphics version supports Hi-Tech's Simple Image Display Card and has eleven commands.

This product requires CP/M-80 2.2, MP/M-II, or CDOS.

New Versions

Microspell
Version 4.4

This update runs under MSDOS and corrects a few bugs in the previous version: the /D switch was omitted from the menu; the INVERT utility under certain conditions searches on only the currently logged drive for the LEX file to be inverted. The new version also includes a revised context printing routine which reduces the occurrence ot word fragments at either end of a line.

A new feature causes SPELL to search for double words and allows you to eliminate the second occurrence. This feature can be disabled with a switch.

This version requires 48K of memory in addition to that required by MSDOS. 160K storage per drive recommended.

ZDM/ZDMZ Debuggers
Versions 1.4/2.3

Four new commands are included with this update: a toggle to enable output to be directed to a list device, a string search command, a block verify command, and a toggle which when used with the TRACE command will break and display branch instructions only.

Bugs

Bugs

ZDT
Version 1.41

Murray Lesser of Yorktown Heights, N.Y. has informed us that when ZDT is used and a breakpoint is reached, ZDT uses the current CPU stack for its own internal operations, instead of saving the current stack pointer and using its own internal stack. This causes some programs to fail when running under ZDT.

Books

Books

DISCOVER FORTH: Learning And Programming The FORTH Language
By Thom Hogan
OSBORNE/McGraw-Hill,
Berkeley, California

Reviewed by Raymond J. Sonoff

In the introduction to this book the author states, "This book is an attempt to put what I've learned about FORTH into a coherent, organized introduction that others new to the language will appreciate...." This reviewer feels that Mr. Hogan's attempt was successful.

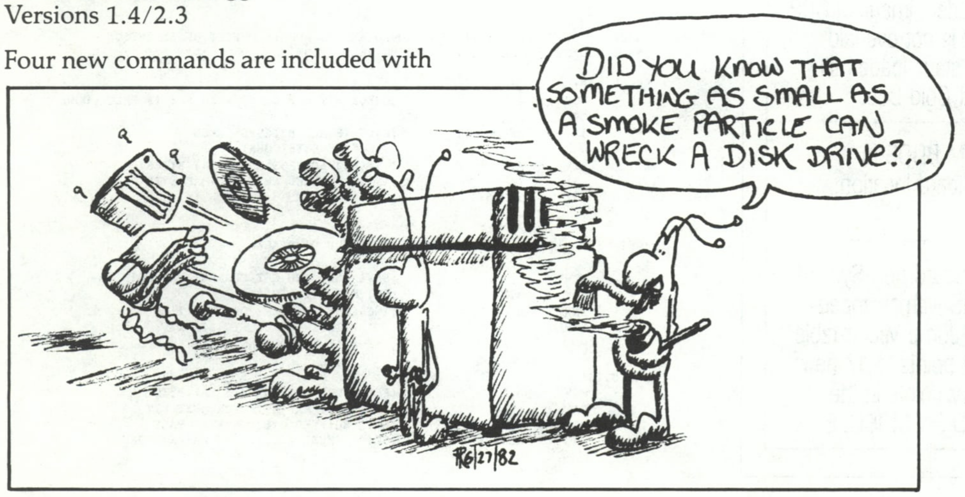
This book is written in a casual yet highly informative style, and I found it easy to visualize the author as an individual who practices a thoughtful "here's what you need to know (and why)" philosophy. DISCOVER FORTH gently leads its reader through such diverse topics as the origin of FORTH, key attributes that make FORTH such a powerful language, essential concepts and definitions from among what initially appears to be a maze of FORTH terminology, descriptive programming examples, and even suggestions on getting started doing your own programming. The book concludes with six appendices that consolidate useful reference material and other detailed information on FORTH.

The twelve chapter titles reveal the structured presentation: A Description of FORTH; The FORTH Dictionary; The Stack; FORTH Arithmetic; Using FORTH; Interpreting and Compiling; Memory Manipulations; Mathematical Possibilities; Control Structures; Input and Output; Programming FORTH; And So: FORTH.

The six appendices were titled as follows: Coding Sheet for FORTH Programming; FORTH-79 Standard Glossary of Words; ASCII Character Codes; Suggested Alternatives to the FORTH Syntax; Error Messages; Some FORTH Extensions.

Each of the twelve chapters in the 142-page softcover book is relatively short. Nonetheless, the author quickly states his intentions, supports his statements with illustrations and examples of FORTH coding, and concludes each chapter with a summary of what should have been learned. Thus, any interested reader who combines actual FORTH software programming experience with regular referral to DISCOVER FORTH will undoubtedly gain more than just a fundamental appreciation for this language.

Perhaps too, the reader will ultimately come to realize the effort that Mr. Hogan had to expend to make everything seem so simple after all. Could it be that the "building block approach" which the FORTH language is based upon has also served the author so well? Read and enjoy!


DID YOU KNOW THAT SOMETHING AS SMALL AS A SMOKE PARTICLE CAN WRECK A DISK DRIVE?...
RG/27/82

(A Memory Resident Floppy Disk Program con't from p. 12)
philosophy to grow in a manner that permits maximum public domain usage and benefit to be derived from their use.

If the reader has a desire to experiment with the BIOS extension scheme discussed here, I will be happy to share the complete source code shown here and a text listing file of this article with anyone that wants it. To obtain a copy just send an eight-inch single density formatted diskette to me in a diskette mailer with sufficient enclosed return postage and address label to this address: Michael J. Karas, 2468 Hansen Court, Simi Valley, California 93065. If you can reach me via phone at (805) 527-7922 some evening between 7:30 PM and 10:30 PM California time, I can also arrange to send you the article text and programs via modem at 300 baud with the standard Ward Christensen MODEM program. Plan for about one hour of modem time at 300 baud.

FIGURE 1
Memory Layout With an ADD-ON MODULE

(RAM Map based upon 56K sized CP/M-80 Ver. 2.2)

Memory Address	Functional Usage Without ADD-ON MODULE	Functional Usage With ADD-ON MODULE
0000H	Warm Boot Jump to BIOS Vector Table to reload CCP/BDOS (JMP 0DA03H)	CCP Restart Jump through 2nd modified BIOS Vector to ADD-ON (JMP 0DA03H)
0005H	System Call to BDOS (JMP 0CC06H)	System Call to BDOS through ADD-ON MOD (JMP 0BA06)
0100H	User Transient Program Area (TPA)	User Transient Program Area (TPA)
BA06H		ADD-ON MODULE image moved here by loader PRLMOVE.
C400H to CC05H	CP/M CCP in standard location that may be overlaid by the transient if needed	Resident image of CCP that is not overlaid but stays loaded till next Cold Boot.
CC06H to D9FFH	CP/M BDOS in standard location.	CP/M BDOS in standard location.
DA00H to DFFFH	Standard 56K System BIOS for dual Single Density Floppy Drives	Standard 56K System BIOS with swapped out Jump Vector table that points to 17 new entry points in the ADD-ON MODULE.

Listing 1 – ADD-ON MODULE Relocator Program

```
;*****************************************************************
;       PRL FILE FORMAT RELOCATER FOR BIT MAPPED FILES
;*****************************************************************
;
;       THIS SMALL PROGRAM BLOCK MOVES A BIT MAP RELOCATABLE
;       FILE FROM ADDRESS 0200H ABOVE THIS MODULE UP TO A SPOT
;       BELOW THE CCP AND THEN JUMPS TO THE BASE OF THE MOVED
;       CODE. THE DIGITAL RESEARCH RELOCATING ASSEMBLER AND THE
;       COMPANION LINKER PERMIT THE GENERATION OF THE PRL FILE
;       FORMAT. LINK WILL PUT THE CODE SIZE WORD IN TO ADDRESS
;       101 AND 102. THE CODE SPOT IS INTENDED TO BE 0200H WITH
;       THE BIT MAP IMMEDIATELY ABOVE ABOVE THE CODE. A ONE IN THE
;       BIT MAP INDICATES THE LOCATION OF A MOVED BYTE THAT
;       REQUIRES A RELOCATION ADDRESS  OFFSET.
;
;*****************************************************************
;
;
;START POINT FOR THE BEGINNING OF THE MOVER MODULE
;
        ORG     0100H
;
;
;GENERAL CP/M BDOS INTERFACE EQUATES.
;
BDOS    EQU     0005H               ;FILE MANAGER ENTRANCE LOCATION
CODE$START EQU  0200H               ;LINK 80 CODE START POINT FOR ORG 0 FILE
;
;
;START POINT OF MOVER CODE
;
        DB      01H                 ;INSERT HEX FILE CODE FOR LXI B,XXXX
        DS      2                   ;INSERT CODE SIZE FROM LINK .PRL FILE
                                    ;WITH DDT HERE
        LXI     H,0                 ;GET CCP STACK FOR LATER PASSING
        DAD     SP                  ;TO RELOCATED PROGRAM
        LXI     SP,CODE$START       ;LET STACK WORK DOWN FROM 0200H
        PUSH    H                   ;SAVE CCP POINTER ON OUR STACK
;
        PUSH    B                   ;SAVE A COPY OF CODE SIZE ON STACK
;
;
;GET BDOS PAGE ADDRESS BOUNDARY
;
        LXI     H,BDOS+2            ;PAGE ADDRESS OF BDOS BASE
        MOV     A,M                 ;INTO (A)
        SUI     8                   ;DECREASE SIZE FOR CCP SIZE
                                    ;OF EIGHT PAGES
        DCR     A                   ;ONE MORE TIME TO ACCOUNT FOR
                                    ;..PARTIAL PAGE CODE SIZE
        SUB     B                   ;SUBTRACT CODE SIZE IN TRUNC INTETGER SIZE
;
        MOV     D,A                 ;(DE) = DESTINATION ADDRESS BASE
        MVI     E,0
        PUSH    D                   ;SAVE LOAD ADDRESS FOR LATER JUMP TO CODE
;
        LXI     H,CODE$START        ;START MOVE POINTER TO (HL)
;
;
;LOOP TO MOVE CODE UP IN RAM UNDER CCP
;
MOVLOOP:
        MOV     A,B                 ;CHECK BYTE COUNT TO SEE IF ALL MOVED YET
        ORA     C
        JZ      MOVDONE             ;EXIT LOOP IF DONE
;
        DCX     B                   ;DECREMENT BYTES TO MOVE COUNT
        MOV     A,M                 ;GET A BYTE TO MOVE
        STAX    D                   ;SAVE AT DESTINATION ADDRESS
        INX     D                   ;BUMP SOURCE DESTINATION POINTERS
        INX     H
        JMP     MOVLOOP             ;GO MOVE MORE BYTES
;
;
;CODE MOVED SO SET UP TO SCAN BIT MAP
;
MOVDONE:
        POP     D                   ;GET BACK A COPY OF THE DESTINATION ADDR
        POP     B                   ;RESET (BC) TO BYTE COUNT FOR BIT MAP SCAN
        PUSH    H                   ;SAVE ADDRESS OF BIT MAP ON TOP OF STACK
;
        MOV     H,D                 ;SET (H) TO RELOCATE PAGE OFFSET
        DCR     H
;
;
;LOOP TO SCAN CODE BLOCK JUST MOVED AND TO ADD IN OFFSET OF EXECUTION
;PAGE ADDRESS ON ALL BYTES NEEDING RELOCATION.
;
RELOCLOOP:
        MOV     A,B                 ;CHECK BIT MAP COUNTER TO SEE IF RELOC DONE
        ORA     C
        JZ      RELOCDONE           ;EXIT IF ALL BYTES CHECKED
        DCX     B                   ;DECREASE BYTE COUNT
        MOV     A,E                 ;IS (DE) ADDRESS MOD EIGHT BYTES?
        ANI     7                   ;IF SO WE NEED NEXT BIT MAP BYTE
        JNZ     SAMEBYTE            ;STILL ON SAME BIT MAP BYTE
;
;
;GET NEXT BIT MAP BYTE VIA POINTER ON TOP OF STACK
;
        XTHL                        ;SAVE (HL) AND GET CURRENT MAP POINTER
        MOV     A,M                 ;GET MAP BYTE TO (A)
        INX     H                   ;INCREASE POINTER FOR NEXT TIME
        XTHL                        ;PUT POINTER BACK ONTO STACK
        MOV     L,A                 ;MAP BYTE TO HOLDING REGISTER
;
SAMEBYTE:
        MOV     A,L                 ;FETCH OUR CURRENT BIT MAP BYTE
        RAL                         ;SHIFT BIT MAP BYTE FOR NEXT BIT
        MOV     L,A                 ;SAVE SHIFTED ONE FOR NEXT PASS
        JNC     NOOFFSET            ;SKIP OFFSET ADD IF BIT WAS NOT ONE
;
;
```

```
;GET CODE BYTE AND ADD IN OFFSET IF MAP BIT WAS 1
;
        LDAX    D               ;FETCH THE DESTINATION BYTE
        ADD     H               ;ADD IN OFFSET
        STAX    D               ;STORE BACK AWAY
;
NOOFFSET:
        INX     D               ;INCREASE THE MOVED CODE BYTE POINTER
;LOOP TO MOVE CODE UP IN RAM UNDER CCP
;
MOVLOOP:
        MOV     A,B             ;CHECK BYTE COUNT TO SEE IF ALL MOVED YET
        ORA     C
        JZ      MOVDONE         ;EXIT LOOP IF DONE
;
        DCX     B               ;DECREMENT BYTES TO MOVE COUNT
        MOV     A,M             ;GET A BYTE TO MOVE
        STAX    D               ;SAVE AT DESTINATION ADDRESS
        INX     D               ;BUMP SOURCE DESTINATION POINTERS
        INX     H
        JMP     MOVLOOP         ;GO MOVE MORE BYTES
;
;
;CODE MOVED SO SET UP TO SCAN BIT MAP
;
MOVDONE:
        POP     D               ;GET BACK A COPY OF THE DESTINATION ADDR
        POP     B               ;RESET (BC) TO BYTE COUNT FOR BIT MAP SCAN
        PUSH    H               ;SAVE ADDRESS OF BIT MAP ON TOP OF STACK
;
        MOV     H,D             ;SET (H) TO RELOCATE PAGE OFFSET
        DCR     H
;
;
;LOOP TO SCAN CODE BLOCK JUST MOVED AND TO ADD IN OFFSET OF EXECUTION
;PAGE ADDRESS ON ALL BYTES NEEDING RELOCATION.
;
RELOCLOOP:
        MOV     A,B             ;CHECK BIT MAP COUNTER TO SEE IF RELOC DONE
        ORA     C
        JZ      RELOCDONE       ;EXIT IF ALL BYTES CHECKED
        DCX     B               ;DECREASE BYTE COUNT
        MOV     A,E             ;IS (DE) ADDRESS MOD EIGHT BYTES?
        ANI     7               ;IF SO WE NEED NEXT BIT MAP BYTE
        JNZ     SAMEBYTE        ;STILL ON SAME BIT MAP BYTE
;
;
;GET NEXT BIT MAP BYTE VIA POINTER ON TOP OF STACK
;
        XTHL                    ;SAVE (HL) AND GET CURRENT MAP POINTER
        MOV     A,M             ;GET MAP BYTE TO (A)
        INX     H               ;INCREASE POINTER FOR NEXT TIME
        XTHL                    ;PUT POINTER BACK ONTO STACK
        MOV     L,A             ;MAP BYTE TO HOLDING REGISTER
;
SAMEBYTE:
        MOV     A,L             ;FETCH OUR CURRENT BIT MAP BYTE
        RAL                     ;SHIFT BIT MAP BYTE FOR NEXT BIT
        MOV     L,A             ;SAVE SHIFTED ONE FOR NEXT PASS
        JNC     NOOFFSET        ;SKIP OFFSET ADD IF BIT WAS NOT ONE
;
;
;GET CODE BYTE AND ADD IN OFFSET IF MAP BIT WAS 1
;
        LDAX    D               ;FETCH THE DESTINATION BYTE
        ADD     H               ;ADD IN OFFSET
        STAX    D               ;STORE BACK AWAY
;
NOOFFSET:
        INX     D               ;INCREASE THE MOVED CODE BYTE POINTER
        JMP     RELOCLOOP       ;GO TO PROCESS MORE BYTES
;
;
;HERE WHEN THE RELOCATION IS DONE READY TO JUMP TO THE MOVED CODE
;REMEMBER THAT (H) HAS PAGE ADDRESS OF MOVED CODE
;
RELOCDONE:
        POP     D               ;GET BIT MAP POINTER OFF STACK
        MOV     D,H
        INR     D               ;SET UP EXECUTION ADDRESS
        MVI     E,0             ;MAKE (DE) AN EVEN PAGE BOUNDARY ADDRESS
        POP     H               ;GET THE SAVED CCP STACK POINTER
        SPHL                    ;RESET FOR GOING TO MOVED PROGRAM
        XCHG                    ;GET TRANSFER ADDRESS FROM (DE)
        PCHL                    ;OFF TO THE MOVED CODE AREA
;
        END
;
;
;+++...END OF FILE
```

Listing 2 – RAM DISK ADD-ON MODULE Demonstration Program

```
;*************************************************************************
;               MICRO RESOURCES CP/M RAM DISK ADDON MODULE
;*************************************************************************
;
;
;       THIS PROGRAM IS A SMALL TRANSIENT PROGRAM BASED BIOS SUBSTITUTE
;       THAT ALLOWS FILE TRANSFER UTILITY BETWEEN THE NORMAL CP/M SYSTEM
;       DISKS AND AN MEMORY RESIDENT "RAM DISK". THIS TRANSIENT PROGRAM IS
;       SETUP FOR ANY VERSION 2.2 CP/M SYSTEM.
;
;       THIS PROGRAM PRESENTS A SMALL DISK DEMONSTRATION VERSION THAT USES
;       20 K BYTES OF THE HOST SYSTEM TPA TO DEMONSTRATE THE "RAM DISK"
;       ADDON MODULE TECHNIQUE. THE DEMO VERSION USES 1K BYTE ALLOCATION
;       GROUPS (BLOCKS) AND ALLOWS FOR A TOTAL OF 32 DIRECTORY ENTRIES AS
;       DEFINED BY THE DISKDEF MACRO CALLS. PLEASE NOTE THAT WHILE THIS MAY
;       NOT APPEAR TO BE A VERY PRACTICAL IMPLEMENTATION OF A RAM DISK
;       IT DOES DEMONSTRATE THE DRIVER TECHNIQUES. A GREATLY EXPANDED VERSION
;       WOULD USE ADDITIONAL BANK SWITCHED RAM BOARDS TO STORE THE DISK DATA
;       SUCH THAT THE DISK STORAGE AREA COULD EASILY BE EXPANDED TO AS MUCH
;       AS A MEGABYTE OR MORE.
;
;       THE SUPPORTED RAM DISK FORMAT IS AS FOLLOWS:
;
;               10 TRACKS MAPPED INTO RAM
;               8 PHYSICAL SECTORS PER TRACK
;               256 BYTES/PER SECTOR
;               SECTORS ACCESSED FROM RAM THROUGH THE STANDARD DEBLOCKING
;                       TECHNIQUES. RAM RESIDENT "HOST SECTORS" MAPPED TO
;                       256 BYTES IN SIZE SO EXPANSION OF THIS PROGRAM TO
;                       BANK SWITCHED MEMORY CAN BE MADE EASILY.
;
;       PROGRAM ACCESS TO THE RAM DRIVE IS DONE IN 256 BYTE SECTORS
;       THAT ARE DEBLOCKED WITHIN A BUFFER CONTAINED INSIDE OF THIS SOFTWARE
;       PACKAGE. THE INITIAL LOADING OF THIS SOFTWARE SWAPS OUT THE NORMAL
;       CP/M DISK I/O ENTRY POINTS TO THE BIOS WITH A NEW SET OF ENTRY POINTS
;       TO THIS MODULE. THIS MODULE THEN CHECKS ALL SELECT DISK ACCESSES FOR
;       LOGICAL UNIT P: AND WILL AND  WILL STEER I/O REQUESTS FOR THIS DRIVE
;       THROUGH DRIVERS CONTAINED WITHIN THIS PROGRAM. NOTE THAT THIS PROGRAM
;       STILL DEPENDS UPON THE HOST CP/M SYSTEM BDOS AS THE FILE INTERFACE
;       MEDIUM.
;
;       OPERATION OF THE PROGRAM IS DONE TO MOVE THE MODULE UP TO A WORKSPACE
;       BELOW THE MEMORY RESIDENT CCP. THE WARM BOOT VECTOR AT THE SYSTEM
;       WARM BOOT ENTRY POINT IS SWAPPED TO A NEW ENTRY POINT WITHIN THE
;       RELOCATED I/O MODULE. THE NEW WARMBOOT FUNCTION SIMPLY RE-ENTERS
;       THE ALREADY PRESENT CCP FOR FURTHER OPERATOR COMMAND PROCESSING.
;       THE BDOS ENTRY VECTOR IS ALSO MODIFIED TO PERMIT THE DYNAMIC
;       MODIFICATION OF THE USER PROGRAM AREA SIZE SUCH THAT THE CCP AND
;       THE RELOCATED I/O MODULE DO NOT GET OVERLAYED BY THE TRANSIENT
;       PROGRAM AREA'S BUFFER SPACES. THE UTILITY, WHEN LOADED PERFORMS
;       A CHECK TO VERIFY WHETHER A RELOCATED MODULE IS ALREADY PRESENT
;       IN MEMORY. THE ALREADY PRESENT CHECK IS DONE VIA A SPECIALLY DEFINED
;       BDOS CALL THAT REQUESTS THE OPENING OF A FILE WITH THE SPECIALLY
;       DEFINED NAME SEQUENCE OF "A,,,,,,,,.,,," AS THE FILE NAME WHERE (A)
;       IS A REQUEST NUMBER FOR PRESENCE CHECKING. AS THIS CHARACTER SEQUENCE
;       IS AN ILLEGAL FILE NAME SEQUENCE, THE CHECK PROGRAM WILL TRAP THE NAME
;       AND RETURN A ZERO BYTE IN THE (A) REGISTER IF THE MODULE IS PRESENT.
;       IF THE ADDRESS BYTE IN THE FIRST BYTE OF THE FCB IS NOT RECOGNIZED,
;       THEN THE MODULE PASSES THE OPEN FILE REQUEST ON TO THE NEXT HIGHER
;       LEVEL BDOS CALL. IN ANY CASE THE NON PRESENCE OF A FILE BY THE NAME
;       OF "A,,,,,,,,.,,," IS VIRTUALLY ASSURED TO CAUSE THE BDOS TO RETURN
;       A NOT FOUND "OFFH" ERROR CODE IN THE (A) REGISTER. THIS WOULD INDICATE
;       THE ABSENSE OF THE MODULE BEING CHECKED FOR.
;
;
;       THIS RAM DISK DRIVER PACKAGE
;       PROGRAM IS COPYRIGHT PROTECTED BY:
;
;       COPYRIGHT (C) 1982
;
;       MICRO RESOURCES
;       2468 HANSEN COURT
;       SIMI VALLEY, CALIFORNIA 93065
;       (805) 527-7922
;
;*************************************************************************
;
;
;
;DEFINE TRUE AND FALSE ASSEMBLY PARAMETERS
;
TRUE    EQU     -1                      ;DEFINE TRUE
FALSE   EQU     NOT TRUE                ;DEFINE FALSE
;
;
;DEFINE RAMDISK MODULE SELECT ADDRESS AS SPECIAL VALUE
;
MODADDR EQU     08AH                    ;ADDRESS OF THIS MODULE
;
;
;CP/M BDOS INTERFACE EQUATES
;
        ASEG
BOOT    EQU     0000H                   ;FIXED BOOT ADDRESS
BDOS    EQU     0005H                   ;FIXED BDOS ADDRESS
DEFFCB  EQU     005CH                   ;DEFAULT FCB LOCATION
DEFBUF  EQU     0080H                   ;DEFAULT SYSTEM BUFFER
RESET   EQU     13                      ;RESET DISK SYSTEM
OPEN    EQU     15                      ;OPEN FILE
STDMA   EQU     26                      ;SET DMA ADDRESS
;
;
;ASCII CHARACTER DEFINITIONS
;
LF      EQU     00AH                    ;ASCII LINE FEED CHARACTER
CR      EQU     00DH                    ;ASCII CARRIAGE RETURN CHARACTER
;
;
;SECTOR DEBLOCKING ALGORITHMS FOR CP/M 2.2
;
        MACLIB  DISKDEF
;
SMASK   MACRO   HBLK                    ;UTILITY MACRO TO COMPUTE SECTOR MASK
;
;       COMPUTE LOG2(HBLK), RETURN @X AS RESULT
;       (2 ** @X = HBLK ON RETURN)
;
@Y      SET     HBLK
@X      SET     0
;
;       COUNT RIGHT SHIFTS OF @Y UNTIL = 1
;
        REPT    8
        IF      @Y = 1
        EXITM
        ENDIF
;
;       @Y IS NOT 1, SHIFT RIGHT ONE POSITION
;
@Y      SET     @Y SHR 1
@X      SET     @X + 1
        ENDM
        ENDM
;
;
;BDOS CONSTANTS ON ENTRY TO "WRITE"
;
WRALL   EQU     0                       ;WRITE TO ALLOCATED BLOCK
WRDIR   EQU     1                       ;WRITE TO DIRECTORY
WRUAL   EQU     2                       ;WRITE TO UNALLOCATED BLOCK
```

(continued next page)

```
;
;
;
;CP/M 2.2 TO HOST DISK CONSTANTS
;
BLKSIZ  EQU     2048                ;CP/M ALLOCATION SIZE
HSTSIZ  EQU     256                 ;HOST DISK SECTOR SIZE
;
HDSPT   EQU     32                  ;HOST HARD DISK 256 BYTE SECTORS/TRACK
HSTBLK  EQU     HSTSIZ/128          ;CP/M SECTS/HOST BUFF
;
SECMSK  EQU     HSTBLK-1            ;SECTOR MASK
        SMASK   HSTBLK              ;COMPUTE SECTOR MASK
SECSHF  EQU     @X                  ;LOG2(HSTBLK)
;
;SECTOR SKEW INTERLACE FACTOR
;
SKEW    EQU     00                  ;SECTOR SKEW FACTOR
;
SECSIZ  EQU     256                 ;NUMBER OF BYTES IN DISK RECORD
;
;
;******************************************************************************
;
;
        CSEG                        ;SET ORIGIN TO ZERO FOR RELOCATABLE
                                    ;ASSEMBLY BY RMAC
;
;
;SETUP STORAGE FOR THE RAM DISK DRIVE DATA BUFFER BELOW THE RELOCATED
;ADDON MODULE.
;
RAMBUF  EQU     $-(10*8*SECSIZ)     ;SIZE SET AT 10 TRACKS OF EIGHT 256
                                    ;BYTE SECTORS PER TRACK
BUFSIZ  EQU     $-RAMBUF            ;SIZE OF BUFFER FOR INIT CLEAR
;
;
;FIRST TIME START UP ENTRY POINT FOR THE RAM DISK AUTO RELOCATING
;I/O MODULE. ENTRY HERE ASSURES PRESENSE OF CP/M 2.2.
;
        JMP     CHKPRES             ;GO CHECK IF A MODULE OF
                                    ;SAME FUNCTION ADDRESS IS PRESENT
                                    ;IN SYSTEM
;
;
;SUBSTITUTE BDOS ENTRY POINT. EXECUTION ADDRESS IS PLACED HERE
;FROM LOCATION 6 & 7 BY THE START UP MODULE PROVIDED THIS
;MODULE IS DETERMINED TO NOT ALREADY BE IN MEMORY.
;
TO$BDOS:
        JMP     $-$                 ;ENTER ADDRESS AT STARTUP
;
;
;START UP CHECK ROUTINE TO SEE IF THIS SOFTWARE WAS ALREADY
;LOADED BY A PREVIOUS OPERATOR COMMAND.
;
CHKPRES:
        MVI     C,OPEN              ;ATTEMPT TO OPEN FILE "A,,,,,,,,.,,,"
        LXI     D,CHKFCB            ;POINT AT THE CHECK FCB
        CALL    BDOS                ;CALL NORMAL BDOS ADDRESS
        ORA     A
        JNZ     NOT$PRES            ;NON ZERO RETURN MODULE IS NOT
                                    ;..PRESENT
        LXI     D,PRESMSG           ;POINT TO PRESENT MESSAGE
        MVI     C,9                 ;PRINT FUNCTION CODE
        CALL    BDOS                ;PRINT ALREADY PRESENT MESSAGE
        RET                         ;SIMPLE RETURN TO THE CCP
;
CHKFCB:
        DB      0,MODADDR,´,,,,,,,,,,´,0,0,0,0
        DS      16
        DB      0
;
PRESMSG:
        DB      CR,LF,´MICRO RESOURCES RAM Disk Already Active´,´$´
;
;
;HERE IF THIS RELOCATED MODULE IS NOT PRESENT IN MEMORY
;
NOT$PRES:
        POP     H                   ;COMPUTE CCP RE-ENTRY POINT
        PUSH    H
        LXI     D,075CH             ;NEGATIVE OFFSET TO CCP ENTRY
                                    ;WITH NO AUTO LOAD
        MOV     A,L
        SUB     E
        MOV     L,A
        MOV     A,H
        SBB     D
        MOV     H,A
        SHLD    CCP$ENT             ;SAVE THAT ENTRY ADDRESS
;
        LHLD    BDOS+1              ;GET PREVIOUS BDOS ADDRESS
        SHLD    TO$BDOS+1           ;SET TO LOCAL REFERENCE VECTOR
;
;
;COMPUTE THE NEW RAM TOP OF TPA TO SET IN A JUMP ONE PAGE BELOW
;FOR PLACEMENT OF BASE OF THE RAM DISK DRIVE FOR BDOS REFERENCE
;
        LXI     H,RAMBUF
        DCR     H                   ;ONE PAGE DOWN
        MVI     L,06H               ;AT CP/M´S BDOS LOOK ALIKE
        SHLD    BDOS+1              ;BASE+6
        LXI     D,BDOS$SCAN
        MVI     M,0C3H              ;SET A JUMP AT TPA TOP
        INX     H
        MOV     M,E                 ;LOW BYTE OF ENTRY POINT
        INX     H
        MOV     M,D                 ;HIGH BYTE OF ENTRY POINT
;
;
;INITIALIZE ALL ITEMS FOR USE IN THIS I/O HANDLER
;
        MVI     B,ENDZ-STARTZ       ;ZERO DATA AREA IN PARAMETER TABLE
        LXI     H,STARTZ
ZLP:
        MVI     M,00H               ;PUT IN A ZERO PARM BYTE
        INX     H                   ;POINT TO NEXT BYTE TO BE ZEROED
        DCR     B                   ;CHECK BYTE COUNT TO SEE IF DONE
        JNZ     ZLP
;
;
;INITIALIZE THE RAM BUFFER TO LOOK LIKE FRESH FORMATTED DRIVE
;
        LXI     B,BUFSIZ            ;RAM DRIVE SIZE
        LXI     H,RAMBUF            ;DRIVE BASE ADDRESS
E5LP:
        MVI     M,0E5H              ;STORE AN E5 BYTE
        INX     H                   ;BUMP POINTER
        DCX     B                   ;DEC BYTE COUNT
        MOV     A,B
        ORA     C
        JNZ     E5LP
;
        LXI     H,RAMSEL            ;DISABLE DRIVE SELECT FOR RAM DISK
        MVI     M,00H
;
        CALL    MOVDN               ;MOVE DOWN THE BIOS VECTOR TABLE
        LXI     H,BOOTENT           ;SET MOVED DOWN TABLE TO LOCAL BOOT HANDLER
        SHLD    BWBOOT+1
;
        CALL    PRTMSG              ;PRINT SIGNON MESSAGE
;
        DB      CR,LF,´Micro Resources RAM Disk Demonstration´
        DB      CR,LF,´Add-on Access Module Version 1.0 of 6/14/82´
        DB      CR,LF,´Copyright (C) 1982 Micro Resources´
        DB      CR,LF,0
;
;
;
;HAVE THIS UTILITY QUEUE BOTH BIOS AND THIS DRIVER TO THE SAME
;CP/M DATA BUFFER ADDRESS
;
        LXI     D,DEFBUF            ;USE DEFAULT BUFFER
        MVI     C,STDMA             ;SET DMA FUNCTION CODE
        CALL    TO$BDOS
;
;
;RETURN TO CCP VIA THE OLD DEFINED REENTRY POINT
;
CCPGO:
        LXI     H,RAMBUF
        DCR     H                       ;ONE PAGE DOWN
        MVI     L,06H                   ;AT CP/M´S BDOS LOOK ALIKE
        SHLD    BDOS+1              ;BASE+6
        LHLD    CCP$ENT.            ;GET THE CCP ENTRY POINT
        LDA     004H                ;GET CURRENTLY LOGGED DRIVE
        MOV     C,A
        PCHL
;
;
;NEW WARM BOOT ENTRY LOCATION THAT RESETS THE DISK SYSTEM
;AND TRANSFERS CONTROL BACK TO THE ALREADY PRESENT CCP
;
BOOTENT:
        JMP     CCPGO               ;NOW GO BACK TO THE CCP
;
;
;HERE FROM A BDOS ENTRY TO TRAP FILE OPEN I/O TO CHECK FOR
;MODULE PRESENT CHECK.
;
BDOS$SCAN:
        PUSH    D                   ;SAVE CALLERS PARAMETERS
        PUSH    B
        MOV     A,C                 ;GET FUNCTION CODE TO A
        CPI     OPEN                ;SEE IF THIS IS AN OPEN FUNCTION
        JNZ     CHKFAIL
        INX     D                   ;POINT TO FCB CHECK BYTE
        LXI     H,10                ;SET SCAN COUNTER TO FAKE FILE NAME END
        DAD     D
        MVI     B,10                ;NUMBER OF "," TO CHECK FOR
SCAN$LOOP:
        MOV     A,M                 ;GET FILE NAME CHARACTER
        CPI     ´,´
        JNZ     CHKFAIL             ;PASS ON IF CHECK FAIL
        DCX     H                   ;DECREASE BUFFER POINTER
        DCR     B
        JNZ     SCAN$LOOP           ;CHECKED ALL POSSIBLE CHARS YET
        MOV     A,M                 ;CHECK IF ADDRESS BYTE IS OURS
        CPI     MODADDR
        JNZ     CHKFAIL             ;BALE OUT IF NOT
        XRA     A                   ;RETURN ZERO BYTE IF ALL CHECK VALID
        POP     B
        POP     D
        RET                         ;BACK TO PRESENT CHECKER
;
CHKFAIL:                            ;PROPER OPEN CHECK FAIL
        POP     B
        POP     D
        JMP     TO$BDOS             ;OFF TO THE NORMAL BDOS ROUTINE
;
;
;
XFRTAB:
;
;       SUBSTITUTE BIOS VECTOR TABLE. THIS JUMP TABLE VECTORS ALL CP/M
;       DISK I/O TO THIS TRANSIENT MODULE FIRST. TABLE IS PUT INTO THE
;       BIOS VECTOR TABLE POSITION BY A CALL TO THE SUBROUTINE "MOVDN"
;
;
        JMP     BCBOOT              ;TO NORMAL BIOS COLD BOOT ROUTINE
        JMP     BOOTENT             ;TO LOCAL WARM BOOT HANDLER
        JMP     BCSTAT              ;TO NORMAL BIOS CONSOLE STATUS CHECK
        JMP     BCIN                ;TO NORMAL BIOS CONSOLE INPUT
        JMP     BCOUT               ;TO NORMAL BIOS CONSOLE OUTPUT
        JMP     BLOUT               ;TO NORMAL BIOS LPT OUTPUT
        JMP     BPUN                ;TO NORMAL BIOS PUNCH OUTPUT
        JMP     BRDR                ;TO NORMAL BIOS READER INPUT
        JMP     HOME                ;MOVE DISK TO TRACK ZERO
        JMP     SELDSK              ;SELECT DISK DRIVE
        JMP     SETTRK              ;SEEK TO TRACK IN REG A
        JMP     SETSEC              ;SET SECTOR NUMBER
        JMP     SETDMA              ;SET DISK STARTING ADR
        JMP     READ                ;READ SELECTED SECTOR
        JMP     WRITE               ;WRITE SELECTED SECTOR
        JMP     BLSTST              ;GO RIGHT TO NORMAL BIOS FOR THIS I/O
        JMP     SECTRAN             ;SECTOR TRANSLATE
;
```

```
;
LOCTAB:
;
;       LOCAL COPY OF THE ORIGINAL BIOS DISK I/O VECTOR TABLE
;       INITIALIZED BY CALLING THE "MOVDN" SUBROUTINE.
;
BCBOOT:
        JMP     $-$             ;TO BIOS COLD BOOT ROUTINE
BWBOOT:
        JMP     $-$             ;TO BIOS WARM BOOT ROUTINE
BCSTAT:
        JMP     $-$             ;TO BIOS CONSOLE STATUS CHECK
BCIN:
        JMP     $-$             ;TO BIOS CONSOLE INPUT
BCOUT:
        JMP     $-$             ;TO BIOS CONSOLE OUTPUT
BLOUT:
        JMP     $-$             ;TO BIOS LPT OUTPUT
BPUN:
        JMP     $-$             ;TO BIOS PUNCH OUTPUT
BRDR:
        JMP     $-$             ;TO BIOS READER INPUT
BHOME:
        JMP     $-$             ;TO BIOS HOME DISK ROUTINE
BSELDSK:
        JMP     $-$             ;TO BIOS SELECT DISK ROUTINE
BSETTRK:
        JMP     $-$             ;TO BIOS SET TRACK ROUTINE
BSETSEC:

        JMP     $-$             ;TO BIOS SET SECTOR ROUTINE
BSETDMA:
        JMP     $-$             ;TO BIOS SET DMA ADDRESS ROUTINE
BREAD:
        JMP     $-$             ;TO BIOS SECTOR READ ROUTINE
BWRITE:
        JMP     $-$             ;TO BIOS SECTOR WRITE ROUTINE
BLSTST:
        JMP     $-$             ;TO BIOS LIST STATUS ROUTINE
BSTRAN:
        JMP     $-$             ;TO BIOS SECTOR TRANSLATE ROUTINE
;
;
;SUBROUTINE TO INTERCHANGE BIOS DISK I/O VECTOR TABLE ENTRIES WITH
;THOSE CONTAINED LOCALLY.
;
TABSIZ  EQU     17*3            ;TABLE SIZE TO INITIALIZE WITH 17 JMP´S
;
;
MOVDN:
        LHLD    BOOT+1          ;GET ORIGINAL WARM BOOT VECTOR POINTER
        DCX     H               ;ADJUST TO BASE OF COLD BOOT VECTOR
        DCX     H
        DCX     H
        MVI     A,TABSIZ        ;SET BYTE COUNT TO MOVE
        STA     BYTCNT
        LXI     D,LOCTAB        ;POINT TO LOCAL TABLE FILL FROM ABOVE
        LXI     B,XFRTAB        ;POINT TO TABLE TO MOVE UP
MDLP:
        MOV     A,M             ;GET A BIOS TABLE BYTE
        STAX    D               ;PUT IN LOCAL COPY TABLE
        LDAX    B               ;GET BYTE OF PATCH TABLE
        MOV     M,A             ;PUT PATCH BYTE INTO BIOS POSITION
        INX     H               ;MOVE UP TO NEXT BYTE
        INX     D
        INX     B
        LDA     BYTCNT          ;SEE IF DONE YET
        DCR     A
        STA     BYTCNT
        JNZ     MDLP            ;CONTINUE IF NOT DONE YET
        RET
;
BYTCNT:
        DB      0               ;LOCAL MOVE BYTE COUNTER
;
;
;**********************************************************************
;
;PARAMETER TABLE FOR TPA REASIDEN RAM DRIVE
;
        DISKS   1               ;ONE LOGICAL DRIVES SUPPORTED
        DISKDEF 0,1,16,,1024,20,32,0,0         ;P: RAM DRIVE
;
SELDSK:
        MOV     A,C             ;GET NEW UNIT NUMBER
        CPI     ´P´-041H        ;IS THIS OUR DRIVE?
        JZ      SDSK1           ;IF SO THEN GIVE THEM A PARAMETER POINTER
;
        XRA     A               ;IF NOT CLEAR THE ZOBEX DRIVE SELECT FLAG
        STA     RAMSEL
        JMP     BSELDSK         ;IF NOT FOR US THEN LET BIOS HAVE SELECT
;
;
;HERE IF DRIVE SELECT WAS FOR THIS PIECE OF SOFTWARE
;
SDSK1:
        SUI     ´P´-041H        ;SET SEKDSK TO THE HEAD SELECT CODE FOR
        STA     SEKDSK          ;..RAM DISK DRIVE
;
        PUSH    PSW
        MVI     A,OFFH          ;SET THE RAM DRIVE SELECT FLAG
        STA     RAMSEL
        POP     PSW
;
        LXI     H,DPBASE        ;PASS BACK DISK PARAMETER BASE
        XRA     A               ;SET A REG. = 00
        RET                     ;RETURN FROM SELDSK
;
;
;DO DIGITAL RESEARCH BUFFER PURGE IF NEED BE AND BALE OUT
;NO RESTORE MEMORY I SRANDOM ACCESS
;
HOME:
        LDA     RAMSEL          ;SEE IF RESTORE FOR US
        ORA     A
        JZ      BHOME           ;NO MUST BE FOR BIOS DRIVE
;
        LDA     HSTWRT          ;CHECK HOST ACTIVE WRITE FLAG
        ORA     A
        JNZ     HOMEIT
        STA     HSTACT
HOMEIT:
        RET
;
;
;SET TRACK NUMBER SPECIFIED BY B&C REGS.
;
SETTRK:
        LDA     RAMSEL          ;SEE IF TRACK FOR US
        ORA     A
        JZ      BSETTRK         ;TO PROM IF NOT LOCAL
;
        MOV     H,B
        MOV     L,C
        SHLD    SEKTRK          ;TRACK TO EMULATE
        RET
;
;
;
;TRANSLATE THE SECTOR GIVEN BY B&C REGS.
;
;       NO TRANSLATE DONE AT THIS TIME. WE WILL NOT NEED TO TRANSLATE
;       RAM DISK SECTOR BECAUSE RAM HAS NO ROTATIONAL LATENCY
;
SECTRAN:
        LDA     RAMSEL          ;SEE IF SECTRAN FOR US
        ORA     A
        JZ      BSTRAN          ;TO BIOS IF NOT LOCAL
;
        MOV     H,B
        MOV     L,C
        RET                     ;RETURN FROM SECTRAN
;
;
;SET DISK SECTOR NUMBER
;
SETSEC:
        LDA     RAMSEL          ;SEE IF SECTOR FOR US
        ORA     A
        JZ      BSETSEC         ;TO PROM IF NOT LOCAL
;
        MOV     A,C             ;GET SECTOR NUMBER
        STA     SEKSEC          ;SECTOR TO EMULATE
        RET                     ;RETURN FROM SETSEC
;
;
;SET DISK DMA ADDRESS
;
SETDMA:
        PUSH    H
        MOV     H,B             ;MOVE B&C TO H&L
        MOV     L,C
        SHLD    DMAADR          ;PUT AT DMA ADR ADDRESS
        POP     H
        JMP     BSETDMA         ;TELL BIOS DMA ADDRESS
;
;
;READ THE SELECTED CP/M 2.2 SECTOR
;
READ:
        LDA     RAMSEL          ;SEE IF OPERATION FOR US
        ORA     A
        JZ      BREAD           ;GO READ IN BIOS IF NOT FOR US LOCAL
;
        XRA     A               ;CLEAR UNALLOCATED COUNT
        STA     UNACNT
        MVI     A,1
        STA     READOP          ;READ OPERATION
        STA     RSFLAG          ;MUST READ DATA
        MVI     A,WRUAL
        STA     WRTYPE          ;TREAT AS UNALLOCCATED
        JMP     RWOPER          ;TO PERFORM THE READ
;
;
;WRITE THE SELECTED CP/M 2.2 SECTOR
;
WRITE:
        LDA     RAMSEL          ;IS THIS WRITE FOR HERE
        ORA     A
        JZ      BWRITE          ;TO BIOS IF SO
;
        XRA     A               ;0 TO A REG.
        STA     READOP          ;NOT A READ OPERATION
        MOV     A,C             ;WRITE TYPE IN C
        STA     WRTYPE
        CPI     WRUAL           ;WRITE UNALLOCATED?
        JNZ     CHKUNA          ;CHECK FOR UNALLOCATED
;
;
;WRITE TO UNALLOCATED, SET PARAMETERS
;
        MVI     A,BLKSIZ/128    ;NEXT UNALLOCATED RECORDS
        STA     UNACNT
        LDA     SEKDSK          ;DISK TO SEEK
        STA     UNADSK          ;UNADSK = SEKDSK
        LHLD    SEKTRK
        SHLD    UNATRK          ;UNATRK = SECTRK
        LDA     SEKSEC
        STA     UNASEC          ;UNASEC = SEKSEC
;
;
;CHECK FOR WRITE TO UNALLOCATED SECTOR
;
CHKUNA:
        LDA     UNACNT          ;ANY UNALLOCATED REMAINING?
        ORA     A
        JZ      ALLOC           ;SKIP IF NOT
;
;
;MORE UNALLOCATED RECORDS REMAIN
;
        DCR     A               ;UNACNT = UNACNT-1
        STA     UNACNT
        LDA     SEKDSK          ;SAME DISK?
        LXI     H,UNADSK
        CMP     M               ;SEKDSK = UNADSK?
        JNZ     ALLOC           ;SKIP IF NOT
;
```

(continued next page)

```
;
;DISKS ARE THE SAME
;
        LXI     H,UNATRK
        CALL    SEKTRKCMP       ;SEKTRK = UNATRK?
        JNZ     ALLOC           ;SKIP IF NOT
;
;
;TRACKS ARE THE SAME
;
        LDA     SEKSEC          ;SAME SECTOR?
        LXI     H,UNASEC
        CMP     M               ;SEKSEC = UNASEC?
        JNZ     ALLOC           ;SKIP IF NOT
;
;
;MATCH, MOVE TO NEXT SECTOR FOR FUTURE REFERENCE
;
        INR     M               ;UNASEC = UNASEC+1
        MOV     A,M             ;END OF TRACK?
        PUSH    B
        MVI     B,HDSPT         ;USE HARD DISK SPT
        CMP     B
        POP     B
        JC      NOOVF           ;SKIP IF NO OVERFLOW
;
;
;OVERFLOW TO NEXT TRACK
;
        MVI     M,0             ;UNASEC = 0
        LHLD    UNATRK
        INX     H
        SHLD    UNATRK          ;UNATRK = UNATRK+1
;
;
;MATCH FOUND, MARK AS UNNECESSARY READ
;
NOOVF:
        XRA     A               ;0 TO A REG.
        STA     RSFLAG          ;RSFLAG = 0
        JMP     RWOPER          ;TO PERFORM THE WRITE
;
;
;NOT AN UNALLOCATED RECORD, REQUIRES PRE-READ
;
ALLOC:
        XRA     A               ;0 TO A REG.
        STA     UNACNT          ;UNACNT = 0
        INR     A               ;1 TO A REG.
        STA     RSFLAG          ;RSFLAG = 1
;
;
;COMMON CODE FOR READ AND WRITE FOLLOWS:
;
RWOPER:                         ;ENTER HERE TO PERFORM THE READ/WRITE
        XRA     A               ;ZERO TO A REG.
        STA     ERFLAG          ;NO ERRORS (YET)
        LDA     SEKSEC          ;COMPUTE HOST SECTOR
;
        REPT    SECSHF
        ORA     A               ;CARRY = 0
        RAR                     ;SHIFT RIGHT
        ENDM
;
;
;LET BIOS PRETEND THAT SECTORS ARE NUMBERED FROM 1 TO AVOID
;OTHER PROBLEMS IN THE "SEKHST" SECTOR NUMBER VALUE
;
        INR     A
        STA     SEKHST          ;HOST SECTOR TO SEEK
;
;
;ACTIVE HOST SECTOR?
;
        LXI     H,HSTACT        ;HOST ACTIVE FLAG
        MOV     A,M
        MVI     M,1             ;ALWAYS BECOMES 1
        ORA     A               ;WAS IT ALREADY?
        JZ      FILHST          ;FILL HOST IF NOT
;
;
;HOST BUFFER ACTIVE, SAME AS SEEK BUFFER?
;
        LDA     SEKDSK
        LXI     H,HSTDSK        ;SAME DISK?
        CMP     M               ;SEKDSK = HSTDSK?
        JNZ     NOMATCH
;
;
;SAME DISK, SAME TRACK?
;
        LXI     H,HSTTRK
        CALL    SEKTRKCMP       ;SEKTRK = HSTTRK?
        JNZ     NOMATCH
;
;
;SAME DISK, SAME TRACK, SAME BUFFER?
;
        LDA     SEKHST
        LXI     H,HSTSEC        ;SEKHST = HSTSEC?
        CMP     M
        JZ      MATCH           ;SKIP IF MATCH
;
;
;PROPER DISK, BUT NOT CORRECT SECTOR
;
NOMATCH:
        LDA     HSTWRT          ;HOST WRITTEN?
        ORA     A
        CNZ     WRITEHST        ;CLEAR HOST BUFF
;
;
;MAY HAVE TO FILL THE HOST BUFFER
;
FILHST:
        LDA     SEKDSK
        STA     HSTDSK
        LHLD    SEKTRK
        SHLD    HSTTRK
```

```
        LDA     SEKHST
        STA     HSTSEC
        LDA     RSFLAG          ;NEED TO READ?
        ORA     A
        CNZ     READHST         ;YES, IF 1
        XRA     A               ;0 TO A REG.
        STA     HSTWRT          ;NO PENDING WRITE
;
;
;COPY DATA TO OR FROM BUFFER
;
MATCH:
        LDA     SEKSEC          ;MASK BUFFER NUMBER
        ANI     SECMSK          ;LEAST SIGNIF BITS
        MOV     L,A             ;READY TO SHIFT
        MVI     H,0             ;DOUBLE COUNT
;
        REPT    7               ;SHIFT LEFT 7
        DAD     H
        ENDM
;
;
;HL HAS RELATIVE HOST BUFFER ADDRESS
;
        LXI     D,HSTBUF
        DAD     D               ;HL = HOST ADDRESS
        XCHG                    ;NOW IN DE
        LHLD    DMAADR          ;GET/PUT CP/M DATA
        MVI     C,128           ;LENGTH OF MOVE; CP/M SECTOR SIZE
        LDA     READOP          ;WHICH WAY?
        ORA     A
        JNZ     RWMOVE          ;SKIP IF READ
;
;
;WRITE OPERATION, MARK AND SWITCH DIRECTION
;
        MVI     A,1
        STA     HSTWRT          ;HSTWRT = 1
        XCHG                    ;SOURCE/DESTINATION SWAP
;
;
;C INITIALLY 128, DE IS SOURCE, HL IS DESTINATION
;
RWMOVE:
        LDAX    D               ;SOURCE CHARACTER
        INX     D
        MOV     M,A             ;TO DESTINATION
        INX     H
        DCR     C               ;LOOP 128 TIMES
        JNZ     RWMOVE
;
;
;DATA HAS BEEN MOVED TO/FROM HOST BUFFER
;
        LDA     WRTYPE          ;WRITE TYPE
        CPI     WRDIR           ;TO DIRECTORY?
        LDA     ERFLAG          ;IN CASE OF ERRORS
        RNZ                     ;NO FURTHER PROCESSING
;
;
;CLEAR HOST BUFFER FOR DIRECTORY WRITE
;
        ORA     A               ;ERRORS?
        RNZ                     ;SKIP IF SO
        XRA     A               ;0 TO A REG.
        STA     HSTWRT          ;BUFFER WRITTEN
        CALL    WRITEHST
        LDA     ERFLAG
        RET
;
;
;UTILITY SUBROUTINE FOR 16-BIT COMPARE
;
SEKTRKCMP:                      ;HL = .UNATRK OR .HSTTRK, COMPARE
                                ;..WITH SEKTRK
        XCHG
        LXI     H,SEKTRK
        LDAX    D               ;LOW BYTE COMPARE
        CMP     M               ;SAME?
        RNZ                     ;RETURN IF NOT
;
;
;LOW BYTES EQUAL, TEST HIGH FIRST
;
        INX     D
        INX     H
        LDAX    D
        CMP     M               ;SETS FLAGS
        RET
;
;
WRITEHST:                       ;PERFORMS THE PHYSICAL WRITE
                                ;TO THE HOST DISK
;
;HSTDSK = HOST DISK NUMBER, HSTTRK = HOST TRACK NUMBER,
;HSTSEC = HOST SECT NUMBER. WRITE "HSTSIZ" BYTES
;FROM HSTBUF AND RETURN ERROR FLAG IN ERFLAG.
;RETURN ERFLAG NON-ZERO IF ERROR
;
WRTSEC:
        CALL    RIOPB           ;SETUP RAM DISK IOPB
                                ;..FROM BIOS VARIABLES
        CALL    RWRITE          ;GO WRITE RAM DISK SECTOR
        XRA     A
        STA     ERFLAG          ;RESET ERROR FLAG
        RET                     ;RETURN FROM "WRITEHST", IF O.K.
;
;
READHST:                        ;PERFORMS THE PHYSICAL READ FROM
                                ;..THE HOST DISK
;
;HSTDSK = HOST DISK NUMBER, HSTTRK = HOST TRACK NUMBER,
;HSTSEC = HOST SECT NUMBER. READ "HSTSIZ" BYTES
;INTO HSTBUF AND RETURN ERROR FLAG IN ERFLAG.
;
READSEC:
        CALL    RIOPB           ;SET RAM DISK IOPB
        CALL    RREAD           ;GO READ SECTOR
        XRA     A
        STA     ERFLAG
        RET
;
;
;ROUTINE TO SETUP THE RAM DISK SOFTWARE ADDRESS VALUES VALUES
;BASED UPON THE CP/M LOGICAL VALUES
;
RIOPB:
        LXI     H,HSTBUF        ;POINT TO HOST BUFFER ADDRESS
        SHLD    XFRPNT
        LDA     HSTTRK          ;CONVERT CP/M TRACK AND SECTOR TO
        ADD     A               ;..256 BYTE RAM PAGE ADDRESS INDEX
        ADD     A
        ADD     A               ;TRACK BOUNDARY INDEX
        MOV     B,A             ;SAVE TO GET SECTOR
        LDA     HSTSEC
        DCR     A               ;HOST BIOS ABOVE THINKS SECTORS START AT 1
        ADD     B               ;SECTOR 256 BYTE INDEX TO RAM DISK
        MOV     D,A
        MVI     E,00H
        LXI     H,RAMBUF        ;POINT TO RAM BUFFER FOR DISK
        DAD     D               ;(HL) IS RAM ADDRESS FOR MOVE
        SHLD    RAMADDR
        RET
;
;
;INLINE PRINT OF MESSAGE TILL A ZERO
;
PRTMSG:
        XTHL                    ;SAVE HL, GET MSG POINTER
;
PRTMLP:
        MOV     C,M             ;GET CHARACTER
        INX     H               ;INCREMENT POINTER TO NEXT CHAR
                                ;..OR RETURN ADDRESS
        MOV     A,C             ;CHECK IF ZERO END
        ORA     A
        JZ      PMXIT           ;EXIT IF ZERO
;
        CALL    BCOUT           ;OUTPUT IT
        JMP     PRTMLP          ;GO CHECK/DO NEXT CHAR
;
PMXIT:
        XTHL                    ;RESTORE HL, RET ADDR
        RET                     ;RET PAST MSG
;
;
;*****************************************************************************
;*****************************************************************************
;
;       MICRO RESOURCES TPA RESIDENT RAM DRIVE I/O ROUTINES
;
;*****************************************************************************
;*****************************************************************************
;
;
;WRITE ONE 256 BYTE RECORD FROM THE HSTBUF TO THE RAM DRIVE
;
RWRITE:
        LHLD    RAMADDR         ;POINT AT THE RAM DRIVE ADDRESS
        XCHG                    ;TO (DE) AS "TO" ADDRESS
        LHLD    XFRPNT          ;GET (HL) AS "FROM" ADDRESS
        LXI     B,SECSIZ        ;PHYSICAL HOST SECTOR SIZE
;
RWXFR:
        MOV     A,M             ;GET A DATA BYTE
        STAX    D               ;PUT AT DESTINATION
        INX     H               ;BUMP POINTERS
        INX     D
        DCX     B               ;DECREMENT BUFFER SIZE COUNTER
        MOV     A,B
        ORA     C               ;CHECK IF ALL MOVED YET
        JNZ     RWXFR
        RET                     ;DONE WITH READ ALREADY
;
;
;READ ONE 256 BYTE RECORD FROM THE RAM DRIVE TO THE HOST BUFFER
;
RREAD:
        LHLD    XFRPNT          ;POINT AT THE HOST BUFFER AS
        XCHG                    ;..(DE) AS "TO" ADDRESS
        LHLD    RAMADDR         ;GET (HL) REAM ADDRESS AS "FROM" ADDRESS
        LXI     B,SECSIZ        ;BYTES PER SECTOR COUNTER
        JMP     RWXFR           ;GO DO THE TRANSFER
;
;
;*****************************************************************************
;
;       STORAGE AREA FOR VARIABLES BEGINS HERE...
;
;
;RELOCATION POINTER STORAGE AREA
;
CCP$ENT:
        DS      2               ;STORE CCP RE-ENTRY POINTER HERE
                                ;TABLE POINTER HERE
;
;RAM DISK DRIVE ACCESS PARAMETER BLOCK
;
RAMADDR:
        DS      2               ;RAM DRIVE POINTER ADDRESS
XFRPNT:
        DS      2               ;READ/WRITE ROUTINE DATA BUFFER POINTER
;
;THE NEXT SEVERAL BYTES, BETWEEN STARTZ AND
;ENDZ, ARE SET TO ZERO AT MODULE INITIALIZATION
;
STARTZ  EQU     $               ;START OF ZEROED AREA
;
;
;HOST DISK BLOCKING/DE-BLOCKING DATA AREA
;
SEKDSK:
        DS      1               ;SEEK DISK NUMBER
SEKTRK:
        DS      2               ;SEEK TRACK NUMBER
SEKSEC:
        DS      1               ;SEEK SECTOR NUMBER
;
HSTDSK:
        DS      1               ;HOST DISK NUMBER
HSTTRK:
        DS      2               ;HOST TRACK NUMBER
HSTSEC:
        DS      1               ;HOST SECTOR NUMBER
;
SEKHST:
        DS      1               ;SEEK SHR SECSHF
HSTACT:
        DS      1               ;HOST ACTIVE FLAG
HSTWRT:
        DS      1               ;HOST WRITTEN FLAG
;
UNACNT:
        DS      1               ;UNALLOCATED RECORD COUNT
UNADSK:
        DS      1               ;LAST UNALLOCATED DISK
UNATRK:
        DS      2               ;LAST UNALLOCATED TRACK
UNASEC:
        DS      1               ;LAST UNALLOCATED SECTOR
;
ERFLAG:
        DS      1               ;ERROR REPORTING
RSFLAG:
        DS      1               ;READ SECTOR FLAG
READOP:
        DS      1               ;1 IF READ OPERATION
WRTYPE:
        DS      1               ;WRITE OPERATION TYPE
DMAADR:
        DS      2               ;DISK DMA TRANSFER ADDRESS
RAMSEL:
        DS      1               ;LOCAL DISK SELECTED FLAG
;
ENDZ    EQU     $               ;END OF ZEROED AREA
;
;
;HOST DATA BUFFER  MEMORY AREA
;
;
HSTBUF:
        DS      HSTSIZ.         ;HOST BUFFER
;
;
;
;SCRATCH RAM AREA FOR BDOS USE
;
        ENDEF                   ;LET DISKDEF FIXUP BDOS BUFFERS
;
        END
;
;
;+++...END OF FILE
```

Last Chance

We're looking forward to hearing from any of you October subscribers who haven't called or written. If your subscription started with the October '81 issue you should have received a letter and reader survey from us, urging you to renew. You can see that **Lifelines/The Software Magazine** has given you value this past year, and we have more goodies planned. Communications, graphics, the C language, and new software for the IBM PC are a few of the subjects to be covered in the coming months.

Please send your check right away. Or get out your VISA or MasterCard and call **Lifelines/The Software Magazine** Subscription Dept. at (212) 722-1700. The address is: 1651 Third Ave., New York, N.Y. 10028.

(P.S. If you've already renewed, but still received a second or third notice, don't be alarmed. Sometimes our letters get crossed in the mail.)

VERSION LIST

August 9, 1982

The listed software is available from the authors, computer stores distributors, and publishers. Except in the cases noted, all software requires CP/M-80, SB-80, or compatible operating systems.

S	Standard Version
P	Processor. Products marked Z80 work on Z80 only; those designated 8080 work on Z80 also.
MR	Memory Required. This, where applicable, refers to the size of the CP/M-80 required. Starred (*) entries refer to the Transient Program Area required; it varies with the RAM available in the computer.

New Products and **new versions** are listed in boldface.

Product	S	P	MR	
ACCESS-80	1.0	8080	54K	
Accounts Payable/Cybernetics				Needs RM/COBOL. Runs w/CP/M-80, OASIS, UNIX
Accounts Payable/MC	1.0	8080	56K	For CP/M-80 2.2
Accounts Payable/Structured Sys	1.3B	8080	45K*	w/It Works run time pkg.
Accounts Payable/Osborne/McGraw-Hill	2.1	8080	48K	Needs CBASIC2
Accounts Payable/Peachtree	07-13-80		48K	Needs BASIC-80 4.51
Accounting Plus		8080	64K	
Accounts Receivable/Cybernetics				Needs RM/COBOL. Runs w/CP/M-80, OASIS, UNIX
Accounts Receivable/MC	1.0	8080	56K	CP/M-80 2.2
Accounts Receivable/Osborne/McGraw-Hill	2.1	8080	48K	Needs CBASIC2
Accounts Receivable/Peachtree	07-13-80	8080	48K	Needs BASIC-80 4.51
Accounts Receivable/Structured Sys	1.4C	8080	49K*	w/It Works run time pkg.
Address Management System	1.0	8080		Requires 2 drives
ALGOL 60	4.8C	8080	24K	Not for rebuilt Osborne
ANALYST	2.0	8080	52K	Needs CBASIC2,QSORT/ULTRASORT
APL/V80	3.2	Z80	27K*	Needs APL terminal
Apartment Management (Cornwall)	1.0	Z80	48K*	Needs CBASIC2
ASCOM	2.02	8080		
ASCOM/86	2.01	8086		Specify operating system: IBMPC/CPM-86/MS-DOS
ASM/XITAN	3.11	Z80		
Automated Patient History	1.2	8080	48K	
BASIC Compiler	**5.30a**	**8080**	**48K**	
BASIC-80 Interpreter	5.21	8080	36K*	w/Vers. 4.51,5.21
BASIC Utility Disk	**2.0a**	**8080**	**20K***	
BaZic II	03/03	Z80	32K	
Benchmark Word Processor	2.2		40K*	Give Name & Model #'s of the video terminal
Benchmark Mail List	1.1		40K*	Give Name & Model #'s of the video terminal
BOSS Financial Accounting System	1.08	8080	48K	Needs 2/3- drives w/min 200k each, & 132-col. printer
BOSS Demo	1.08	8080	44K*	
BSTAM Communication System	**4.6**	**8080**	**16K***	
BDS C Compiler	1.46c	8080	46K*	w/'C' book
Whitesmiths' C Compiler	2.1	8080	56K*	
BSTMS	1.2	8080	24K	
BUG/uBUG Debuggers	**3.3**	**Z80**	**24K***	
C/80 C Compiler	**2.0**			**For HDOS also**
CBASIC2 Compiler	2.08	8080	24K*	w/CRUN(2,204P, & 238)
CBS Applications Builder	1.33	8080	48K	Needs no support language
CIS COBOL Compiler	4.4,1	8080	43K*	
CIS COBOL Compact	3.46	8080	26K*	
FORMS 1 CIS COBOL Form Generator	1.06	8080		
FORMS 2 CIS COBOL Form Generator	1.1,6a	8080	43K*	
Interface for Mits Q70 Printer				CP/M-80 1.41 or 2.XX
COBOL-80 Compiler	4.6	8080	48K	
COBOL-80 PLUS M/SORT	4.6	8080	48K	
CONDOR II	2.06	8080	42K*	
CREAM (Real Estate Acct'ng)	2.3	8080	64K	CBASIC needed
Crosstalk	1.4	Z80		
DATASTAR Information Manager	1.101	8080	34K*	
Datebook-II	2.04	8080	52K	Needs 80x24 terminal, N/A for CDOS, CP/M-80 1.4, MP/M-80
dBASE-II	2.3B	8080	42K*	Includes "Zip"
dBASE-II Demo	2.3B	8080	42K*	
Dental Management System 8000	8.7A	8080	48K	Needs CBASIC
Dental Management System 9000 & Demo	**2.05**	**8080**	**48K**	**Needs CBASIC**
DESPOOL Print Spooler	2.1A	8080	19K	
DISILOG Z80 Disassembler	4.0	Z80		Zilog mnemonics
DISTEL Z80/8080 Disassembler	4.0	8080		Intel mnemonics,TDL extensions
Documate/Plus	1.4	8080	36K*	
Documate/Plus/Demo	1.5	8080	36K*	
EDIT Text Editor	2.06	Z80	24K*	
EDIT-80 Text Editor	2.02	8080	20K*	
EM 80/86	1.00	8086		Specify operating system: IBM PC/CPM-86/MS-DOS
Emulator-86	1.0	8086		Runs CP/M-86 programs on MS-DOS, PC DOS
FABS-I	2.7	8080	32K	
FABS II	4.15	8080	48K	
FILETRAN	1.20		32K	1-way TRS-80 Mod I,TRSDOS to Mod I CP/M-80
FILETRAN	1.4		32K	Needs TRSDOS. 2-way TRS-80 Mod I,TRSDOS & Mod I CP/M-80
FILETRAN	1.5		32K	1-way TRS-80 Mod II,TRSDOS to Mod II CP/M-80
FinalWord	1.0	8080	56K	Runs under CP/M-80, CP/M-86 or IBM PC DOS
Financial Modeling System	2.0		48K	
Formula w/General Accounting System	1.01			
FORTH (Timin)	3.5	8080	28K	
FORTRAN-80 Compiler	3.44	8080	32K*	
FPL 56K Vers.	2.6	8080	56K	
FPL 48K Vers.	2.6	8080	48K	
General Ledger 9000 & Demo	1.06			Needs CBASIC
General Ledger/Cybernetics				Needs RM/COBOL. Runs w/CP/M-80, OASIS, UNIX
General Ledger/MC	1.0	8080	56K	Needs CP/M-80 2.2 or MP/M-80

VERSION LIST

Product	S	P	MR	
General Ledger/Osborne/McGraw-Hill	2.4	8080	48K	Needs CBASIC2
General Ledger/Peachtree	07-13-80	8080	48K	Needs BASIC-80 4.51
General Ledger/Structured Sys	1.4C	8080	45K*	w/It Works Package
GLECTOR Accounting System	2.02	8080	48K*	Use w/CBASIC2, SELECTOR III
GLECTOR IV Accounting System	1.0	8080		Needs SELECTOR IV
GrafTalk	1.0		48K*	Requires 180Kb/drive. Available for certain Terminals Printers, & Plotters. N/A CDOS.
HDBS	1.05A	+	52K	
High Yield		**8080**	**38K**	
HOE	2.1	8080	48K	
IBM/CPM	1.1	8080		CP/M 1.4 only
Insurance Agency System 9000 & Demo	1.10	8080		Needs CBASIC
Integrated Acctg Sys/Gen'l Ledger		8080	48K	Needed for 3 pkgs. below
Integrated Acctg Sys/Accts Pyble		8080	48K	
Integrated Acctg Sys/Accts Rcvble		8080	48K	
Integrated Acctg Sys/Payroll		8080	48K	
Interchange		Z80	32K	
Inventory/MicroConsultants	5.3	8080	56K	Needs CP/M-80 2.2
Inventory/Peachtree	07-13-80	8080	48K	Needs BASIC-80 4.51
Inventory/Structured Sys	1.0C	8080	48K*	w/It Works Package
ITOZ/ZTOI	**1.0**	**Z80**		**For 8″ IBM, Micropolis, N'Star, Apple**
JANUS	1.4.3	8080		Also runs w/8086
Job Cost Control System/MC	1.0	8080	56K	Requires CP/M-80 2.2
JRT Pascal System	1.4	8080	50K*	
Legal Time Acctg Series 9000 & Demo	1.07	8080		Needs CBASIC
LETTERIGHT Text Editor	1.1B	8080	48K*	
LINKER		Z80		
LP-DISK	1.0	8080	48K	Also for TRS-80 I/III
MAC	2.0A	8080	20K*	
MACRO-80 Macro Assembler Package	3.43	8080		
MAG/base1 (LMS)	3.0	8080	56K	Needs CBASIC, 2.06 or later & 180K/drive
MAG/base2 (IMS)	3.0	8080	56K	Needs CBASIC, 2.06 or later & 180K/drive
MAG/base3 (ADS)	3.0	8080	56K	Needs CBASIC, 2.06 or later & 180K/drive
Magic Typewriter	3	Z80	48K	
Magic Wand	1.11	8080	28K*	
MAG/sam-E3	**3.0**	**8080**	**32K**	**Specify: CBASIC or BASIC-80**
MAGSORT-C	1.0		12K*	For CBASIC
MAGSORT-M	1.0		12K*	For MBASIC
MAGSORT-R	1.0		12K*	For Compilers — BASCOM, FORTRAN-80, PL/I-80
MAILING ADDRESS Mail List System	07-13-80	8080	48K	
MailMerge	3.0	8080	44K*	
Market Time		**8080**	**34K**	
Master Tax	1.0-80	8080	48K	
Matchmaker		8080	32K	
Math★	3.043	8080		Math add-on to WordStar
MDBS	1.05A	+	44K*	
MDBS-DRS	1.02	+	48K*	
MDBS-QRS	1.0	+	52K	
MDBS-RTL	1.0	+	52K	
Medical Management System 8000	8.7a	8080		Needs CBASIC
Medical Management System 9000	**2.05**	**8080**		**Needs CBASIC**
Medical Management System 9000 Demo	2.03	8080		Needs CBASIC
Microcosm		Z80		CP/M-80 2.X or MP/M-80
MicroSEED	B.10G	8080	48K	
Microspell	**4.3**	**8080/8086**	**48K**	**For MS-DOS also**
Microspell Demo	1.0	8080		For Dealers Only
Microstat	**2.09**	**8080**	**48K**	**Needs baZic 03/03 or BASIC-80, 5.03 or later**
Microstat for Apple	2.0	Z80	48K*	
Mince	2.6	8080	56K	
Mince Demo	2.6	8080	48K	
Mini-Warehouse Mngmt. Sys.	5.5	8080	48K	Needs CBASIC
Money Maestro		8080	48K	CP/M-80 1.4 or 2.2
MP/M-I	1.1			
MP/M-II	2.0	8080	48K	32K RAM needed
Mr. EDit	2.0	8080	24K	Needs 24K TPA, 12 x 64 column terminal
MSORT	1.01	8080	48K	
MuLISP-80/MuSTAR Compiler	2.12	8080	24K	
MuSIMP / MuMATH Package	2.12	8080	48K	muMATH-80
NAD Mail List System	3.0D	8080	48K	
Nevada COBOL	2.1	8080	32K	
Order Entry w/Inventory/Cybernetics		Z80		Needs RM/COBOL
Panel	**3.03**		**44K**	
PAS-3 Medical	1.79	8080	56K	Needs 132-col. printer & CBASIC
PAS-3 Dental	1.66	8080	56K	Needs 132-col. printer & CBASIC
PASM Assembler	1.02	Z80		
Pascal/M	4.02	8080	56K	CP/M 2.x only
PASCAL/MT Compiler	3.2	8080	32K	
PASCAL/MT+ w/SPP	5.5	8080	52K	Needs 165K/drive
PASCAL/Z Compiler	4.0	Z80	56K	
Payroll w/Cost Acct./Osborne/McGraw-Hill	2.2	8080	48K	Needs CBASIC2
Payroll/Peachtree	07-13-81	8080	48K	Needs BASIC-80 4.51
Payroll/Structured Sys	1.0E	8080	60K	w/It Works run time pkg.
PEARL SD	3.01	8080	56K	w/CBASIC2, ULTRASORT II
PLAN80 Financial Package (Z80/8080)	2.3	8080	56K	Specify Z80/8080
PLAN80 Demo	1.2			
PL/I-80	1.3	8080	48K	
PLINK I Linking Loader	3.28	Z80	24K*	

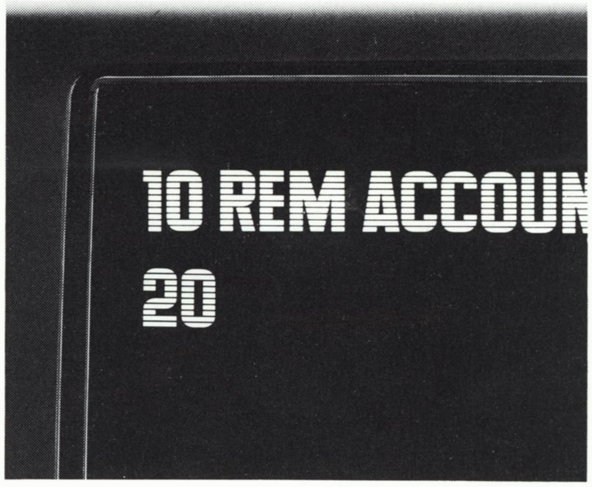
(continued next page)

VERSION LIST

Product	S	P	MR	
PLINK-II Linking Loader	1.14	Z80	24K*	
PMATE	3.02	8080	24K*	
PMATE-PC	1.05	8086		For the IBM PC
POSTMASTER Mail List System	3.5	8080	48K	
Professional Time Acctg	3.11a	8080	48K	Needs CBASIC2
Programmer's Apprentice	1.2	8080	56K	Needs BASCOM 5.3, 2-251 Kb/drive
Property Management Program (AMC)	4.2	Z80	48K	Needs CBASIC 2.07+, CP/M-80 2.0+
Property Management System	07-13-80	8080		Needs BASIC-80 4.51
PSORT	1.4	8080		N/A-Durango
QSORT Sort Program	2.0	8080	48K	
Quic-N-Easi	1.4	Z80	48K	Also runs on TRS-80 Mod III
Real Estate Acquisition Programs	2.1	8080	56K	Needs CBASIC
Remote	3.01	Z80		
Residential Prop. Mngemt. Sys.	1.0	Z80	48K	
RAID	5.0.2	8080	28K	
RAID w/FPP	5.0.2	8080		
RECLAIM Disk Verification Program	2.1	8080		
Sales Pro	5.0	8080		
SBASIC	5.4a	8080		
Scribble	1.3	8080		
SELECTOR-III-C2 Data Manager	3.24	8080	48K	Needs CBASIC
SELECTOR-IV	2.17	8080	52K	Needs CBASIC
SELECTOR-V	5.0	8080	48K	
Shortax	1.2	Z80	48K	TRSDOS,MDOS too, needs BASIC-80 5.0
SID Symbolic Debugger	1.4	8080		N/A-Superbr'n
Spellguard	2.0	8080	48K	Needs Word Processing Program
STATPAK	2.1	8080		Needs BASIC-80 4.2 or above
STIFF UPPER LISP	2.8	8080		
STRING BIT FORTRAN Routines	1.02	8080		
STRING/80 bit FORTRAN Routines	1.22	8080		
STRING/80 bit Source	1.22	8080		
SUPERSORT I Sort Package	1.6	8080		Max. record=4096 bytes
SELECT		8080	40K	
T/MAKER II	2.6	8080	48K	Avail. for CDOS
T/MAKER II DEMO	2.4	8080	48K	
TEX Text Formatter	2.1	8080	36K	
TEXTWRITER-III	3.6	8080	32K	
TIM-III	3.12	8080	32K	Needs 2 240 Kb/drives
TIM-III	3.11	8086		For the IBM PC
TINY C Interpreter	800102C	8080		
TINY C-II Compiler	800201	8080		
Torricelli Author	**1.04d**	**Z80/8085**	**48K**	**24x80 CRT, 2-100Kb/drive**
TRS-80 Customization Disk	1.3C	8080		
ULTRASORT II	4.1C	8080	48K	
UT-86	1.00	8086		Specify operating system: IBM PC/CPM-86/MS-DOS
Lifeboat Unlock	1.3	8080		Use w/BASIC-80 5.2
VISAM	2.3p	8080	40K*	
Wiremaster	**4.03**	**Z80**	**44K**	**Needs 180K/drive**
WordIndex	3.0	8080	48K	Needs WordStar
WordMaster	1.07A	8080	40K	
WordStar	3.0	8080	48K*	
WordStar French	**2.26**			
WordStar IBM PC	3.02M	8086		
WordStar Customization Notes	3.0	8080		
XASM-05 Cross Assembler	1.05	8080	24K	
XASM-09 Cross Assembler	1.07	8080	24K	
XASM-51 Cross Assembler	1.09	8080	24K	
XASM-75	**1.0**	**8080**	**24K**	
XASM-F8 Cross Assembler	1.04	8080	24K	
XASM-400 Cross Assembler	1.03	8080	24K	
XASM-18 Cross Assembler	1.41	8080	24K	
XASM-48 Cross Assembler	1.62	8080	24K	
XASM-65 Cross Assembler	1.97	8080	24K	
XASM-68 Cross Assembler	2.00	8080	24K	
XASM-Z8	**1.0**	**8080**	**24K**	
XYBASIC Extended Interpreter	2.11	8080		
XYBASIC Extended Disk Interpreter	2.11	8080		With EDIT features
XYBASIC Extended Compiler	2.0	8080		Requires the XYBASIC w/EDIT features to create SOURCE
XYBASIC Extended Romable	2.1	8080		
XYBASIC Integer Interpreter	1.7	8080		
XYBASIC Integer Compiler	2.0	8080		
XYBASIC Integer Romable	1.7	8080		
ZAP-80	1.4	8080	24K	Needs 50K/drive
Z80 Development Package	3.5	Z80		N/A-Magnolia,Superbr'n,mod.CP/M-80
ZDM/ZDMZ Debugger	**1.4/2.3**	**Z80**		**For N'Star,Apple,IBM 8", Micropolis Mod II**
ZDT Z80 Debugger	1.41	Z80		N/A-Superbr'n,mod.CP/M-80
ZSID Z80 Debugger	1.4A	Z80		N/A-Superbr'n,mod.CP/M-80

+These products are available in Z80 or 8080, in the following host language: BASCOM, COBOL-80, FORTRAN-80, PASCAL/M, PASCAL/Z, CIS-COBOL, CBASIC, PL/I-80, BASIC-80 4.51, and BASIC-80 5.xx.

BOY, IS THIS COSTING YOU.



It's really quite basic: time is money.

And BASIC takes a lot more time and costs a lot more money than it should every time you write a new business software package.

Especially when you could speed things up with dBASE II.

dBASE II is a complete applications development package.

Users tell us they've cut the amount of code they write by up to 80% with dBASE II.

Because dBASE II is the high performance relational database management system for micros.

Database and file handling operations are done automatically, so you don't get involved with sets, lists, pointers, or even opening and closing of files.

Instead, you write your code in concepts.

And solve your customers' problems faster and for a lot less than with BASIC (or FORTRAN, COBOL or PL/I).

dBASE II uses English-like commands.

dBASE II uses a structured language to put you in full control of your data handling operations.

It has screen handling facilities for setting up input and output forms.

It has a built-in query facility, including multi-key and sub-field searches, so you can DISPLAY some or all of the data for any conditions you want to apply.

You can UPDATE, MODIFY and REPLACE entire databases or individual characters.

CREATE new databases in minutes, or JOIN databases that already exist.

APPEND new data almost instantly, whether the file has 10 records or tens of thousands.

SORT the data on as many keys as you want. Or INDEX it instead, then FIND whatever you're looking for in seconds, even using floppies.

Organize months worth of data in minutes with the built-in REPORT. Or control every row and column on your CRT and your printer, to format input and output exactly the way you want it.

You can do automatic calculations on fields, records and entire databases with a few keystrokes, with accuracy to 10 places.

Change your data or your entire database structure without re-entering all your data.

And after you're finished, you can protect all that elegant code with our run-time compiler.

Expand your clientbase with dBASE II.

With dBASE II, you'll write programs a lot faster and a lot more efficiently. You'll be able to write more programs for more clients. Even take on the smaller jobs that were out of the economic question before. Those nice little foot-in-the-database assignments that grow into bigger and better bottom lines.

Your competitors know of this offer.

The price of dBASE II is $700 but you can try it free for 30 days.

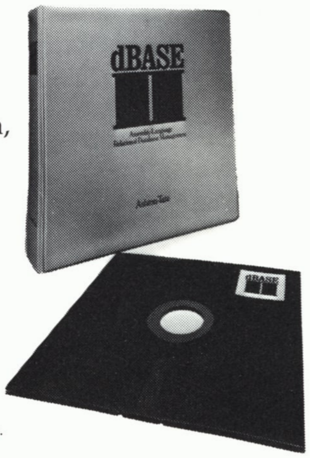
Call for our Dealer Plan and OEM run-time package prices, then take us up on our money-back guarantee. Send us your check and we'll send you a copy of dBASE II that you can exercise on your CP/M® system any way you want for 30 days.

Then send dBASE II back and we'll return all of your money, no questions asked.

During that 30 days, you can find out exactly how much dBASE II can save you, and how much more it lets you do.

But it's only fair to warn you: business programmers don't go back to BASIC's.

Ashton-Tate, 9929 Jefferson, Los Angeles, CA 90230. (213) 204-5570.



Ashton-Tate

Also available from Lifeboat Associates.

LIFELINES™/The Software Magazine™

1651 Third Avenue, New York, New York 10028

Second Class Postage Paid
At New York, N.Y.