LIFELINES
The Software Magazine™

$2.50 April 1982 Volume II, No. 11 (ISSN 0279-2575, USPS 597-830)

The Future Of UNIX

Full Screen Program Editors: MINCE

A Lisp Computing Environment

What About SpellStar?

Bit Manipulation In PL/I-80

The Ultimate Printer?

XLT86, A Review And How To Minimize 8080 to 8086 Translation Grief

OKIDATA 100% DUTY CYCLE PRINTERS

SETTING NEW STANDARDS IN QUALITY
APART FROM THE REST

MODEL	ML80	ML82A	ML83A	ML84	2350
Columns:	80	80	136	136	136
Print Speed: (cps)	80	120	120	200	350
Bidirectional/Short Line Seeking:	—	✓	✓	✓	✓
Throughput: (lpm)					
20 Char/line	86	187	173	266	500
40 Char/line	51	123	117	184	340
80 Char/line	28	73	71	114	210
136 Char/line	—	—	46	74	136
Head Warranty	— 200 million characters —				500 million
Graphics Option:	Block	60x66	60x66	72x72	72x72
RS 232:	Opt.	Std.	Std.	Opt.	Opt.
Tractor Feed:	Opt.	Opt.	Std.	Std.	Std.
Friction Feed:	✓	✓	✓	✓	—
Pin Feed:	✓	✓	—	—	—
Super Scripts • Sub-Scripts • Underline:	—	—	—	—	✓
Colors:	—	—	—	—	2

Immediate Delivery • Technical Assistance • Leasing • Maintenance • Interface Cables • Ribbons

DISTRIBUTED by: **GRAYDON-SHERMAN, INC.**

(212) 289-3199 (201) 467-1401 * TWX #710-983-4375 (GRAYDON MAWD)

LIFELINES The Software Magazine

April 1982

Volume II, No. 11

Editor-in-Chief: Edward H. Currie
Managing Editor: Jane Mellin
Administrative Assistant: Patricia Matthews
Production Assistant: K. Gartner
Typographer: Harold Black
Cover by K. Gartner

Lifelines (ISSN 0279-2575, USPS 597-830) is published monthly at a subscription price of $18 for twelve issues, when destined for the U.S., Canada, or Mexico, $40 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.

LIFELINES The Software Magazine

April 1982

Volume II, No. 11

Contents

Opinion Editorial Comments

Edward H. Currie

Operating System Standards Emerge

This month Lifeboat Associates and Microsoft participated in MS™DOS/SB-86™ symposia in Santa Clara California and New York City. Both symposia were a resounding success and made it very clear that MSDOS will be *the* operating system for the 8088/8086 world. The all-day presentation was divided into two sessions. The morning session was a general discussion of the eight- and sixteen-bit operating systems for the single user, single tasking environment, followed by a technical sessions in the afternoon on the various aspects of SB-86/MSDOS. Particularly interesting was the discussion of the role that XENIX™, Microsoft's version of UNIX, was to play as their offering for the multi-user, multi-tasking environment.

The attendees were left with an appreciation for Microsoft's commitment to operating systems and for the technical and commercial expertise which went into MSDOS and XENIX. Particularly gratifying was the reception by the press and their enthusiastic interest in the discussion of standards. Watch the trade publications for more and more discussions of a fully sanctioned, industry standard operating system for the 8088/8086 world.

In the months to come we will reprint some of the presentations and review the others in editorials.

Intel's entry into the operating system arena is an exciting product called iRMX-86™. This sophisticated real time operating system is designed to provide the system designer with reduced application development costs, improved portability of applications, optimal utilization of hardware, and support of sound development methodology. iRMX-86 is intended to support multi-user (actually multi-programming), multi-tasking, interrupt mapping, timer support, memory allocation, device support, file support, loading support and a human interface.

Often in real time environments it is necessary to have several programs running concurrently. Each such program is called a task and this situation is referred to as a multi-tasking environment. Note that such programs may actually only appear to be running concurrently, as in the case of a single processor environment where events are occurring slowly enough to allow tasks to be effectively running simultaneously. In other environments there may be multiple processors each running one or more tasks, but each is under the purview of a master processor.

Since most real time environments are event-driven there must be an interrupt structure to manage the various tasks. Tasks are typically assigned priorities as to relative importance in terms of system response. Thus a task of lower priority will be "interrupted" when a higher priority task must be serviced. The system then returns to servicing the original task until such time as the task is completed or another higher priority request occurs.

Memory allocation is an important aspect of real time systems in which concurrent tasks are supported. Obviously each task will have specific memory requirements, at least in terms of minimal memory allocation to the task. This approach is vastly superior to fixed partitioning of memory.

Device support refers to such things as device independent I/O, in which the application does not have to concern itself with the specific nature of an I/O device but rather acts as if all I/O devices are accessed the same way. This greatly simplifies the tasks of the application program; it is in fact interacting with a wide variety of I/O devices but is able to communicate with all of them in the same manner.

File support refers to the management of a secondary mass storage device. Such management is concerned with dynamic as opposed to fixed or static allocation of random access space, locations of each file, file naming, etc. Thus the operating system allows the application to treat a single random access device as if it were many separate devices, each of which may be randomly accessed.

Loading support refers to the loading of applications from mass storage devices, as opposed to programs which reside in read-only memory.

The so-called human interface provides for common editing functions and invocation of application programs; it offers functions which permit the application program to determine which parameters have been specified by the operator.

iRMX consists of a nucleus which provides intertask communication, multitasking, timer support and interrupt control. The operating system interfaces "encapsulate" the specific details of a given implementation by implementing each mechanism as a type manager. Such type managers provide a set of objects that are defined by their attributes and the operations performed upon them. The basic object types are defined as task, segment, mailbox, semaphore, region, job and extension. Interestingly enough, tasks are also a subject, since tasks are active elements of the system and perform operations upon objects.

You are probably asking yourself: why the interest in YAOP (i.e. Yet Another Operating System)? Each of the operating systems we have discussed in recent months has an important role to play, and they are all fine systems. In the case of single user, single tasking systems where available memory size is a significant issue, clearly MSDOS is *the* operating system. For development environments, multi-user/multi-tasking applications and users interested in a UNIX type environment, XENIX is the best choice. Finally for real time, perhaps stand-alone applications, such as ROM-based or industrial control systems iRMX86, is an excellent choice.

Lifeboat Associates is jointly sponsoring the iRMX Users Group with Intel. This group will gather iRMX public domain software for 8088/8086 systems, publish a newsletter, distribute diskettes containing the programs selected for distribution from those contributed and in general provide a library of programs and services to stimulate exchange of iRMX software.

ON UNIX
The Future of UNIX

Jean L. Yates

The UNIX operating system from Bell Laboratories is at the heart of a dynamic market that is changing rapidly. Presently, ATT accepts about one VAX or PDP-11 per week and many of these computers run UNIX either on or along with VMS, the DEC operating system. It is estimated that about five thousand PDP-11 and VAX computers run UNIX today and the number continues to expand, although the largest increases will no longer be in DEC hardware but in microprocessor driven hardware as we move into the 1982-1983 timeframe. Today there are over one hundred UNIX based micro and minicomputers under development, and although only a few of them have been released, we will see a deluge of products in the next two years. The major manufacturers, along with small companies, are devising microprocesser driven UNIX based systems.

One key to the pick up in the UNIX market was the introduction by The Bell System of low end user prices. Binary licensing per user is now as little as $40 to a source code licensee for resale. This brings UNIX into the same price range as CP/M.

UNIX based office automation systems are beginning to achieve market share from mostly PDP-11 or VAX based equipment. Microprocessor driven machines such as the Fortune Systems single user workstation will also achieve significant marketshare, particularly networked together with networking software and hardware, such as the 3COM product for netting UNIX into Ethernet.

Major corporations are also accepting UNIX as an internal standard. Companies such as Hughes Aircraft, Ford Aerospace, and TRW utilize UNIX as a standard for their standard operating system for their DP shops.

Creating a de facto Standard

How did UNIX become so popular and widespread? To begin with, a key factor was the donation of UNIX to universities by Bell Labs. For about $200, universities could purchase source licenses for educational purposes. This resulted in a widespread utilization of UNIX as the training tool of choice in computer science departments. Today, almost ninety percent of the computer science departments in universities utilize UNIX in some form or another.

Students and computer science professors proved UNIX's portability as a concept. They had a vested interest in maintaining the portability of their research data and software, and much of the development of the UNIX portability features was achieved under joint agreements with the Bell System and universities. Berkeley, in particular, has produced an outstanding port of UNIX to the VAX computer and universities such as MIT and Stanford have produced microprocessor ports. In MIT and Stanford's case, the ports were to the 68000 microprocessor.

UNIX has been available to universities for about six years, and the result is that the students of these computer sicence departments are now in industrial organizations across the country, and they know and like UNIX as a tool. More and more students knowledgeable in C and UNIX are entering the labor force.

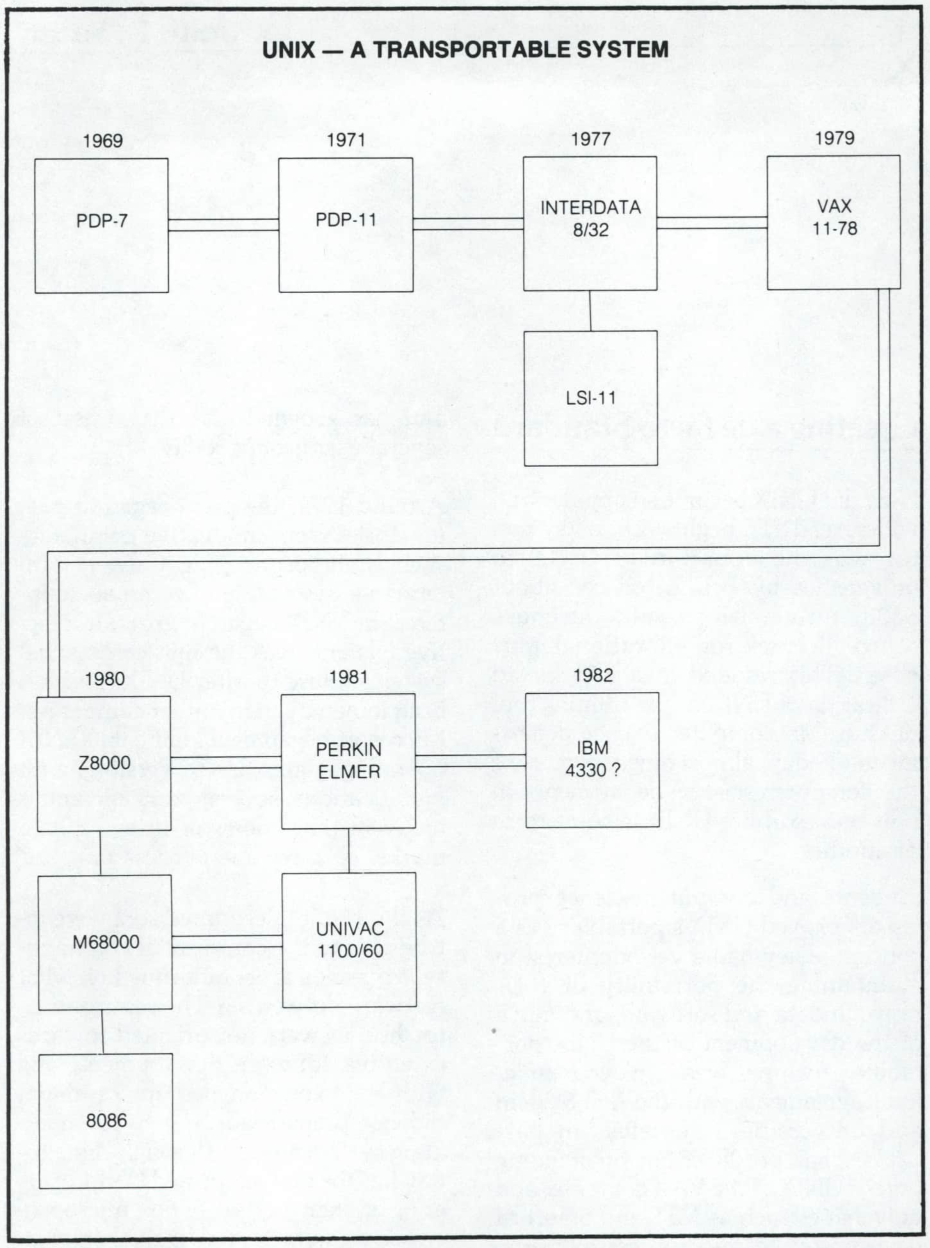
Users groups began in the academic world with a 1974/1975 meeting called by Dr. Fabry, a senior professor at Berkeley. Dr. Fabry today is in charge of Berkeley's BSD 4.1 VAX UNIX and continues to train students in the use and theory of operating systems. The USENIX user group started in 1975, and has grown to be the largest of several user groups today.

Around 1979, the press began to pick up stories from innovative companies such as Microsoft, Zilog, Onyx and Interactive Systems, and commerical interest in UNIX began to grow. Interactive Systems was the first commercial systems house to offer UNIX, and has been joined by many other companies. Microsoft began developing its XENIX operating system, UNIX Version 7 with modifications, several years ago and is today in the process of bringing it to market on three microprocessors.

As the 16-bit microprocessors were introduced to the commercial community, a question arose quite quickly: what software do we use? The semiconductor houses were not oriented towards extensive software development, and with the exception perhaps of Zilog, did not prepare adequately for operating system needs. Although Zilog today has the best supported UNIX operating system of the major microprocessor manufacturers, the other companies are coming up quickly behind Zilog as they recognize the demand for the UNIX operating system. Although Intel is attempting to push its own operating systems, lack of applications software and relative simplicity in comparison to UNIX may limit its desirability to end users. Business people today see comparisons of $2-5 million for developing their own operating system versus buying UNIX source and modifying it to their needs. Also, by maintaining compatibility with UNIX System III at an applications software level, they know that there will be many different applications packages available for their equipment.

The last two years have seen a slow speed-up in the demand for ports. Cross compilers and other materials

(continued next page)



needed to port UNIX have taken time to develop. Successful downloading of minicomputer developed applications programs is now beginning to occur behind successful ports of UNIX onto microprocessor driven systems. This trend of porting minicomputer developed applications software is one point for UNIX winning out over CP/M in the 16-bit world. Multiuser, multitasking applications packages designed on minicomputers for medium-sized corporations, in languages including COBOL and FORTRAN, are being downloaded to microprocessor driven systems. This software is tested, debugged, and has been, in some cases, installed in minicomputer installations for several years. Although the pricing must be lowered to impact the microcomputer industry, this software offers such significant advantages over microcomputer developed products that it, if enough is downloaded, will prevail over any number of CP/M developed applications—even if those applications are moved to greater complexity by the developers. Within the UNIX operating system itself, several applications programs require very little clean up to become word processors, desktop calculators, spread sheets, and other applications.

UNIX— A Transportable System

I will now go over the history of UNIX briefly. UNIX was developed in the assembly language of the PDP-7 in 1969 by Ken Thompson, and later rewritten into the language C, developed by Dennis Ritchie, and moved onto the PDP-11. Starting with Version 5, the first version released to universities, UNIX became more transportable; a number of projects in the following years moved it onto different machines. Practically the entire series of PDP-11 computers have UNIX on them and in 1977, LSI and Interdata 8/32 machines were ported. In 1979, Berkeley ported to the VAX and another research institution ported UNIX onto the Prime computer.

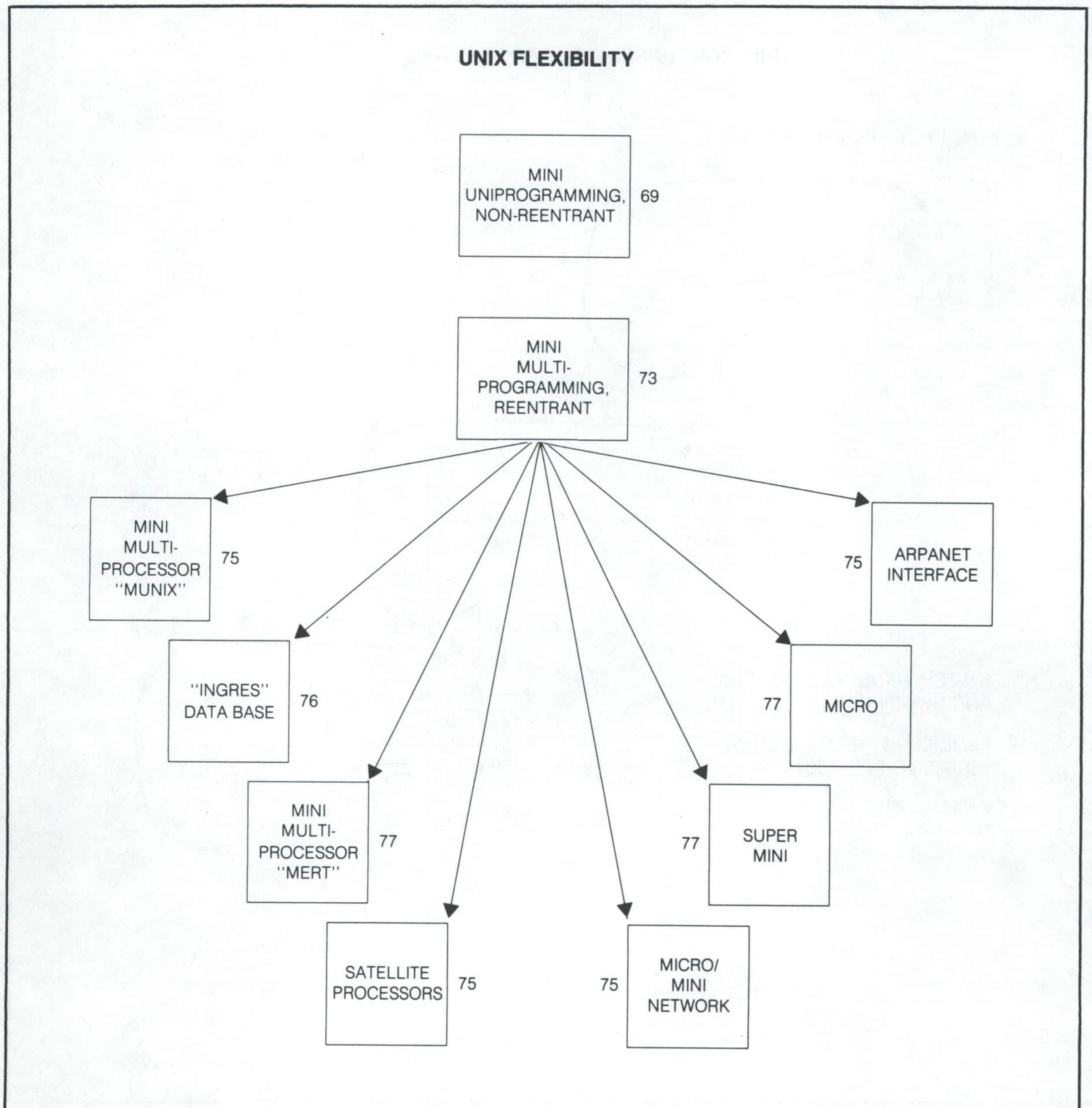
1980 saw the beginning development of porting tools to the microprocessors, and the introduuction of the UNIVAX 1100/60. In 1981, the Perkin-Elmer 32--bit super minicomputer with UNIX was greeted with enthusiasm by end users and the Amdahl 470 with UNIX was introduced with unanticipated levels of success. 1982 will see the introduction of IBM's Series One with UNIX and at least 40 microprocessor driven computers using the UNIX operating system.

UNIX Flexibility

The flexibility of UNIX is seen in the foil presented here. In 1969 it was a non-reentrant non-transportable product, but by 1973 it had become a multiprogramming reentrant product with fairly high levels of portability. 1975 saw multiprocessor UNIX called MUNIX and the Arpanet interface. 1976 saw the INGRIS data base at Berkeley which still continues and 1977 saw the first ports to microprocessors as well as to the superminis. MERT, a real time version of UNIX, was developed at Bell Labs around 1977. Also in the 1977 timeframe were the initial research oriented developments of micro and mini networks and satellite processors. This charted course in the foil could be elaborated on to include enhancements such as the UCLA security kernel, and a wide variety of board level ports and application packages developed for the UNIX system.

Major UNIX Milestones

As you can see from the foil, UNIX being written into C was our first milestone, followed by its transport to a variety of CPUs. From a commercial standpoint, the first UNIX system house introduction was quite significant. 1980 saw the microprocessor use of UNIX interest exploding, etc. 1981 introduced System III with significant price reductions and the beginnings of increasing developments of commercially oriented applications packages. 1982 will see the IBM Series One, lending increasing credibility to UNIX as a commercial product. Perhaps support from DEC for UNIX, and an applications package that will sell UNIX, Writers Workbench, will also have major impact. This unique package is not a word processor, but rather is an automated style guide that checks grammar, syntax, spelling and consistency within documents to increase the speed and accuracy with which the written material can be prepared as business or technically oriented documents. Also in 1982 we will see the introduction of mini and microcomputers from major manufacturers with the UNIX operating system, along with at least several 68000 based smart terminals using UNIX, with upgrade capabilities.



UNIX FLEXIBILITY
MINI UNIPROGRAMMING, NON-REENTRANT 69
MINI MULTI-PROGRAMMING, REENTRANT 73
MINI MULTI-PROCESSOR "MUNIX" 75
ARPANET INTERFACE 75
"INGRES" DATA BASE 76
MICRO 77
MINI MULTI-PROCESSOR "MERT" 77
SUPER MINI 77
SATELLITE PROCESSORS 75
MICRO/ MINI NETWORK 75

MAJOR UNIX MILESTONES

1969
- First UNIX (Assembly Language)

1973
- UNIX Written in C

1977
- 16-32 Bit Transportability Project
- First UNIX System House

1980
- Transported to Various Micros
- UNIX Interest Explodes
- UNIX Touted as OS Standard

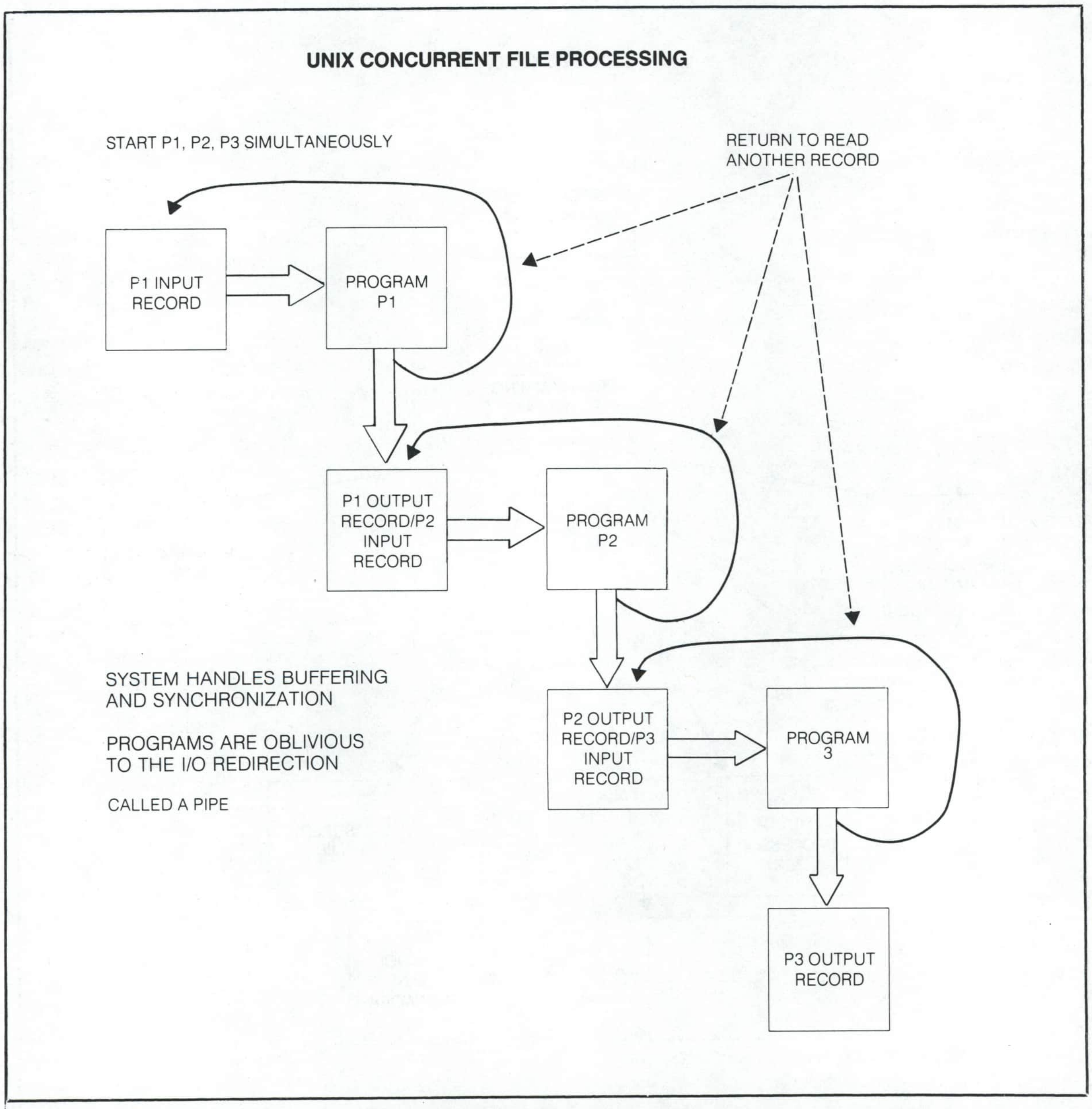
1981
- IBM 4430 Proposal

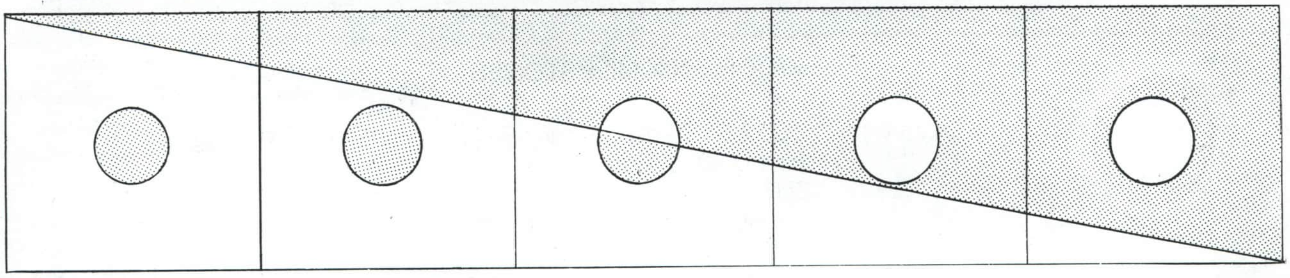
(continued next page)

UNIX Concurrent File Processing

For a moment, let's briefly take a look at what for many of you must be a very simple concept but for others may be new. As you can see from processes 1, 2, and 3 in the drawing, this system handles buffering and synchronization. The programs are oblivious to the directions of the input or output. This is called a pipe and is one of the basic concepts of UNIX that makes it so powerful. For an end user, it simply results in the ability to send a file through several programs and gather results that would otherwise take many more steps to achieve.

Although pipes were designed in a programming environment, for applications they offer significant advantages. For example, a systems house or OEM devising added value for their product can create a series of pipes and filters, systems calls, and related UNIX utilities to perform a series of common office tasks requiring file manipulation and management. Creation of these pipes is much faster than writing programs. In this sense, UNIX commands act as sort of a quasi-macro language.


UNIX CONCURRENT FILE PROCESSING
START P1, P2, P3 SIMULTANEOUSLY
RETURN TO READ ANOTHER RECORD
P1 INPUT RECORD
PROGRAM P1
P1 OUTPUT RECORD/P2 INPUT RECORD
PROGRAM P2
P2 OUTPUT RECORD/P3 INPUT RECORD
PROGRAM 3
P3 OUTPUT RECORD
SYSTEM HANDLES BUFFERING AND SYNCHRONIZATION
PROGRAMS ARE OBLIVIOUS TO THE I/O REDIRECTION
CALLED A PIPE



Full Screen Program Editors: MINCE

Ward Christensen

MINCE (from Mark of the Unicorn) is the editor that got me thinking about doing this series of editor reviews. Its promise of multi-file edits and split screen was irresistible.

The MINCE promise is true. It has proven to be enjoyable, sometimes amazing, and just plain fun.

MINCE, which stands for "MINCE Is Not Complete EMACS", is based upon the EMACS editor from MIT. It quite faithfully mimics the functions of EMACS.

I enjoyed my first experience with MINCE immensely. I installed it, then spent the next five hours trying every command. The neighbors must have thought I was nuts, laughing and chuckling with each new discovery.

Evaluation

DOCUMENTATION. The manual is huge - about an inch thick - although printed on only one side. Its style is very friendly and humorous, yet precise and complete. An example of the humor: "Remember, a computer is, among other things, a device for making more mistakes faster than ever before possible."

Documentation consists of:

(1) an installation guide;
(2) eight lessons, including some on disk for you to work with;
(3) Programmer's Introduction to MINCE, for those who are familiar with other editors;
(4) MINCE User's Guide: Explains the basic principles you should understand;
(5) complete command list;
(6) Command Summary; and
(7) a separate one-page command summary.

The one-page command summary is something you will frequently use. I laminated mine in clear contact paper to make it more durable.

The material I have mentioned so far is all that used to be provided. Now it accounts for only about one half of the larger inch-thick manual that Mark of the Unicorn has been shipping since August 1981. This more complete manual includes the following additions:

(1) Program Logic;
(1a) Generalities, (1b) Specifics, (1c) Extending and Modifying MINCE;
(2) Entry points;
(3) Source Code;
(4) The Terminal Abstraction;
(5) Theory and Practice of Text Editors: this section is terrific - let me give you some samples: Memory Management; Buffer Gap; Gap Size; Multiple Gaps and Why They Don't Work; The Hidden Second Gap; Multiple Buffers; Incremental Redisplay; - heavy, interesting reading if you like that kind of detail.

SPEED. MINCE is adequately fast. MINCE earned an early but now untrue reputation for being slow. Written in BDS C, it suffers a bit from the simple fact that the 8080 cannot efficiently handle the stack-relative references heavily used by the C language. For version 2.6, Mark of the Unicorn rewrote the most time-dependent routines in 8080 assembler. That pretty much solved the speed problem.

The speed of MINCE is also affected by the need for MINCE to utilize a virtual memory scheme using a large swapping file. This comes from its powerful generalized nature i.e. multiple buffers, multiple files. They recommend a 64K swap file placed close to the front of the disk so it doesn't have far to seek when accessing the directory.

I highly agree with the author's recommendation that you place the swapping file near the front of the disk. I prefer that it even be on another disk. This will speed up starting and ending the edit by minimizing seek time as the file is being transferred to and from the swap file. MINCE looks for the swapping file first on the logged in disk, then on A:. I keep my swap file on B:, and log in B: before issuing the MINCE command.

ERGONOMICS. MINCE uses a *mnemonic* approach to laying out the keyboard. I call it that because of the dictionary definition of mnemonic: "an aid to memory". Control-B is used for cursor back, control-F for cursor forward, etc.

MINCE has over 80 commands. Don't let this throw you - the most frequently used subsets of these are quite easy to remember. For example, control-N moves the cursor to the (N)ext line. Other similarly easy-to-remember control keys are: (F)orward, (B)ack, (P)revious line, (D)elete character, (K)ill line, etc. In a consistent manner, ESC extends the functions of the control keys. For example, control-F moves the cursor forward one *character* but ESC-F moves it forward one *word*.

I feel there is a cost to using the mnemonic approach in laying out the keyboard. Specifically, the control-B for character-back is just too long a reach. I feel like I'm back in eighth grade tied to the piano chair forced to practice my finger exercises.

The eighty or so functions in MINCE add to the ease of use. For example, in addition to being able to go back one character, you can return to the front of the file, front of a sentence, or front of a word. Particularly nice for programming is the "newline and indent" key. It places you on the next line, indented the same distance as the current line. Surely intended for structured programming, it is also nice for assembler, where most of the time you are pressing return and then tab, to write the next op-code. With MINCE, if the line you are on started with a tab, and you want

(continued next page)

the next one to start the same way, just press the indent key. This feature is also convenient for text. For example, when writing the return address in a letter, where you indent forty characters or so; you can write the street on one line, city on another and date on the last, then press "newline and indent" and save typing all the tabs.

Another useful ergonomic factor is that the bottom of the screen shows what percent of the way through the file you are.

MINCE relies heavily on the ESC key. This makes the location of the ESC key on your keyboard quite relevant, as you will find yourself using it quite often.

I am sorry MINCE doesn't scroll the screen a line at a time. Instead, when you are about to go off the screen, it erases and redraws the entire screen, positioning the cursor in the middle line. You can "sort of" get it to scroll by using a re-draw control key; if you move the cursor down one line then press re-draw, it clears and redraws. This effectively only moves one line, although somewhat slowly.

There is one key assignment in the standard MINCE which I didn't care for and have changed by reconfiguration. Control-V is used to scroll one screen down, which is OK. To scroll up, however, you must press ESC-V. I like a full screen editor which performs all of its commonly-used cursor movement functions by allowing you to "park your pinkie on the control key", then letting you press the various keys necessary to move up or down a line or a screen, etc. I just didn't like switching from control-V to ESC-V.

CONFIGURABILITY. You initially set up MINCE for a specific terminal, by running a configure program. It has a terminal selection menu including eighteen terminals!

In addition, if you have BDS C, you can completely tailor the keyboard to your needs. You can even write your own command processors, or modify the supplied ones. This is most impressive if you aren't afraid to tackle it. I was, and only peeked at the code with a disassembler, and patched a few things. Then I decided to tackle a total keyboard reconfiguration, and was able to execute it in one afternoon. I even added a couple of relatively trivial functions to implement control keys for insert mode and overtype mode, which MINCE did not implement in a way I liked. When that worked, I tackled a bigger project – adding a new command to process characters with the high bit on, such as produced by PMATE and even more by WordStar. Without this change MINCE cannot adequately process files which do not have the high bit of each character zeroed. There are other alternatives: WordMaster, and CP/M's ED, and PIP with the [Z option, all produce files which have the high bit zeroed.

EASY TO LEARN. As I mentioned, I tried out every function MINCE was capable of in five hours. That is more a sign of how rich the functions are than of how hard it is to learn. The basic cursor movement, delete, and insert functions are easy to learn because of their mnemonic significance. With the one-page command summary at your side, you will have no trouble.

The lessons and documentation help the learning process too.

Objective Evaluation

Video-Related Criteria

FULL SCREEN. There are keys to move the cursor one line up or down, one character left or right, one word left or right, a sentence back or forward, or a paragraph back or forward.

The cursor-up and cursor-down control keys do not behave as they do on other editors I have seen; they remember the column and try to stay in the same column after movement.

There are keys to move to the beginning of the line, end of line, to open up a new line to type into, etc.

The repeat key (see below) nicely extends these functions.

SCROLLING. There are keys to scroll up or down one screen. The repeat key may be used. Separate ESC-key combinations are used to go to the top or bottom of the file.

There is NO line-at-a-time scrolling! This makes it hard to use MINCE to read a document. It completely redraws the screen any time you move the cursor off the top or bottom of the screen. It is quite hard to pick up where you left off.

File scrolling is transparent to the user because of the virtual memory scheme implemented with the swapping file.

INSERT. In the normal mode of operation, MINCE control characters move the cursor or perform commands, and printable characters insert into the file and on the screen. I used the configurability of MINCE to make the normal mode NOT insert. Instead I use a control key to select insert mode and another to go back to overtype mode.

OVERTYPE. Only in a mode called "page" do you overtype. It allows you to move the cursor anywhere on the screen. For example, when you move the cursor up from the end of a long line to a shorter line, it automatically extends the shorter line with spaces. Keys behave a bit differently; cursor-back doesn't wrap from the front of a line to the end of the previous line. Tabs automatically expand to spaces, making page mode unsuitable for assembler programming.

UNDO-KEY. This is a slick feature. At the bottom of the screen on the right of the status line is an *. When you have deleted something either by tagging a block and deleting it, or by pressing one of the line delete keys (delete to end of line or delete entire line), the deleted data goes into a delete buffer and *+ appears at the end of the status line. The + means you may do more deleting and it will append to the buffer. When you next do some non-deleting function such as cursor movement, the + turns off. At this time you still have the deleted text in the delete buffer and can yank it back in anywhere you want. Doing any additional delete operation will overwrite the delete buffer. Using a special control key, you may explicitly ask to append to a non-+ delete buffer.

The MINCE undo key is nicely implemented to allow recovery from disasters – erroneous block deletes for example – and facilitates a simple block mark-and-copy operation.

REPEAT. WordMaster introduced me to the idea of a repeat key in an editor, much as a child might be introduced to eating out by going to MacDonald's. MINCE's repeat key is more like a fine out-of-the-way French Restaurant that you take that special someone to. Like WordMaster, pressing it once causes the next key to repeat four times. It may repeat itself, each time multiplying by four, allowing up to 16384 repeats.

The real charm of MINCE's repeat key is the ability to handle an *arbitrary* repeat count. Once you have pressed the repeat key, you may take the default of four, or press any numeric keys to override it. Want a line of seven dashes? Just press: repeat, 7, -, and you get: -------. I feel this is simply *superb* and should establish a standard for *every* editor to follow.

TEXT EDITING ABILITIES. MINCE has more of a text editing flavor than any of the other editors I reviewed. It has a fill mode, which wraps a word to the next line when you pass the right margin while keying in. I'm using it right now to write this article. It does not diddle with special control characters in the file; for instance, it doesn't turn on the high bit of ASCII characters like WordStar or PMATE do. Instead it defines a paragraph as being delimited by two consecutive carriage returns, or a carriage return followed by a tab.

Caution: If you have a document which does *not* meet MINCE's criteria for paragraphs (for instance with just a five-space indent) typing ESC-Q to reform the current paragraph will instead lump the entire document into one huge paragraph. When it goes away for a long time to do so, you will find it is un-interruptible. (ugh!)

In general MINCE is a good text editor with document, paragraph, word, and character-based commands, word-wrap, and page mode. It can reformat paragraphs but not justify them. It does not have any ability to send data to a printer. A separate text processor, Scribble, is available. I am not familiar with it but I presume it to be very good, considering the quality of MINCE.

Command-Related Criteria

NOTE: MINCE does not have command strings in the sense that the other editors in this review do. Being used to editors with such command strings, *I* consider that a shortcoming. Anyone *not* familiar with command strings (such as in CP/M's ED) would not miss it. Commands which are absolutely necessary in any editor, such as "find and change", are available, and invoked by command keys which then prompt for arguments.

Here is an example of a typical use of commands for an editor which *has* commands: I had a file of about 2500 file names (my MAST.CAT) and wanted to delete all .BAK names from it. In an editor with a command mode I could search for each occurrence of .BAK then go to the front of the line and kill the line, all by typing one command string.

MOVE/DELETE/INSERT/TYPE. These do not apply to command mode since they are implemented in full screen mode.

FIND. Both forward and reverse searches are supported. A control sequence causes a "Forward Search <ESC>" or "Reverse Search <ESC>" prompt. You key in the string and press ESC. To find the next occurrence, specify just ESC as the string.

CHANGE. Two forms of change are supported. One defaults to replacing all occurrences, but may take an argument for the number of changes to make, and one is a "query replace". The latter stops at each possible change and awaits one of several characters to either skip to the next occurrence, replace, quit, or finish doing all the rest without stopping to ask.

MOVE and COPY. These are implemented in an almost ideal way: by marking one end of the block to be moved or copied, then going to the other end and issuing a command to either move or copy the the marked section to the kill buffer. You then move elsewhere and re-insert it. It may be re-inserted multiple times. Although the marked block is invisible, pressing control-X twice swaps the current cursor point with the place previously marked, thus somewhat outlining the marked block.

I found this ability very useful in assembler programming. For example in a program I had typed:

```
CPI     'C'
JZ      CANCEL
```

and then wanted to add:

```
CPI     'S'
JZ      SEND
```

This is a trivial example, but I tagged the front of the line containing the first CPI then went past the JZ line and typed the "wipe region" command. Then I pressed the "yank killed" key to bring it back and pressed it again to bring it back a second time. If I had wanted to have, say, ten CPI/JZ instructions, I could have just hit the Yank key 10 times (or using the repeat key: repeat 1 0 yank i.e. 4 key-strokes to bring in 10 copies!)

COMMAND STRINGS. Simply non-existent in MINCE as I mentioned in NOTE above.

MULTIPLE EDITS. MINCE is very comfortable editing several files without returning to the CP/M command prompt. When you are finished editing one file and want to start with another, just save the first file and create a new buffer (control-X B then give a name). Then kill the old buffer so there is room in the swap file (an unnecessary step when editing small files). After that, read in the next file to edit. Alternatively, after a file is saved, control-X control-R will read a new file, replacing the previously saved contents of the buffer.

You can also edit several files at once. A quote from my tape recorder which I used while preparing this review: "After about two weeks, it is very easy to get used to the MINCE capability of editing three files at once. You find it is really quite natural, and you start to wonder how you got along without it." At times I would have a comment on the recorder about PMATE then one on MINCE then one on VEDIT. Using MINCE, I edited three notes files at once – one for MINCE one for PMATE and one for VEDIT. I could switch from one to the other quite easily.

As I am typing this, I am using the split-screen ability of MINCE with six lines of my notes file at the top of the screen

(continued next page)

and 14 lines of this review on the bottom. NICE! It is also useful for comparing two files, say .ASM files, or for picking pieces out of one file and putting them in the other.

Another example: I had some documentation I was working on. First I wrote the overview. Then in writing the detailed documentation, I split the screen with a few lines of the overview at the top forming an outline, and used the rest of the bottom of the screen for keying in the details. Before having MINCE, I would have had to use a printed copy of the overview - but what fun is that?

File Related Criteria

BACKUP. Simply non-existent in MINCE. When I first worked with MINCE I considered the lack of backups a dreadful shortcoming. Perhaps I had been spoiled by ED and WordMaster and such editors that automatically create a backup for you.

I have now learned to live without them but force myself to maintain my own backups with PIP. I do not *like* living without automatic backups but do not consider it a show-stopper.

If you have limited editing space you will appreciate the ability to edit without a backup. However, you must consider the space taken by the swap file.

SAVE. You can easily save the file you are editing with a two-keystroke command.

QUIT. You can kill a buffer, thereby discarding the edit, or you can quit back to CP/M. In either case, if you have made changes, MINCE cautions you with a question such as "Abandon Modified Buffers?".

READ. To insert a file from disk into the one you are editing, you must switch buffers, read the file, mark and delete the buffer, then switch buffers back and bring in the other buffer. I know of one "big system" implementation of EMACS that supports inserting a file directly into the current position in the buffer.

WRITE. You can write out a block indirectly, by moving it to another buffer, then writing that buffer. Not as easy as writing the marked block, but it does the job.

NOTE on SAVE, READ, and WRITE: When you use the READ or WRITE ability of MINCE, the file name which MINCE keeps track of changes. This is something you have to watch out for. Here is an example: I said MINCE B:XYZ.ASM - a new file. Since it was a new file, I wanted to bring in my skeletal .ASM file of macros, equates, and subroutines, which I frequently use to help me get started on a program. To do so, I typed in control-X control-R which is the READ command, then said EQU.LIB, the name of my equate and subroutine library. Later, after deleting about 80% of the skeletal code and writing quite a few new lines of code, I did a control-X control-S (SAVE) and found that because EQU.LIB was the last file referenced, my XYZ.ASM was ignored and the file written to EQU.LIB. The READ command is treated more like a "which file is it you want to edit" command. This is well-documented but is a significant pitfall for the unwary, which I certainly am. In addition, it won't work except on an empty buffer which I had when I edited a new file. Apparently, you should bring the file into another buffer, then bring pieces of it over as you need them. A better alternative is to copy the skeletal program BEFORE executing MINCE: PIP XYZ.ASM= EQU.LIB then MINCE XYZ.ASM.

DIRECTORY. MINCE does not have a way to get a directory listing into the file.

I'm not sure if I have enough knowledge, or if MINCE has enough expansion room, but some day I would like to ADD the directory function to MINCE. (The fixed location of the MINCE global variables limit how much code you may add to it).

Statistics

MINCE retails for $175. By now, all the dealers should be handling version 2.6 or later as that is the version with the significant speed improvement. MINCE.COM takes 30K of memory. It also takes a swapping file whose size is dependent upon how you plan to use MINCE. 64K is recommended.

Room For Improvements

MINCE effectively implemented a quite complete subset of EMACS. A couple features that weren't implemented, but which would be nice to have, are:

(1) the ability to insert a second file into the file being edited;

(2) support of numbered buffers like EMACS with MINCE's naming feature; it is easier to type a single digit number than to remember the full exact buffer name, when switching buffers.

(3) When you're editing a new buffer MINCE should assume you want to create a new buffer instead of asking you.

(4) EMACS and its host operating system allow escaping to the operating system to execute commands. This is particularly nice during program development when you might wish to escape to the operating system, do a compile, then return to the editor to fix the mistakes the compiler found. It is more practical than with any other CP/M editor.

(5) Some commands are not interruptible. For example, a replace ESC-R defaults to changing ALL occurrences of the string specified. If you realize you forgot to give it an argument of "one", you can't seem to interrupt it.

The virtual memory swapping scheme suffers a bit on floppies. There is no solution to this other than perhaps a hard disk, which makes the swapping barely noticable. This problem is most evident when editing a very large file. The swap file must be somewhat bigger than the file being edited. If the swap file and the file being edited are on the same disk and some distance apart you may think you are running a "seek test", not an editor. This is a price you must expect to pay to gain the advantages of such generalized multi-file multi-edit capability.

The choice of keyboard characters was dictated by faithfully mimicking EMACS. Thus such implementations as control-V to scroll one screen down and ESC-V to scroll one screen up, while not comfortable, are *properly* done. The ability to configure MINCE using BDS C eliminates this as a real problem.

The lack of a true "macro" command language means that there will likely be tasks for which you would like another editor. CP/M's ED or a low-cost full screen editor such as WordMaster or VEDIT would serve the purpose. It would be nice, of course, to have the power of MINCE *and* command strings. However with the .COM file already at 30K it would become unwieldy, or require overlays.

The paragraph-fill command tosses redundant spaces making typographically correct "period space space" into incorrect "period space".

Recommendations

There are several things I at first would have liked changed. The addition of a ".BAK"capability would have been helpful, and I would have preferred MINCE not to rename the file when you read another file. However, the longer I use MINCE, the more comfortable I become with these. So, the only changes I would *really* like to see are:

(1) support for scrolling a line at a time; and
(2) modification of the feature which shows up the percent at the bottom of the screen. It is updated every keystroke – a bit annoying.

Conclusions

MINCE set out to do a task: to bring the power of a large system editor EMACS down to a microcomputer scale. That goal was very nicely accomplished. I'm sure this will put the pressure on the authors of other editors (such as PMATE) to add split screen features.

MINCE is an interesting, enjoyable editor. In a delightful way it is very different from the others in this series of reviews. There are many things I haven't mentioned which help create this image. For instance control-T transposes the last two letters you typed. If you frequently get these transposition "finger checks" you'll appreciate this quick solution. Also the ability to capitalize words or make them entirely lower or upper case is nice. These features are more text-oriented, but find use in C programming where case is used to distinguish between global variables, defined symbols, local variables, etc.

If you have a need for multi-file, split-screen editing (as I do in reviewing CPMUG material and preparing abstracts) or if you are a strong BDS C fan and would like an editor which has much of the source supplied to dink with, then MINCE is for you. Also, those who have access to EMACS or an EMACS-like editor at work will certainly enjoy the familiarity of MINCE. Others whose editing tends more to text but not word processing, and who don't know of or feel they need a command language, will like it. With the addition of the Scribble text processing package it should make quite a formidable combination.

Support

Support from Mark of the Unicorn is so good that I added this paragraph to the review. They responded to my letter, and provide a dial-in number for assistance. They even helped a user (Joseph Freda) by customizing a version for his IBM 3101 display, to use its special function keys. In his words "...Mark of the Unicorn went beyond the call of duty in the matter of cooperation". Their replies to my review reflect a sincere interest in the quality of their product.

Comments From Mark Of The Unicorn

I sent an early rough draft of this review to Mark of the Unicorn. Here are *some* of the points they made:

Regarding my displeasure with MINCE renaming the buffer after reading a file into it: "Consider, however, if you were using just one buffer, read in a file, edited it, then wrote it back out. Then, read in another file, edit it, then C-X C-S [save it] and you've bashed the first file you were editing."

They further commented: (1) "Perhaps the modifications are really sequential operations the way you think about them"; (2) "The virtual memory scheme is happier if you throw away stuff which you don't really need"; (3) "You don't understand buffers yet, and just want to do something simple". I guess they're right, especially on that last point! The more I use MINCE, the more natural it becomes.

Regarding the control-V and ESC-V for scrolling: "We will be changing [ESC]-V to Control-Z soon."

Regarding documents which have 5-space indents: "...can be solved by using a tab instead of five spaces for a new paragraph."

In summary, they said "All that stuff above is minor, and much of it based upon a difference of opinion on how the universe should 'really work'". Right! I "grew up" in a "WordMaster universe", and tend to measure others by it, right or wrong.

They also sent a product announcement for "the Final Word", a completely new editor and formatter. "It will include a keyboard configuration program so that you don't have to hack C code, directory access, more modes with extra memory space, user command macros, and a buffer state-save option that lets you come back to editing whatever you were working on whenever you like! An MS-DOS version will hit the market soon after the initial introduction."

———

Next month yet another editor – PMATE. It is so powerful it is almost as much a programming language as an editor. Another month, a VEDIT review. Then, a little surprise: a $35 CP/M Full Screen editor. While somewhat more limited than the other editors, it is very usable, and implements configurability nicer than *any* of the "big" editors.

I also received favorable comments from John Dye, about Software Development and Training's "ED-80", and will be looking into that.

When all the individual editors have been reviewed, I'll present a wrapup review comparing the editors to each other.

As usual, I solicit your comments to me ℅ *Lifelines/The Software Magazine*, 1651 Third Ave., N.Y., N.Y. 10028.

Features

Custom Enhancements To WordStar

Robert Van Natta

Much has been written about WordStar from MicroPro International and it has been highly praised for its versatility and usefulness.

Rarely, however, have I seen information on how to squeeze the last ounce of performance out of WordStar or to customize it otherwise to the user's personal preferences or tastes. Although the following remarks are specifically written for a Radio Shack Model II, the ideas are generally not unique to that machine and may be applied to others.

Using Those Special Function Keys

There are seven keys on the Radio Shack keyboard which could be described as 'Special Function' keys. On the right of the keypad there are the two keys labeled 'F1' and 'F2'. These keys, as everyone but the rankest of novices knows by now, generate ↑A and ↑B codes. There is no way, short of attacking the innards of your keyboard with a pair of wire cutters, to make these keys generate other than those codes; they, therefore, present no opportunity for software patchers. Likewise, the HOLD key generates a null, and, so far as this writer can tell, cannot be implemented. To the left of the keypad, however, are four keys with arrows on them. They generate, from top to bottom, codes 1CH, 1DH, 1EH, and 1FH. The arrow down key (1FH) on the bottom is patched by the WordStar INSTALL program to duplicate the DELETE or RUBOUT function common on other terminals. The actual, but unlabeled, RUBOUT key (7FH) is the rather obscure control dash (↑-). It would thus be possible to patch the arrow down key to some other function while retaining full control of WordStar. This would compel the operator to use the more inconvenient ↑- key for the delete left functions, which I suspect would be unpopular.

This keyboard limitation leaves an odd set of three arrow keys unsupported by WordStar. Patching WordStar to give some direction to these arrows is an honorable goal and an effort will be made hereafter to document the procedure for doing so.

These patches may be installed with either DDT or using the WordStar patcher located in the back end of the INSTALL program. The use of both is described in some detail on pages 27 and 28 of the January 1982 issue of *Lifelines*, so some knowledge of these utilities is assumed here.

This writer has read a lot of patches consisting of a bunch of numbers with no explanation of what is being done or why. This bothers me greatly – I always expect explosions, smoke and other unmentionable occurrences during patch installations. Some months ago, for example, I happened across a patch (*not* in *Lifelines*) for simultaneously erasing the directories on all four of my drives in one easy operation. This illustrates my belief that a theoretical justification for a given patch should be presented first, not only to help the ulcer condition often associated with patch installation, but also to give the reader the chance to use his own inventiveness to apply this concept to other versions of WordStar. So this concept can be applied to enhancing the installation of WordStar on any terminal which has unused special function keys.

WordStar (WS.COM) contains a keystroke dispatch table. In layman's terms, this table sifts every keystroke entered and directs it to the address of the appropriate control function. Keystrokes which are not recognized by the table are ignored, or, where appropriate, are accepted as text to be entered in the file. This is why nothing happens when you push a key that is not supported or is not on the list of known commands. It also provides the theoretical basis for making those 'dummy' keys on your keyboard do something. You need only add the key to the list of keys searched for and pair it with the dispatch address of the function being emulated.

The table thoughtfully has blank space for 8 or 10 keys on the end of it, just waiting for your additions. Sounds simple! But wait! Where are the dispatch addresses? That's simple, too. Just read the table, and mimic the address that follows your favorite key. (More on that later.)

The next problem is to decide which functions you would like to have your dummy keys perform. Three is an odd bunch and you should give it some thought before doing it because the patches become familiar; others who use the system will cut off your DDT and maybe a few other things if you sneak out in the night and change them. It is hard enough to run a computer without some yo-yo rearranging the keyboard from time to time, so I suggest you get it the way you want it the first time.

It was my conclusion after some thought that the most useful and appropriate uses for the 3 keys would be the ↑QS, ↑QD and ↑KD functions; although, frankly, the ↑QE function was a close second to the ↑KD with the ↑QP not far behind. My justification for this choice was based on the fact that the ↑QS and ↑QD functions would not only be useful, in that a single key would duplicate relatively awkward functions requiring a three key sequence, but at the same time do something that the key label suggested, i.e. move the cursor in the direction indicated by the arrow. My choice of the ↑KD function for the arrow up key was made only after some reflection, and was done solely on frequency of use. In hindsight, it has been well accepted by my office staff who daily use my computer (coupled with a NEC printer for heavy duty word processing).

How Are The Patches Installed?

In WordStar versions 2.xx and 3.00 the empty part of the dispatch table will begin at label **XTAB:**, about 0649h in version 3.00 and a few bytes lower in version 2.xx. For what it's worth, version 1.00 doesn't respond to **XTAB:** and the beginning patch address is 0551h. (See Note 1 at the end of the article.) To use the patcher in INSTALL.COM, proceed through the menu installation routines (this may either be a first time installation of WSU.COM or a reinstallation) and reach the patcher by answering 'N' to the question 'ARE THE MODIFICATIONS TO WORDSTAR NOW COMPLETE?'

The first question will be:

```
LOCATION TO BE CHANGED(0=END): XTAB:
```

This will, in version 3.00, display address 0649h and advise that the present value is **00** and invite you to enter a new value. If the old value is not **00**, a patch has already been made to this table, possibly by the factory install routine (the current Radio Shack routine doesn't, but if you are using this idea and applying it to some other computer you might run into this), and should not be overwritten. (Instead, advance to the first zero byte and begin patching at that location.)

The addresses that follow should be thought of as being in groups of four bytes each. The first will be the hex value of the first key, and the second will be the hex value of the second key where two key combinations are being used (or null for single key commands); the third and fourth bytes will be the dispatch address.

Succinctly stated, the three keys that I have described may be enabled by patching the following values in this order, beginning at XTAB: or address 0649H:

```
1C,00,F1,63,1D,00,D3,63,1E,00,02,02
```

If you are using the patcher in WordStar version 3.0, it will automatically advance to the next address if you simply enter a carriage return in response to the subsequent questions of addresses to be changed.

As suggested above, DDT.COM is a reasonable alternative, and unless you are scared silly of using it, this is a good time to give its use consideration. DDT is about as user unfriendly a program as I can imagine, but it gets the job done effectively and rapidly. Its advantage is that you may modify an already working version of WordStar without having to reinstall the whole thing and can make the patch even if you can't remember which disk you left your INSTALL program on. Simply enter the command **DDT WS.COM**. Appropriate drive references may be used, if necessary, and in due course you will get the echo:

```
DDT VERS 2.2
NEXT PC
3F00 0100
-
```

Make a note of the 3F, as this is needed to compute the SAVE command that you will need later when the patches are complete.

Now enter the command **s0649**. In response to this you should get: '0649 00'. This means that the current value of address 0649 is 00 and that if you enter something it will be substituted for the 00. In this case you should enter '1C', which will cause an advance to the next memory location. Then you can [in sequence] type in the values listed above for the patcher.

As you do this you will get a column of numbers like this:

```
0649  00  1C   < SPECIAL KEY BEING IMPLEMENTED
064A  00  00   < EMPTY UNLESS TWO KEY COMBINATION
064B  00  F1   < TWO BYTE DISPATCH ADDRESS FOR
                 ↑QS FUNCTION

064C  00  63
064D  00  1D   < SPECIAL KEY BEING IMPLEMENTED
064E  00  00
064F  00  D3   < TWO BYTE DISPATCH ADDRESS FOR
                 ↑QD FUNCTION

0650  00  63
0651  00  1E   < SPECIAL KEY BEING IMPLEMENTED
0652  00  00
0653  00  02   < TWO BYTE DISPATCH ADDRESS FOR
                 ↑KD FUNCTION

0654  00  02
0655  00  .    < TERMINATE SEQUENCE WITH PERIOD
-↑C            < HIT THE BREAK KEY AFTER YOU ENTER THE
                 PERIOD TO EXIT DDT.
```

Once you have exited DDT with the break key or a **g0** you are ready for the SAVE command. The DDT Users Guide says the correct way of executing the SAVE command is to SAVE using the high order byte of the top load address and converting it to decimal. (See manual page 2). Stated plainly, you must convert the first two digits of the first number that displayed when you loaded DDT to a decimal number and use it in the SAVE command. This is why you were told to write the '3F' down somewhere. If you are a math whiz you can convert this mathematically to a decimal based number but there is an easier way.

In the very back of your Radio Shack Owner's Manual is an appendix which nicely lists out the various hex values and their decimal equivalent. (I mention this because it is the only useful item I have ever found in this document.) Read down the table and you will find that the equivalent is 63. Accordingly, enter **SAVE 63 WS.COM** to write your patched version of WordStar back to the default disk. You may use a drive reference to place the program on another drive (such as **SAVE 63 B:WS.COM**). If you save onto the same drive as you loaded off of, you will overwrite your original file. You may avoid this by saving on a different drive or under a different name. (See January 1982 *Lifelines* for the problem and solution if you want to permanently assign a different name to your new version.)

You are now ready to test your patch. Run your revised version of WordStar in the normal fashion. Edit a file you don't

(continued next page)

value and hit the arrow keys. The Left arrow should move the cursor to the left edge of the screen, the Right arrow to the right end of the text, and the up arrow will save and exit to menu; and, of course, the traditional ↑QS, ↑QD, and ↑KD will still work as they always did. But once you get used to these new enhancements you will never use them again.

But I Want The Keys To Do Something Else!

If you have an earlier version of WordStar, or don't like my choice of commands for the unused keys, all is not lost. You need only jump through one more hoop. You must learn to look up your own dispatch addresses. Do this by reading the dispatch table yourself. Once again load WS.COM with DDT and display using the DDT 'd' command the contents beginning at about 04A1.Do this by entering **d04A1** after the '-'. This will display about half a screen of table. The memory contents will be displayed 16 bytes wide with the address of every 16th byte in the left hand column. If you don't see what you are looking for the first time, you can display more by simply entering the letter 'd' again for another 12 lines. Read the table just like you would read a book, i.e. from left to right. You will see two bytes for a key combination and two bytes for an address, alternating clear down to location 0649 where you will find about three lines of empty space. There is no easy way to figure out which are addresses and which are key codes, but a little imagination is very helpful. Most of the empty bytes are places for the second key in single key commands. Thus, if you see a sequence where every third byte is null, as in the patch suggested above, you can tentatively conclude that the empty byte is the second byte of the key pair. Secondly, look for repetition. All the ↑Q commands are, for example, bunched together. So when you find a sequence in which every 3rd byte is 11, you can pretty well be sure you are looking at the ↑Q portion of the table. So when you read down to about 04B1, you will find the sequence '11 13 F1 63 11 04 D3 63'. Get the message?

If you still don't, here is one more chance. The ASCII values of 11,13, and 04 are ↑Q,↑S and ↑D respectively. Suffice it to say that if you decide to get hyperactive in this area with your patcher, you can actually change the entire WordStar command structure (probably for the worse).

In summary, to implement unused keys on your keyboard in a redundant fashion do the following:
1) locate the dispatch address for the desired function in the dispatch table;
2) locate the end of the dispatch table;
3) patch the keyboard value of the unused key followed by an empty byte (if creating a one key combination) followed by the the dispatch address.
4) If implementing several keys, repeat routine as necessary.
5) Do not overwrite anything or attempt to patch beyond the end of reserved blank space at the end of the table.

Notes

1. The dispatch addresses given are also for version 3.00 and are different in each version. Lookup procedures are described *infra*.

GOSH, THIS COMPUTER SURE IS TAKING IT'S TIME!
YEAH...
IT SLOWS DOWN LIKE THIS WHEN THE GRAPHICS LAB RUNS PARTICULARLY COMPLICATED PROGRAMS
MAYBE I SHOULD GO CHECK ON THEM
COULDNT HURT
?
GRAPHIC LAB →
MUMBLE MUMBLE-
MEET! MOOT!
!
WATCH OUT! ALIEN TO YOUR LEFT!!
ZAT! ZOOT!
Seaton & Bess 82

A Lisp Computing Environment

Harry Tennant, Ph.D.

The environment of an organism, according to Webster, is the "complex of factors that act upon an organism and ultimately determine its survival and form". The computing environment of a user could likewise be defined as the complex of factors that acts upon the user, determining productivity and the form and extent of computations. As this definition suggests, the environment is the combined effect of a collection of factors. The environment determines the user's productivity by making the facilities easy or hard to learn and by providing the user with the functionality to make tasks easy or hard to complete. As a result, the environment determines what will ultimately get done. Users have a limited amount of effort to expend on their tasks. If the environment is too limiting, ambitious projects will not even be attempted. Attention to detail tends to increase as it becomes easier to accomplish the more basic tasks.

A great deal of attention has been given to developing good computing environments for Lisp programmers. This is due, at least in part, to the fact that it is generally easier to improve the programming environments in Lisp than in other languages (since programs can be manipulated as data in Lisp - see appendix). It is also due in part to the fact that Lisp programmers, usually working on enormous experimental artificial intelligence programs, have had a more urgent need for congenial programming environments. This article will describe the computing environment designed for users of The Stiff Upper Lisp. We will discuss some of the factors that influence the users and what has been done about them in The Stiff Upper Lisp.

The Stiff Upper Lisp is a dialect of Lisp that runs under CP/M (the CP/M version is available from Lifeboat Associates) and TRSDOS (the TRSDOS version is available from Tennant and Tennant Computing). It has been engineered to include many of the most appealing features of the Lisps used in artificial intelligence research, and yet it is small enough to allow the user sufficient room in which to compute.

Dialogues

When a user is interacting with a computer, the two are involved in a dialogue. The key difference between a dialogue and an arbitrary collection of utterances is that there is coherence to a dialogue. A context is established, and there is structure to what the two participants say and when they say it. Human dialogues are characterized by very detailed contexts. In contrast, computer systems typically don't maintain any context.

Lisp is an interpretive language which provides an environment for the user. When programming in Lisp, the user is "in" Lisp. The same is true for BASIC and APL. These interpreted languages provide a workspace for the user. Within the workspace the user may define and test functions one at a time and set global variables. As the dialogue continues, the changing collection of programs, variables and values indicate what has transpired in the dialogue: it is part of the context.

Dialogue History

Coherence is maintained in human dialogues through repeatedly referring to the same concepts. Some specialized mechanisms have been developed to make repeated references easier. Some of them are:

Pronouns	he, she, it, they, them, etc.
Definite Noun phrases	the ball, those idiots, these ideas
Ellipsis	What is the square root of 2? Of 5?

In order for these mechanisms to work, one not only needs to maintain a context of what is being talked about, but also when it was mentioned in the dialogue. The flow of the dialogue itself, the sequence of utterances, becomes part of the context.

In The Stiff Upper Lisp a history is maintained of the utterances from both the user and the system. The user is prompted by one of eight numbered prompts (*1, *2, ... *7, *0). The system stores the user's command expression with the prompt (the value of the variable *1 becomes whatever expression the user typed in when prompted by *1). Also, after the user's expression has been evaluated, the value returned by the system is stored with a corresponding variable (=1, =2, .. =7, =0). Thus, the history maintains the context of the most recent portion of the dialogue.

For example, as pictured in Figure 1, the user types in the expression (setq my-variable 'howdy). This expression sets the value of the variable MY-VARIABLE to HOWDY. It then returns the value HOWDY. When the user types in an expression, the prompt variable, *3, automatically gets set to (SETQ MY-VARIABLE (QUOTE HOWDY)). (Notice that 'howdy is a shorthand expression of (quote howdy)). The variable =3 is set to the value of the expression, HOWDY.

The dialogue history can be used by any user-defined function, since *1, =1, *2, =2, etc. are just ordinary global variables. However, there is a function provided with The Stiff Upper Lisp, REDO, which is made for dialogue history related operations. Simply typing (REDO) will cause the last expression to be redone. Typing (REDO *2 thru *5) or just (REDO 2 thru 5) will cause the sequence of commands prompted by *2, *3, *4 and *5 to be redone.

One of the useful applications of REDO and the dialogue history is in correcting

(continued next page)

minor typing errors in commands. If a user types "pirnt" instead of "print", as in,

```
*3 (pirnt 'howdy)
Unknown Function Name: PIRNT
*4 (redo using print for pirnt)
```

the user can now type (REDO using print for pirnt). The substitution of print is made for pirnt, and the expression is evaluated again. In this example, of course, it would have been easier to just type in (print 'howdy), but had the original been a ten line expression, the user would be extremely grateful for the simple REDO substitution.

If a more serious error occurs in a long expression, the user may enter the editor, edit the expression, then redo the edited version.

The REDO facility is also useful for developing and debugging the body of a loop, prior to putting the body into a loop construct. The user can work on perfecting a loop body then give a command like (REDO 2 thru 5 50 times) or (REDO 2 thru 5 until (equal flag-var 'stop)).

The dialogue history is maintained automatically at the top level. It is done by Lisp calling the function TOPLEVEL which assigns the input to the prompt variable (*1, *2, ...) evaluates the input, then assigns the output to the corresponding output variable (=1, =2, ...) (incidentally, the variables ** and *** are also assigned the last input and the last value of ** respectively). The function TOPLEVEL is available to the programmer making it easy to put the same interface into the programs he or she writes. This serves to make the human interface consistent.

Digressions

Another feature of human dialogues is that they do not always proceed smoothly. A speaker may start, then digress, then return to the original point. In interactive programming the same thing happens frequently, especially when writing loops. Programmers very frequently will begin work on the body of a loop, then have to go back to initialize a variable before the loop can be executed. Another common interrupt is that a user will get deep into writing an expression and forget the syntax. Then it is necessary to digress to check the manual, then perform the computation.

The Stiff Upper Lisp provides a facility for digressions. The facility is splicing readmacros. As the Lisp-reader is reading in an expression, if it comes across a readmacro, the readmacro (a function), is evaluated immediately. Readmacros can perform arbitrary computations, so they are equivalent to interrupting the current read operation with another process. If the splicing readmacro returns NIL (which is easy to arrange), the readmacro will leave no trace behind in the input stream. The interrupted read operation will then be continued as if the interrupting readmacro had not been there. This interrupt capability is extremely useful.

In the example shown in Figure 2, readmacros have been used four times. First, the ;E readmacro allows the user to enter algebraic expressions in familiar algebraic form. The code of the readmacro converts it to the somewhat clumsier Lisp form, then inserts that generated Lisp code into the input stream. The next two are uses of the ;@ (called DO IT!) readmacro, which allows the user to digress from the task to do something else, then pick up where he or she left off. DO IT! reads the next expression in the input stream, and evaluates it, but leaves no trace in the input stream. Thus, the user can digress to examine the on-line help or set a global variable without disrupting the primary task of typing in the LOOP statement. In the example in Figure 2, the user uses the DO IT! readmacro to use the on-line help facility and to set the value of a variable that should have been set before writing the LOOP expression. The fourth readmacro, ;;, is the comment readmacro. It reads to the end of the current line (the comment), but leaves nothing in the input stream.

Ignoring Details

In programming, it's the little things that count. A programmer will approach a problem with a limited amount of time and a limited amount of energy. The small frustrations the programmer encounters consume time and waste energy. Reinventing frequently used utility functions is unproductive and monotonous.

Garbage Collection

The Stiff Upper Lisp provides for some needs that most other languages do not. First, it provides dynamic memory management. When all the Lisp cells are exhausted, the garbage collector is invoked to search the memory for cells which were used temporarily but are not currently in use. These are "garbage" cells. The garbage collector then collects these cells and makes them available for future use: it recycles the garbage. In large programs (i.e., large enough to fill the programmable memory of the computer), memory allocation tends to become one of the most constant and troublesome concerns for the programmer.

In the Stiff Upper Lisp, the user need not be concerned with dynamic memory allocation because Lisp does it for the user.

Position Independence

When programming in BASIC, one frequently faces the limitations of line numbers. In the midst of editing, one must stop to renumber, then find ones place and continue editing. Stopping to renumber is not an oppressive burden, but it is an interruption that hinders rather than helps the programming process. It adds some mental overhead and another opportunity to forget what one is doing. Programmers don't need that.

In Lisp, programs are position independent. The memory is (mostly) composed of a large workspace. Programs and data can reside anywhere within the workspace. As a result, any combination of programs and data can coexist in the workspace (as long as there are no name conflicts).

Position independence not only facilitates editing, but also promotes the development of program libraries. The fewer familiar algorithms the user must reimplement, the more time can be devoted to solving the problem. Think of what a disadvantage it would be if every time one wanted to write a program that multiplied two numbers, one had to write a multiplication routine. High level languages anticipate a variety of user needs (such as mathematical functions) and provide them. Many more could be provided, but aren't.

Since programs can live together in the Lisp workspace, one can put frequently useful programs into a library, then reuse them instead of rewriting them when needed. BASIC prevents effective and convenient use of software libraries because of line number conflicts.

Output Control

Another common but needless annoyance occurs when a display of information goes scrolling off the top of the screen before it can be read. The Stiff Upper Lisp keeps a count of the number of lines that have been sent to the screen. When the screen is full, it stops the display, types "more..." and awaits action from the user. The user may allow the display to continue on to the next page, or terminate it and return to the top level of Lisp. The system also keeps track of the number of characters printed so far on the current line. Before printing each word or number, the system checks whether there is room on the current line to print it. If there is, it is printed. If not, the system goes to the next line to avoid splitting words or numbers across line boundaries.

Finally, The Stiff Upper Lisp provides a pretty-printer. The pretty-printer displays programs and data in a format that shows their structure, making them much more readable. Instead of printing,

```
(lambda ($exp) (cond ((atom $exp)
$exp) (t (cons (cdr $exp) (car $exp))))
```

the pretty-printer prints a much more readable form, making the structure (or, more importantly, errors in the structure) obvious.

```
(lambda ($exp)
    (cond ((atom $exp) $exp)
        (t (cons (cdr $exp) (car $exp))))
```

Debugging Facilities

The process of programming will always produce bugs. The Stiff Upper Lisp, like most interactive languages, provides facilities for identifying bugs so they may be corrected. One advantage that Lisp has over the other languages is that it is better suited to the symbol processing tasks that are required in debugging. The Stiff Upper Lisp provides a trace package, which shows when each traced function was entered and exited, what the parameter values were when called and what the return value is. The break package allows the user to stop computation at a selected point, and allows the examination of the current stack of expressions being evaluated, the current values of any variables (either global variables or local variables) and the variable binding stack. The user has the full power of the Lisp top level available, so is free to perform whatever computation he or she likes (including redefining the very function that caused the break!). Finally, the single step facility allows the user to step through computations one expression at a time. This is particularly useful in locating catastrophic bugs. The course of the computation can be monitored up to the point where the bug occurs.

If a bug takes a long time to fix, it is probably because the programmer is trying to fix the wrong piece of code. Once the code that actually holds a bug is pinpointed, repairs are usually trivial and quick. Finding bugs is therefore the key to debugging. The tools provided in The Stiff Upper Lisp provide enough control over the computation and enough insight into the dynamic state of computation to make it much easier to locate bugs.

The Stiff Upper Lisp also provides an editor that "knows about" the structure of list structures. The user can use this structure editor while in the Lisp environment. It is not necessary to leave the environment to edit a function, then read it back in (although the user can do this if he or she prefers). (See Figure 3.)

Customization

One of the greatest advantages of Lisp is its extensibility. Users can define functions which, once defined, are indistinguishable from the functions that came built in with the system (in Lisp, all procedures are functions: they all return a value). Lisp allows the user to have his or her definitions of functions replace those that the system came with. If you don't like the way LOOP works, for example, you may define your own definition of LOOP, and from then on, that will be the one used. In BASIC, by contrast, one could not redefine the IF-THEN-ELSE, PRINT or FOR-NEXT. In Lisp, nothing is sacred.

This level of extensibility enables the user to make the environment do just what he or she would like it to do. It can be thoroughly customized, down to the level of changing the language itself, if that is desired.

The Stiff Upper Lisp supports customization and personalization by reading an initialization file. When the user enters the system, an initialization file is read. The file can contain definitions of the functions and data values the user wants his or her environment to include. By automatically reading the initialization file, the system can always appear just as the user wants it without any further effort on the user's part.

Autoloading

The most limited resource in contemporary small computers is memory space. It cannot be squandered. There is thus a tension between the desire to include extensive functionality and the need not to fill up memory with infrequently used (although useful) functions. The answer is autoloading. Instead of actually putting the function definitions into the Lisp workspace, surrogate function definitions are put there. When the function is called, the surrogate definition is evaluated. The surrogate loads the real definition into the workspace from disk, then the real definition is evaluated. (See Figure 4.)

Final Remarks

The quality of interactive computing environments has gradually improved over the years, especially in the Lisp community. More has been learned about what would be useful to include in the environment.

The extensibility of the Lisp environment has allowed users to experiment with new facilities, whereas in other languages improvements must come from the systems designers or maintainers, or not come at all.

In addition, Lisp users have been working interactively longer than other users. They have simply amassed more experience.

(continued next page)

The facilities described in this article are included in The Stiff Upper Lisp. Few, however, were invented for The Stiff Upper Lisp. The Stiff Upper Lisp has borrowed a small subset from the tools collected into Interlisp, UCILisp and Lisp Machine Lisp. These dialects are very large, wholly impractical for use on a computer with a small address space and slow floppy disks. However, these limitations are no excuse for making computing environments as bleak as they usually are. The computing environment provided with The Stiff Upper Lisp has been engineered to deliver a congenial environment within the constraints imposed by the limited systems it resides on.

Appendix

Programs are Data in Lisp

One of the most commonly cited advantages of Lisp is that programs can be manipulated like data. One can construct a list in a Lisp program, and if that list is a sensible expression (such as (setq my-variable (quote howdy))), the expression can be evaluated.

In the ordinary course of programming, one would not want to write programs that write other programs which in turn get evaluated (and, who knows, perhaps those write other programs and ...). Lisp makes it possible to do so, but the programs written by program writing programs are so horrendous to debug that the advantages have to be stupendous to justify the approach. The ability to manipulate programs as one would manipulate data is very useful, however, for building congenial computing environments.

There were many features in this article for which the ability to manipulate programs as data was essential. First was the history-keeping facility. Expressions read in from the user's terminal are set as the values of the prompt variables. Few languages would allow that. More significant, however, is the way that the REDO function can cause the history expressions to be reevaluated on demand, particularly after making changes (such as correcting a spelling error).

Another example was seen in the discussion of readmacros. the ;E readmacro read in a data expression (the algebraic expression), massaged it and output the equivalent Lisp program. Also, the DO IT! readmacro read in an expression and caused it to be evaluated immediately.

It is not unusual for an interactive language to have an editor. An editor is, of course, a program written to manipulate programs. The Stiff Upper Lisp has an editor, but BASIC has an editor too. The difference is that in Lisp, the editor can be written in Lisp (and in The Stiff Upper Lisp it is). In BASIC, the editor must be written in the language that the BASIC interpreter is written in. Consequently, a Lisp editor can be improved or rewritten, if one wishes to do so. In fact, many Lisp users improve their editors, adding pattern-matching search facilities, program templates, improved display facilities or whatever moves them. In BASIC, the editor is immutable. You are stuck with the same editor after ten years of use that you had on the first day. It is very likely that what you would like out of your editor would have changed considerably over those ten years.

The final example of manipulating programs as data is in the implementation of the TRACE and BREAK functions. When one traces a function, as in (trace fact) shown in the text, the trace function actually rewrites the definition of the function. In this case, the function, fact, is rewritten to first print out the entry information, next evaluate the body of the original function, then print out the exit information. The UNTRACE function rewrites the function again so it is back the way it started. Once again, most implementations of BASIC have some trace facilities. Like the editor, though, the trace facilities are not and cannot be written in BASIC. They are immutable (and generally insufficient). Trace facilities in Lisp, however, are just like any other program and may be rewritten by the user to fit his or her own needs and taste.

The ability to manipulate programs as data does not simply allow one to have an editor or a trace package or a break package or a redo package. Any one of these could have been built into any language, including BASIC, if the language designers had thought to do it. The difference is that with Lisp, the language designers don't have to think of everything. The users can invent facilities and integrate them into their personal copies of the language just as the language implementers can. This gives Lisp a future. It can continue to evolve after it is in the hands of the programmers. Evolution, enabled by giving programmers the ability to improve their computing environments, is what has made the Lisp computing environments far and away the most well developed environments of any language.

Harry Tennant is the author of The Stiff Upper Lisp.

A Call for Manuscripts

Perhaps you've done some writing before. Or maybe you've always wanted to write. It could be that reading *Lifelines/The Software Magazine* has given you some ideas on what you have to contribute. We're interested in hearing what you have learned, and so are other readers. Whatever *serious* CP/M-80 compatible software you've been using, we'd like you to write for us. We like to publish both long essays and those short gems which can hold so much important information.

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 for our Writer's Guide: Editorial Dept., Lifelines Publishing Corp., 1651 Third Ave., New York, N.Y. 10028. Telephone: (212) 722-1700.

MAG Announcement

Micro Applications Group has announced that it is changing the names of its products as of March 1st. For your reference, here is a list of the old and new names:

New	Old
MAG/sam3	MAGSAM III
MAG/sam4	MAGSAM IV
MAG/sam-E	MAGSAM-E
MAG/sort-M	MAGSORT-M
MAG/sort-C	MAGSORT-C
MAG/sort-R	MAGSORT-R
MAG/base1	PRISM/LMS
MAG/base2	PRISM/IMS
MAG/base3	PRISM/ADS

Figure 1

```
        *3 (setq my-variable 'howdy)    ;; *3 gets set to the
                                        ;; expression the
                                        ;; user typed in.
         HOWDY                          ;; The expression is
                                        ;; evaluated, setting
                                        ;; MY-VARIABLE to HOWDY.
                                        ;; The system also sets =3
                                        ;; to the returned value, HOWDY.
                                        ;; The values of *3 and =3 help
                                        ;; to establish the context
                                        ;; of the course of the dialogue.
             *4                         ;; The system displays the next
                                        ;; prompt, awaiting input from
                                        ;; the user.
```

Figure 2

```
        *2  (loop  (print  ;E  ((index+ 82)*4) )
                               ;; The first readmacro, ;E, reads
                               ;; in the expression that follows
                               ;; and converts it from the
                               ;; common algebraic notation to
                               ;; Lisp syntax,
                               ;; (times (plus index 82) 4),
                               ;; and inserts the Lisp code
                               ;; into the input stream.
                      ;@ (help loop)
               <help on the LOOP function displayed here>
                               ;; The DO IT! readmacro, ;@, reads
                               ;; the next expression and evaluates
                               ;; it immediately.  In this case, it
                               ;; is a call to the on-line
                               ;; help system to get a
                               ;; reminder on the syntax of LOOP.
                      exit ((equal index 100) 'finished)
                      (setq index (addl index)) ;@ (setq index 0) )
                               ;; Here the DO IT! readmacro is used
                               ;; initialize the variable INDEX,
                               ;; which the user forgot to
                               ;; initialize before writing the
                               ;; LOOP expression.
```

Figure 3

```
        *4 (de fact (n) (cond ((eq n 1) 1)
                              (t (* n (fact (subl n))))))
                                ;; Recursively define
                                ;; factorial.
        FACT

        *5 (fact 3)
        6
```

(Figure 3 continued next page)

Figure 3 (continued)

```
*6 (trace fact)          ;; Trace entries and exits
                         ;; for the function fact;
(FACT)                   ;; returns a list of all
                         ;; currently traced functions.
*7 (fact 3)              ;; Levels of embedding shown
!enter: FACT(3)          ;; graphically.
!!enter: FACT(2)
!!!enter: FACT(1)        ;; Calling parameters and
!!!exit: 1               ;; returned values displayed.
!!exit: 2
!exit: 6
6

*0 (untrace fact)        ;; Take fact off the trace list.
NIL

*1 (step t)              ;; Turn on single step feature
=T

*2 (fact 3)
:(FACT 3)                ;; Forms indented by :, values
                         ;; indented by =.
::(COND ((EQ N 1) 1) (T (* N (FACT (SUB1 N)))))
:::(EQ N 1)              ;; Step waits for the user after
                         ;; printing each form:
                         ;;    <space> to continue,
                         ;;    control-g to quit and
                         ;;      return to the top level,
                         ;;    S to turn off the stepper
                         ;;      and continue the
                         ;;      computation,
                         ;;    P to pretty-print the
                         ;;      current form.
===NIL
:::(* N (FACT (SUB1 N)))
::::(FACT (SUB1 N))
:::::(SUB1 N)
=====2
:::::(COND ((EQ N 1) 1) (T (* N (FACT (SUB1 N)))))
::::::(EQ N 1)
======NIL
::::::(* N (FACT (SUB1 N)))
:::::::(FACT (SUB1 N))
::::::::(SUB1 N)
========1
::::::::(COND ((EQ N 1) 1) (T (* N (FACT (SUB1 N)))))
:::::::::(EQ N 1)
=========T
========1
=======1
======2
=====2
====2
===6
==6
=6
6
```

Figure 3 (continued)

```
*3 (step nil)              ;; Turn off stepper.
:(STEP NIL)
NIL

*4 (break fact)            ;; Initialize fact for
                           ;; a break.
(FACT)                     ;; Returns all current
                           ;; break functions.

*5 (fact 3)
Break for FACT
BREAK: stack ok quit return eval    Level= 1
                           ;; A break has been entered,
                           ;; special options are listed.
ok                         ;; Proceed through the break.
Break for FACT
BREAK: stack ok quit return eval    Level= 2
                           ;; The next recursive call of
                           ;; fact is broken.
n                          ;; Display current value of n.
2
BREAK: stack ok quit return eval    Level= 2
(setq n 1)                 ;; The full power of THE STIFF
                           ;; UPPER LISP is available in
                           ;; a break.
1                          ;; This will alter the result
                           ;; (fact 3).
BREAK: stack ok quit return eval    Level= 2
ok                         ;; Proceed with the evaluation.
3                          ;; Altered answer because of
                           ;; intervention in the break.
*4 (unbreak fact)
NIL
```

Figure 4

```
                              ;; This is an autoloading
                              ;; definition of HELP.  It
                              ;; reads in the real definition,
                              ;; then evaluates that.
(df help $key
        (readfile help lsp)
                              ;; Read in the file
                              ;; HELP.LSP which contains
                              ;; the full definition of
                              ;; the function HELP.
        (apply 'help $key))
                              ;; Apply the function HELP
                              ;; to the same argument list.
                              ;; This time when APPLY looks
                              ;; up the definition of HELP,
                              ;; it will find the one read
                              ;; in from HELP.LSP.
```

Opinion
Zoso

Hello Dear Hearts,

In past columns I have tried to lighten things up a bit (get it?) with some mini travelogues, so very possibly some of you who saw the postcard I sent in lieu of a column last month were looking forward to hearing about my recent trip. Sorry! If you want to travel vicariously, read National Geographic instead. Through no special choice of mine, here comes another computer potpourri.

I know this is the April issue, but I'd be remiss in not mentioning something which caught my eye a few weeks ago. (I'm writing this some six weeks before you'll see it).

The editorial in Infoworld's February 1st issue was essentially an unabashed encomium on behalf of Minnie Floppy (written, some suspect, by M.F.'s alter-ego). This editorial hailed M.F. as the first writer to bring whimsy and amusing irreverence to a computer oriented publication. What can I say, Minnie, my lithe little punsmith? If Infoworld claims that you originated the Solid State Side Splitter, then I guess it must be true. I'll just have to keep looking for your columns which appeared prior to June, '80 (the month I began here). Please advise me, Minnie. I shall never again find inner peace until I have been richly amused by some of your formative masterpieces.

I have consistently disdained low-end hardware in this column. Perhaps the term 'toy computers' will jog your memories. That's all out the window now. The Bally Arcade 'computer' is actually kind of fun. Now don't get me wrong, you wouldn't want to automate an office with this machine but you can play some excellent video games with it. Believe it or not, this unit offers a wide variety of games which are almost identical in quality with those one sees only after parting company with a quarter. So, if you want to turn your brain into cottage cheese in the privacy of your home and save a pocketful of coins in the process, the Bally Arcade is highly recommended.

I don't know how many of you get as bored as I do waiting for disk intensive activities to complete; (editing, assembling, compiling, linking and the like). Even the best of floppy disk systems are slow and there are lots of very bad ones in the field right now. The standard wisdom has dictated an upgrade to hard disks, but when speed is more important than storage, there is a cheaper and faster way. For lack of a better name, let's call it 'silicon disk' technology. Essentially, a silicon disk is several hundred kilobytes of memory, implemented as a floppy disk. The idea is to do all the [usually] slow stuff on a silicon disk, while occasionally dumping the results on a regular magnetic disk just to play it safe.

Recently, several products incorporating the 'silicon disk' concept have been introduced. At this point, the only one I have seen in action is made by dy-4 Systems of Ottawa. The Orion system by dy-4 consists of impressively rugged hardware driven by equally impressive software. That's the good news. The bad news is that dy-4 only supports the STD bus. Whereas the STD is a splendid eight bit bus, it is hardly the most common.

In the software department, I know of a nearly completed project which will prove extremely popular with hobbyists and others eager to explore the inner workings of complex, highly proprietary software. Make no mistake, this item will cause lost sleep aplenty amongst principals of some of the most prolific software firms. I don't want to let the cat out of the bag just yet, but if you can conjure up an idea of the very last product some of these guys would like to see, then you've probably got it figured out already. I'll be keeping you posted.

Another nifty software item looming on the immediate horizon is 'BKG'. If you own a cursor addressable terminal and a Z80 CPU, you will be able to use your computer as a tournament quality backgammon opponent. Now I know most of you are thinking, "Oh no, not another backgammon game". This one is different!!! It's light years ahead of anything you've seen before. It features a splendid screen presentation and typically requires five seconds or less to compute and display the machine's move. The beta test version, which I have enjoyed for hours on end, appears as 28K of carefully hand optimized Z80 machine language. It will be sold with an equally impressive version of Gomoku. CP/M computer games have arrived at last!!!

For some time now, I've had this nagging feeling that there was more than coincidence behind the similar editorial content of *Time* and *Newsweek*. Roughly twenty weeks a year, the cover stories of these magazines will concern the same [often obscure] topic. I like to think of it as America's answer to Tass, Izvestia and Pravda, (that evil Soviet troika, dedicated to distorting and suppressing the news – or so we are told). In a most unusual turn of events, *Time* and *Newsweek* somehow fell out of synch with their respective cover stories about the microcomputer revolution. (*Time* published their version in the February 15 issue; *Newsweek* did so a week later). The article in *Time* was probably more informative than nothing. I'm afraid I can't say as much for the one in *Newsweek*.

What follows are some of the [expurgated] things I muttered to myself while reading various parts of that

Newsweek article. We'll use the familiar notation (N: =*Newsweek* and Z: =me).

N: "...software, the vital instruction menus that tell the computer what to do."

Z: Well, at least this garbled bit of techno-speak did include a computer 'buzzword' (menus).

N: "Japan, Inc., (whatever that means) is also interested in the lucrative personal-computer market and looms as a major threat to U.S. manufacturers..."

Z: I've been hearing this for at least a couple of years. All I can tell you is most Japanese computers which I've seen to date are derivative garbage. I'm sure they'll get the hardware cooking at some point in the future. From what I've seen so far, I doubt that quality software will follow unless it has been written in North America or Great Britain.

N: "... in fact, the Japanese make most of the disk drives, video screens and printers used in the United States."

Z: Head for the fallout shelters! The herd of flying bisons overhead are on a bombing run. In fairness, the Japanese do make some excellent printers. (The NEC Spinwriter is one of my all time favorites).

N: [about word processors] "...lets you and your computer manipulate words on a page, edit the text and print out flawless copy..."

Z: Weren't these the very programs used to create the abundance of unattractive and almost totally unreadable technical manuals which I see more often than not. 'Flawless copy', my [bad word deleted]!!!

N: "Load in the games software package that transforms your computer into an electronic playing field,..."

Z: Nice hyperbole! Could it be that N: meant transforming one's CRT into an electronic playing field?

N: "...'Programming is as simple as Lego Blocks or Tinker Toys,' says Atari's Alan Kay..."

Z: Back to the fallout shelters!

N: "...'Why does my CP/M card fail so often?' asked one man in the crowd. 'I sent mine back to the factory,' someone replied. 'They made some mods, put in a new EPROM and sent me a new book.' The questioner wanted to know why his Control Program for Microprocessors – software that oversees a system's overall operation failed. The man who answered explained that Apple had solved his similar problem by making some modifications, changing a chip and sending new instructions."

Z: There's nothing which impresses me more than adroit translations from spoken English to written English. Here, however, the treat is spoiled by a dearth of fact. To wit, the 'CP/M card' is Microsoft's 'Softcard'. It is not a product of Apple! Some versions of Microsoft's CP/M for the Apple prior to version 2.20B had problems. The appropriate fix involved updating a disk. In no case are we discussing firmware CP/M for the Apple. Some early production samples of the 'Softcard' required a single faster buffer IC and/or an extra capacitor to work properly, again not EPROMS. In fact, I have never seen the telltale EPROM 'skylight' on any chip used in the 'Softcard'. More importantly, it's been quite some time since the Softcard and accompanying software have suffered from reliability problems. Just for fun, I'd like to meet the guy who sent his 'CP/M card' back to Apple for a new EPROM. On second thought, no! I wouldn't like that at all!

For the record; I've written things good and bad about Microsoft during the last few years. In both cases, I called things as I saw them. With this in mind, here's a belated and heartfelt compliment to Microsoft for the 'Softcard'. This product definitely established new standards for excellence of concept, design, documentation and packaging.

N: "There is more software made [sic] for the Apple than for any other computer,..."

Z: Well I'm sure this will come as a quite a shock to DECUS and a few comparable outfits.

N: [about the Xerox Star] "... the Star uses a 'mouse' – a small hand-held device to guide the computer's [cursor] around the video monitor..."

Z: No, no, no! It's a 'turtle'. Here's an easy way to keep things straight; the turtle has a shell, the mouse is a furry little rodent.

N: "In a famous experiment, MIT professor Nicholas Negroponte drove through Aspen, Colo., taking pictures every five feet. The photographs were loaded on a video disk, and viewers can simulate a drive through Aspen."

Z: Why did he do that? Let's assume the professor wanted to simulate a drive anywhere. The first thing he'd need would be a bogus steering wheel and some dummy pedals. If, on the other hand, he'd wanted a visual representation of a drive through Aspen, he could have gotten much more viewable results with a roof mounted Super-8 camera. Also, one doesn't 'load' things onto video disks anymore than one loads things onto phonograph records. Both are 'read only' media. I suppose one could use a video disk as a table top or small hammock and load things onto it, but I doubt if that's what N: was trying to tell us.

Late breaking flash: (The following morsel of reliable gossip is for your edification only and has nothing to do with Professor Negroponte's 'experiment'). I have just found out that a New York State based mega-conglomerate is engaged in top secret development of laser scanned video disks which the user can also 'program' (by laser etching). The possibilities for mass storage of programs and data are awesome. Anyway, I chanced upon this news scoop by getting one of this companies research engineers sufficiently drunk to divulge some of the details. As I don't want this resource to dry up (my friend getting fired), I'm not going to name the company he works for. Suffice it to say that I've always wondered why an outfit which makes such great duplicating machines had not yet gotten into the computer business...

N: "...Estridge [of IBM] envisions using the computer to summon and explore anything mankind can record..."

(continued next page)

Z: Why not! I suppose a fair test would be to 'summon' up a live, in person recital by Luciano Pavarotti in the comfort of my living room and to invite you know who (in her tight Calvins) over for the 'explore' part.

Perhaps the most telling aspect of all this is that in addition to all the editor types who approved the final copy, no less than five *Newsweek* correspondents were responsible for this 'Home is Where the Computer is' article. Back when *Newsweek* used to cost only a quarter a copy, there was a better bargain to be had in content as well.

Finally, my friend Charlie G. called from Los Angeles to tell me about the new Apple II he had bought for his personal amusement. (Lord knows he deserves some amusement. He is a physician who performs human autopsies – no fooling). Anyway, C.G. asked me why all his little $30 and $40 dollar Apple games were copy protected and why all the far more costly CP/M stuff in my collection wasn't. My learned reply; "Beats me, Charlie".

Bye for now,
Zoso

Attention Dealers!

There are a lot of reasons why you should be carrying Lifelines/The Software Magazine *in your store. To provide the fullest possible service to your customers, you must make this unique publication available. It will keep them up to date on the changing world of software: on updates, new products, and techniques that will help them use the packages you sell.* Lifelines *can back up the guidance you give your customers, with solid facts on the capabilities of different products and their suitability to a variety of situations. Now we can also offer you an index of all back issues of* Lifelines, *opening up a full library of information for you and your customers.*

For information on our dealer package, call (212) 722-1700, or write to Lifelines Dealer Dept., 1651 Third Ave., New York, N.Y. 10028.

Gift Subscriptions

You should consider gift subscriptions to *Lifelines/The Software Magazine* for your friends and relatives who are involved in microcomputing. As you probably realize from your own experience, the price of a subscription is small for the money *Lifelines* can save you in a year. Just send a check or credit card number and fill out the form below*. (Or call [212] 722-1700.) We'll send your gifted one a note to let them know of their good fortune, *and* we'll send you a free Zoso T-shirt. (Don't forget to tell us your size.)

Your name and address:

Name ____________________
Address ____________________
City ____________ State ____ Zip ______

Shirt size ________
☐ Check enclosed
VISA or MasterCard Number
Expiration Date

____

____
Signature (if payment is by credit card)

The name and address of the gifted one:

Name ____________________
Address ____________________
City ____________ State ____ Zip ______

*All orders must be prepaid by VISA, MasterCard or check. Checks must be in U.S. $, drawn on a U.S. bank. Subscription rates are $18 for twelve issues (one year) when the destination is the U.S., Canada, or Mexico. For subscriptions going to all other countries, the price is $40 for twelve issues.

Features

What About SpellStar?

Robert Van Natta

Editor's Note: This is the first section in an evaluation of spelling checkers and spelling correction software. Next month, James Mills will cover several other products.

Surely no one has looked at WordStar version 3.00 and not at least wondered about the "S" command for SpellStar. Dictionary programs are this year's fad in word processing and SpellStar is MicroPro's effort to avoid missing out on it. The November 1981 issue of Byte carried a comprehensive comparison of five leading spelling programs, but SpellStar was not on the list because it was released long after the deadline for that publication.

For the sake of convenience, the benchmarks used will be similar to those used in the aforementioned review. The Byte author used a Superbrain and this writer used a Radio Shack Model II; thus the times will not be directly comparable; it appears that the 8-inch drives give much better performance than the 5-inch Superbrain drives.

For example, using SpellGuard, this writer can complete the corrections of the benchmark test in under three minutes - over forty seconds faster than the reported time for the Superbrain. However, the Superbrain required a disk change, which the Radio Shack didn't, and the Radio Shack would read the file and do the actual checking in twenty-seven seconds flat compared to fifty six seconds for the Superbrain. Similarly, the Radio Shack would complete the proofreading of a ten thousand word file using SpellGuard in fifty four seconds compared to a reported 1:06 for a three thousand word file on the 5-inch machine.

How is Spellstar Put Together?

SpellStar consists of two files. The first is SPELSTAR.OVR. It is 30k in size and is the program overlay. The second is SPELLSTAR.DCT, the dictionary. It consists of about twenty thousand words and consumes 98k of disk space.

How Does SpellStar Work?

The operation of SpellStar is very simple. First, you push 'S' off of the main menu of WordStar. Then you enter the name of the file to be checked or maintained, followed by a declaration of whether you wish to check spelling or do dictionary maintenance. Next, a menu of spelling check controls must be reviewed, which presents an opportunity to designate the dictionary and a supplemental dictionary. (SpellStar will allow a check of your text against two dictionaries at the same time.) Finally, a chance is given to designate a work drive (more about this later) and the checking begins. All in all, however, the menu sounds worse than it is, because default choices are provided and intervention is only needed if variation from the default is required.

Once the checking is complete, an opportunity to abandon the check is presented along with an option to see a list of the suspect words *en masse*. Finally, the suspect words are all flagged in a file which is of the type .@@@. WordStar is then reloaded in a quasi-edit mode and the review begins. (Translation: Superficially it looks like the edit mode but doesn't work like the edit mode because the keyboard is dead except for the five keys which control the disposition of the marked words.)

Any WordStar user having version 3.00 can display the review menu with the command ↑QL, regardless of whether they have the SpellStar option. You can first set the help level to 3 (so all instructions show). Second, load some file into the editor that has some null bytes in it. These will display as a ↑@ on your screen. Load it in the editor as if you were going to edit it anyway. Most likely you will see a Digital Research copyright notice and about half a screen of garbage. Now use the command ↑QL which is a working command in version 3.00 of WordStar, even though you don't have SpellStar. Answer the question about whether you want to begin searching at the beginning or end of the file, and you are off on a cheap trip demonstrating how the SpellStar review works. Remember to abandon the edit when you are done playing around or you will have murdered the file you played with; but with a little imagination you can get the feel of how the SpellStar review works without spending a dime.

The correction process works as follows: the dictionary checking routine flags all suspect words. You move through the file in a global search fashion, moving from word to word by hitting 'I' for ignore, 'D' for add to dictionary, or 'F' for fix word. Upon arrival at a suspect word the flag is automatically deleted and the word is highlighted (on highlighting terminals). If you choose to ignore or add it to the dictionary this will be done and the next word will be highlighted. If the fix option is chosen, then you drop into the edit mode with the cursor on the first letter of the suspect word and any editing of the operator's choice may be done (including saving the file, exiting WordStar, and going out to coffee); the SpellStar correction routine may be resumed at any time with a ↑L command (or a ↑QL if you have shut the system down or done global searches in the interim).

Words which are set aside for the dictionary during the error review procedure may be designated for the main dictionary or a supplemental dictionary, but are not actually added to the dictionary. Instead, they go into a file bearing the name of your file being edited with the type 'ADD'. They will never become a part of the dictionary until you expressly incorporate them using the maintenance routines.

The 'ADD' file can also be processed into a supplemental dictionary. This is useful if the same document must be processed repeatedly, as the words peculiar to that document may be made

(continued next page)

into a supplemental dictionary after the first run. This feature offers the advantage of a custom dictionary without adding clutter to the main dictionary.

What are the Good Features?

SpellStar is menu-driven, with reasonably good explanations on the menus. The checking-correction routines are similar to SpellGuard's, and are easier to use, since the mismatch review routine is inside of WordStar and the words are viewed in their proper context. The automatically disappearing flag, and the ability to bring the full power of the WordStar editor to bear on an offending word before going onto another word tend to encourage the use of the trip through the file as an opportunity to do general editing and spelling correction at the same time. Also, for what it's worth, SpellStar drops you at the beginning of the word when you enter the fix routine.

On the dictionary maintenance end, the comparison is much harder. In this respect, SpellGuard and SpellStar are not at all comparable. SpellGuard is, at least to some extent, self-maintaining. Words designated for addition to the dictionary are on occasion sorted automatically into the dictionary, without operator intervention. There is not much other maintenance possible, so SpellGuard would have to be rated 'simple, and easy to use' from a maintenance standpoint, but also 'weak'. By comparison, SpellStar has a very powerful (translation: complex) set of maintenance features. If you like menus with more choices than you will ever use or understand, you will love running maintenance with SpellStar. WordStar has never been criticized for being short of features, and SpellStar follows suit. The maintenance menu will let you do anything you can imagine to a dictionary, and more.

As far as documentation goes, SpellStar comes with a scant twenty-one pages of documentation, intended to drop into the WordStar manual as Chapter 13. It is similar in conciseness and clarity to the rest of the WordStar manual. It should be rated as adequate for the job.

What Are The Problems?

SpellStar, in this writer's opinion, has some very serious limitations. However, one problem is not with SpellStar: its difficulty with Lifeboat CP/M version 2.25b for the Radio Shack Model II (if WordStar is installed using the Radio Shack INSTALL routines which implement the memory mapped video board). In any event, this writer solved that problem by reinstalling WordStar using the ADM-31 terminal option.

The real problem with SpellStar is that it is not only slow but also a drive hog. The Radio Shack with SpellGuard will complete the Byte benchmark through the checking routine in a fast forty seconds; the forty seven suspect words can be reviewed and the seven corrections made (using WordStar) in 2:57 minutes. With SpellStar the checking is done after a leisurely 1:23 minutes (with forty five suspect words - if soft hyphens were used in the text) and the entire correction job can be completed in 4:03 minutes. With a small file like the one used for the benchmark, the delay is only annoying and not horrible. But it gets horrible in a hurry if a large file is substituted.

Don't try too big a file! First off, although it isn't mentioned in the documentation, SpellStar bombs out during the checking routine if you attempt to check a file with more than 9500 words in it. An error message is displayed but the error is not trapped, so the reset button is the only recovery method. A 9500 word file is pretty sizable (22.5 pages single spaced or about 66k) but it was still a rude shock to learn of this limitation. It is frustrating to sit in front of your glass teletype for over six minutes staring at it uselessly before it gets to the crash point. The only way this writer could find to avoid the six minute wait for a crash on such a file was to attempt to run SpellStar on relatively full disks. In this event you would crash with a fully trapped disk-full error after only two or three minutes and could get on to something else without so long a delay.

The reason for SpellStar's delay is its irrepressible urge to write many large work files. This accounts for both the inordinate delays in getting the job done, and the huge amount of disk space required. For example, to process the maximum sized 66k file, file workspace of not less than 152k is required. Fortunately, the operator can designate a work drive and override the default designation of the logged in drive (a small consolation if you have only one or two cluttered disks to choose from, but a big help on systems with three or four drives). It boggles the mind to contemplate a word processing system so large that three or more double density 8-inch drives are necessary to make it work conveniently, but this is not much of an exaggeration if 50k and larger files are to be routinely managed.

SpellStar, in this writer's opinion, has some very serious limitations. The most trivial of the problems is its incompatibility with Lifeboat CP/M version 2.25b for the Radio Shack Model II (if WordStar is installed using the Radio Shack INSTALL routines which implement the memory mapped video board). It can be assumed that this problem will be cured by the bug catchers, as it this was not a problem in version 2.24, and version 2.25c has already been released (but who knows if this bug was fixed). In any event, this writer solved that problem by reinstalling WordStar using the ADM-31 terminal option.

Next the file BOOK.@@1 is sorted into a file called SORT.$$$. This file will be just as large as BOOK.@@1 (2k larger on this writer's test) and will contain all the same words, but they will be sorted alphabetically by length. Thus, if the source file contains the word 'an' twenty seven times, you will find twenty seven repetitions near the beginning of the file. Similarly, nineteen-letter words beginning with a 'Z' will be near the end of the file. Once this file is written, still a third work file is written, called BOOK.@@2. This file is much smaller and contains only the unique words. Its size is echoed to the screen as the "NUMBER OF DIFFERENT WORDS". On my test file it contained 1515 different words and consumed 14k of drive space. Once this third work file is complete, the two previous work files are deleted and the actual spelling check routine is ready to begin in earnest. Elapsed time for the 66k test file to this point is approximately five minutes.

Once this housekeeping is done, the dictionary files are opened and the unique word list is matched against the dictionary or dictionaries, as the case may be. This happens in about 1:15 minutes and then another work file is written containing the flagged suspect words (BOOK.@@3); finally, if you do not abandon the check, a flagged version of your original file (BOOK.@@@) is written and WordStar is loaded, with the cursor placed at the first suspect word and ready for review and corrections about 6:48 minutes after you started. Only after you do a "save" from the edit mode is your original file renamed to BOOK.BAK and your corrected version named BOOK. You may also find a work file showing up on your disks called BOOK.@@4. It appears as a companion to BOOK.@@3 but its function has not been deciphered by this writer.

As a final note about the work files, you should recognize that they are just that - work files. If SpellStar is run to its normal conclusion, they are deleted. However, any time you make an unscheduled departure from SpellStar, (which will be frequently until you remember to keep generous quantities of workspace available, and further remember to either log that drive in or expressly designate it upon loading SpellStar), several work files will be left behind. They should be deleted forthwith as some of them may be very large, and will add clutter quickly.

Conclusion

In conclusion, SpellStar has been a disappointment. This writer is a lawyer in a small office. SpellStar was acquired with the idea that it would augment WordStar in the daily production of dozens of legal documents. My hopes were shattered when my personal secretary flatly refused to use it, summarily declared "It's too slow!", and then added insult to injury by demanding to know which files related to it so she could recover the space on her system disk for other uses.

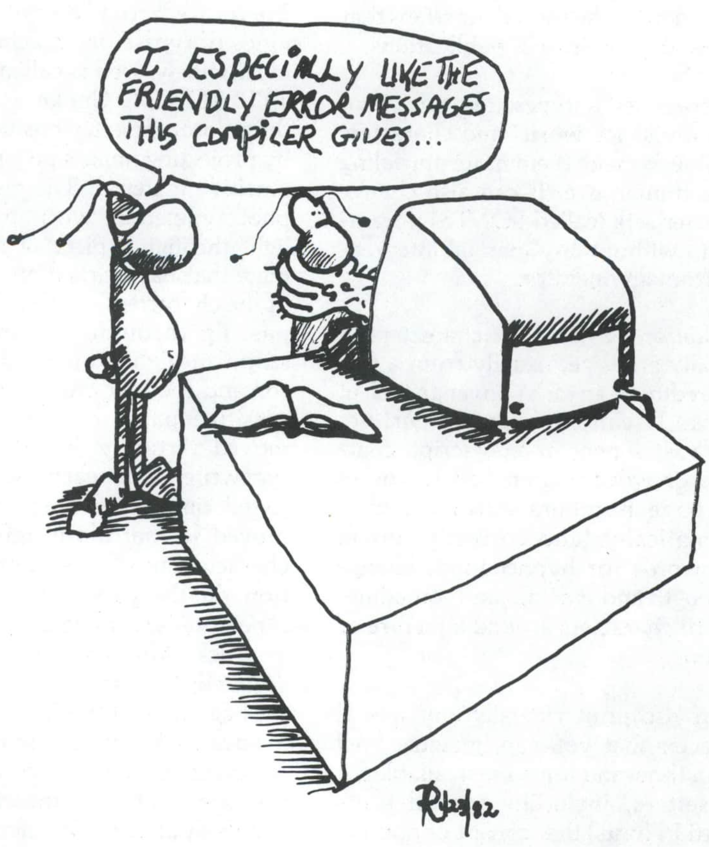
It is tempting to simply declare that it is an unsatisfactory product and leave it at that. Such a conclusion would be unfair, however. There are three general causes of unsatisfactory results in the computer world. Lame software is only one of those causes. Inherent hardware limitations, and misapplication or unrealistic user expectations are others.

On further reflection, this writer is convinced that the true failure is one of unrealistic expectations and poor systems analysis. A microcomputer may be applied to word processing tasks which range from a large volume of short documents (1 to 5 pages which are lightly processed) to a low volume of large documents (over five pages which are extensively processed). It is only rational to junk a one thousand dollar typewriter in favor of a eight thousand dollar micro-computer if there is a reasonable expectation of increased productivity. A moment of reflection will reveal that: a) spelling checking routines are non-productive; and b) their adverse effect on productivity is a function of how often they are used, and how fast they operate. It, likewise, logically follows that the frequency of use is going to be application-dependent. The Byte benchmark of about four hundred words would take a moderately productive typist eight minutes to type. If it takes four minutes to run through the spelling verification and word review, you have generated a whopping 33% production loss. Similarly, shorter documents or faster typists would aggravate the productivity drop even more. It is hard to believe that merely typing a little more carefully, or manually proofreading the documents would not be less of a penalty.

At the other end of the spectrum, it is obvious that the productivity penalty is going to be insignificant if the application consists of only a single document a day. This writer has reapplied SpellStar to occasional use for review of his free lance writings, computer program documentation, Supreme Court briefs, and the like - and found the results very satisfying.

It is, therefore, the combined wisdom of all the self-proclaimed microcomputer spelling program experts in St. Helens, Oregon (yours truly and my pet tomcat) that: a) spelling programs are not properly applied to production typing situations where the documents are short; and b) they work well for creative typists (translation: slow) working on relatively long documents; and c) the poor performance of SpellStar is hardware caused and could be expected to improve markedly on a high performance hard disk system (or would that be a hardware fix for a bad software design)!

(*Editor's Note*: Spelling checkers are recommended for those who write for publication!)


I ESPECIALLY LIKE THE FRIENDLY ERROR MESSAGES THIS COMPILER GIVES...
Rb 12/82

The Ultimate Printer?

Dave Hardy

Introduction

Most computer people think of printing as the stuff that comes out of the system list device of their favorite computer. Usually, the results look like a bunch of closely spaced dots that, upon close inspection, only vaguely resemble the characters they are supposed to represent. More expensive printers make better looking characters, but still, the results are often not as good as can be found in even the cheapest daily newspaper or magazine.

The reason for this is that most publications use a very specialized type of printer, called a TYPESETTER, to print their data. If you think you had trouble hooking your SpinWriter or Epson printer to your system, you should try connecting a typesetter! Although the hardware interface can be relatively simple, many typesetters have over a hundred different commands that are required to do the special functions that you see in professional publications.

For example, a typesetter can automatically space words and characters on a line to make them more appealing to the human eye. It can also change character sets (called FONTS) instantly, and without any manual intervention from an operator.

In addition, it can print characters in virtually any size, usually from a few hundredths of an inch to over an inch in height. It can italicize, underline, boldface, super- or sub-script characters or words, often add footnotes and page numbers automatically, automatically (and correctly) break long words for hyphenation, change line length and spacing, and sometimes even fit characters around a picture or drawing.

It can also print virtually any special character that you can imagine. Well over a thousand fonts are available for typesetters, including special fonts (called PI fonts) that consist of nothing but unusual characters (like the little TV channel numbers you see in TV GUIDE, or the little square boxes that are used on ballots to "check here").

Background Of Phototypesetters And Typesetters In General

As an engineer, and definitely NOT a Printer (which I spell here with a Capital "P" to show that I am referring to the person or job classification, and not the machine), I am really only slightly interested in the evolution of the Printer's tools. The following brief overview is presented to give the reader an idea of the printing industry's machines and to "lay the groundwork" for those who are not familiar with typesetting and printing in general. It is by no means, however, a complete picture of the printing industry or of all of the typesetting machines available.

There are actually several different kinds of typesetting machines, but the most widely used is called a PHOTOTYPESETTER. Unlike a line printer, the phototypesetter creates its output by projecting images on a piece of light sensitive material. The most common phototypesetters work by projecting light through a piece of film or glass plate that has marked on it the images of the characters it is to produce. After passing through this special "font strip," the light is bounced off of a mirror and focused onto a piece of light sensitive paper or film that can be moved vertically, like the paper in a typewriter. After each character is projected onto the paper, the mirror is moved so that it will reflect the next character onto the next horizontal position on the paper, the same way a typewriter's carriage moves after a key is struck. After an entire line has been "printed," the mirror returns back to the beginning of the line, and the paper is advanced vertically to the next line. If you think this is very complicated, you are right. It is theoretically very simple, but actually incredibly complex because of all of the very precise movements that must be made at high speed.

Less common than the "projecting" kind of phototypesetter, but rapidly becoming very popular, is the CRT phototypesetter. This machine works by passing photographic paper or film over a very high resolution CRT on which each character to be typeset is displayed. The image is painted on the CRT, usually a line at a time, using the same "raster scan" techniques that are used in a regular television set, except the scan is usually vertical, and the number of lines scanned per inch is usually between three hundred and nine hundred . After each line is "printed", the paper is moved vertically to the next line. It's rather like taking a snapshot of a TV set, except the picture tube is narrower and produces a much sharper picture. (And, of course, there are no commercials.)

The output of the phototypesetter is also one of its biggest disadvantages, since photographic paper is fairly expensive (up to $1 per foot) and requires chemical development, which requires a special machine called a photographic processor.

Recently, LASER-based phototypesetters have become available that can produce the same kind of output that a phototypesetter makes and at a much lower cost. They often use a metal coated paper (shades of AXIOM!) which eliminates the cost and hassle of photographic paper. Although LASER based machines can be much faster than conventional ones, they are not yet as widely accepted as conventional phototypesetters and may remain in the background for a while before becoming popular.

There are also still "old style" typesetting machines in use that actually cast each character in molten metal as it is typed by a skilled (and usually old) machinist-operator. These are the machines seen in the movies with giant mechanical levers and gears moving about, clicking and snapping while some old figure smoking a big cigar plucks away at an oversized keyboard that looks like the pipe organ used in "Phantom of the Opera." These "hot metal" machines are rapidly being eliminated and replaced by the newer more cost-effective phototypesetters (which are sometimes called "cold type" machines for their lack of molten metal).

"Hot metal" machines do have one advantage over "cold type" machines, however. Their output is actual metal characters, ready to be mounted on a printing press and used. "Cold type" must be photographed on a special film, which must then be sent through a "plate-making" process to make a plate that can only then be mounted on the printing press. Although "cold type" requires a few more steps to get a finished product, it is so much more efficient and easily produced that the extra processes it requires are relatively insignificant.

Why Interface A Small Computer To A Phototypesetter?

OK, so much for history. Now, why would anyone want to connect a computer to a typesetter? The reason is actually threefold - efficiency, quality, and cost.

Efficiency is increased because the amount of human labor required to create a typeset product is decreased. In many cases, data from the computer can be turned into camera-ready typesetter output without any operator intervention whatsoever. For example, a reporter's newspaper story can be sent directly from his/her word processor to the computer, formatted and translated, and from there be sent directly to a phototypesetter without ever being touched by human hands. This procedure works very well, and it is used routinely where I work, with impressive results. In fact, it is actually easier for me to get a CP/M file listing typeset than it is to get it listed on one of our SpinWriters!

The difference in quality between a typeset article and one that has been listed on a line printer is startling. If you don't believe it, compare some of the earlier un-typeset copies of *Lifelines* with some of the later issues that have been typeset. The typeset issues are

ASCII CODE	CHARACTER NAME	TTS CODE	ASCII CODE	CHARACTER NAME	TTS CODE
00H	TAPE FEED	00H	40H	@	36H 13H
01	THIN SPACE	01	41	A	06
02	EN SPACE	1D	42	B	32
03	EM SPACE	0B	43	C	1C
04	QUAD LEFT	1B	44	D	12
05	QUAD CENTER	3D	45	E	02
06	COMMAND	N/A	46	F	1A
07	BELL	17	47	G	34
08	UNSHIFT	3E	48	H	28
09	UPPER MAGAZINE	1F	49	I	0C
0A	LOWER MAGAZINE	05	4A	J	16
0B	UPPER RAIL	33	4B	K	1E
0C	LOWER RAIL	37	4C	L	24
0D	SCREEN RETURN	N/A	4D	M	38
0E	QUAD RIGHT	1F	4E	N	18
0F	SUPERSHIFT	25	4F	O	30
10	SHIFT	36	50	P	2C
11	1/8	36 3B	51	Q	2E
12	¼	36 27	52	R	14
13	3/8	36 03	53	S	0A
14	½	36 15	54	T	20
15	5/8	36 21	55	U	0E
16	¾	36 2B	56	V	3C
17	7/8	36 0F	57	W	26
18	EM DASH	36 0D	58	X	3A
19	+1 UNIT	36 05	59	Y	2A
1A	-1 UNIT	05	5A	Z	22
1B	START BLOCK	N/A	5B	SHIFT ADD THIN	36 09
1C	END BLOCK	N/A	5C	SHIFT EN LEADER	36 2F
1D	UNSHIFT ELEVATE	3E 04	5D	SHIFT EM LEADER	36 29
1E	SHIFT ELEVATE	36 04	5E	SHIFT VERT RULE	N/A
1F	DIVIDE	N/A	5F	CENT SIGN	N/A
20	SPACE	08	60	OPEN QUOTE (‘)	36 11
21	!	36 07	61	a	06
22	"	N/A	62	b	32
23	#	N/A	63	c	2C
24	$	07	64	d	12
25	%	N/A	65	e	02
26	&	25 0F	66	f	1A
27	APOSTROPHE (')	11	67	g	34
28	(	36 23	68	h	28
29	)	23	69	i	0C
2A	*	25 15	6A	j	16
2B	+	N/A	6B	k	1E
2C	UNSHIFT COMMA	19	6C	l	24
2D	HYPHEN	13	6D	m	38
2E	UNSHIFT PERIOD	39	6E	n	18
2F	/	N/A	6F	o	30
30	0	2D	70	p	2C
31	1	3B	71	q	2E
32	2	27	72	r	14
33	3	03	73	s	0A
34	4	15	74	t	20
35	5	21	75	u	0E
36	6	2B	76	v	3C
37	7	0F	77	w	26
38	8	0D	78	x	3A
39	9	31	79	y	2A
3A	:	36 35	7A	z	22
3B	;	35	7B	UNSHIFT ADD THIN	09
3C	SHIFT COMMA	36 19	7C	UNSHIFT EN LEADER	2F
3D	=	N/A	7D	UNSHIFT EM LEADER	29
3E	SHIFT PERIOD	36 39	7E	UNSHIFT VERT RULE	N/A
3F	?	36 2D	7F	RUBOUT	3F

TABLE 1. - A sample ASCII to TTS conversion table.
Note that some ASCII characters require 2 TTS characters for conversion, and that some characters have been omitted for clarity. The CHARACTER NAME columns contain the TTS character name, although it is usually also the ASCII character name as well.

(continued next page)

much clearer, easier to read and understand, and generally a good deal more appealing to the eye. If you've ever cursed the illegible program listings in the back of some of the popular computer magazines, then you know at least one of the advantages of typesetting. An old ASR-33 is just not going to produce output that can be printed in a magazine and easily read. Many publications refuse to typeset program listings because it is so easy to introduce errors into the program. Instead, they just photograph the original listings submitted by the authors, which reduces their quality even further. Sending a program directly from a computer to a typesetter could eliminate this problem completely.

(*Editor's Note*: Communicating listings from a computer to certain typesetters can be a problem however. On some typesetting equipment, it is impossible to dispense with proportional spacing of the characters. For this reason, typeset program listings can be hard to understand or incorrect.)

Sounds pretty good so far, but what about price? Well, a new, high-quality, high-speed CRT-based phototypesetter could easily set you back over $50,000, not including the interface to your machine. However, the popularity of CRT-based typesetters has caused the value of the older "projected light" phototypesetters to decrease dramatically to a point where they can often be purchased for less than the price of a SpinWriter. Many old machines (especially the Compugraphic 2900 and 4900 series) can frequently be purchased for just a few hundred dollars from printing houses that have updated to newer technology. Mergenthaler Linotype's VIP series phototypesetters can often be purchased for less than $3000, and these machines are still being used to produce high quality output by thousands of very demanding printing houses. Of course, they lack some of the features of the $50,000 units, but the quality of their output is very nearly as good. If you have a few more bucks, many manufacturers will be glad to sell you a small but adequate phototypesetter brand new, for $15,000 or so.

Type Face	[Fxxx]	
Point Size	[Px] or [Px.5]	
Height Change	[Hx]	
Width Change	[Wx]	
Film Advance	[Lp] or [Lp.25]	
Line Length	[Mpc] or [Mpc.p]	
Hyphenation	\AH=Allow \-=Discretionary	\XH=Cancel
Letterspace	\AL=Allow	\XL=Cancel
Italicize	\I=Allow	\R=Cancel
Stop	\S	
Immediate Film Movement	[Ap]=Advance	[Bp]=Reverse
Cancel Film Advance at EOL	\Z	
Kerning (in 54ths)	\ABxx=Negative	\AFxx=Positive
Kerning (in 18ths)	\Qx=Negative	\x=Positive
Character Spacing	\AKx=Tighten	\AEx=Expand
Indent	[IpLn]=Left	[IpRn]=Right
Hanging Indent Left	[IpH]	
Indent - Outline	[ItLn]=Left	[ItRn]=Right
Paragraph Indent	[IpS]=Set	\E=Enable
Cancel All Indents	[IC]	
Auto Leader	\Jx	
Ragged Right	\AR	
Ragged Left	\AG	
Ragged Center	\AC	
Cancel Ragged	\XR	
Set Tab Columns	[T0MpT1Mp...T29Mp]	
Start Tab	\T	
Justify Tab Column	_	
End Tab	—	
Start Vertical Tab	\VS	
Change Vertical Tab	\VC	
End Vertical Tab	\BE	
Display Superiors	\Ln	
Superiors	\Ucpn	
Inferiors	\Dcpn	
Bottom Rule	\B	
Horizontal Rule	\{M} or \{N}	
Vertical Rule	\{T}	
Auto Fraction	\Yx/x\Y	
Piece Accents	\Px	
Base-Line Jump	[Jp]	
Variable Start of Line	[Vpc] or [Vpc.p]	
User Format	[Dx[data]]=Define	[Ux]=Use
Breakpoint in Format	\\	
Postpone Format	[GDxUx]=Points [GLxUx]=Inches	[GNxUx]=Lines

TABLE 2 — A TYPICAL TYPESETTER COMMAND SET
p=Points, pc=Picas, n and x represent required numbers

Bear in mind, that, unless you purchase a machine that uses aluminized paper or something, you will also have to add in the cost of a photographic processor. You could spend anything from a few dollars (the amateur photographer route) to several hundred or even a few thousand dollars (automatic, motorized, temperature controlled, etc.).

What this all means is that, unless you are a VERY dedicated hobbyist, you are probably not going to invest in all this stuff just to read your master catalog in Bodoni Bold Italic (a common but distinctive font). But if you have a need for typeset output in your business, or want to start a small typesetting business to support your computer habit, this means of getting "on-line" could save you a small fortune and a lot of grief.

Requirements For Hardware Interface

Connecting a phototypesetting machine to a small computer is usually very easy, in the physical sense. Phototypesetters are really just mini-or microcomputers with very sophisticated video displays, so they are often remarkably simple to interface. Most phototypesetters use a paper tape reader as their main data input device, so getting your computer's data into them is usually no more difficult than removing the reader and wiring a simple parallel interface with a strobe and acknowledge signal. (In reality, there are usually a few problems that have to be worked out, but the basic idea is just to trick the typesetter into thinking that your computer is a paper tape reader.) Some typesetters also have serial (RS-232-C) input capability, which makes interfacing even easier. The Compugraphic EditWriter series is such a machine, and it even accepts ASCII input. If you have a typesetter that doesn't have a serial input, you can sometimes buy one from the manufacturer, but this is often an expensive (>$1000) option that requires adding another circuit board to the typesetter and implementing the manufacturer's protocol in software and hardware. It is very often easier to just use the paper tape reader.

Requirements For Software Interface

The software interface is almost always the most difficult part. Many of the simplest things to a computer printer become enormously complicated for a typesetter. This may be due to the fact that typesetting is mostly an art that has developed to produce things that are appealing to the human eye, while computer printing is the result of a more scientific endeavor that tends to make things less appealing to the eye but more appealing to the brain.

For example, in the computer world, each character to be printed occupies the same amount of horizontal space, say one-tenth of an inch. In the typesetting world, characters are assigned variable widths that are determined by the typesetter at the time that the line is printed. Although proportional-spacing printers like the DIABLO HYTYPE or NEC SpinWriter are readily available, none has yet achieved the very high resolution required of a typesetter. Not even dot-resolution graphics printers can match the ability of a typesetter, even though many manufacturers claim that they can.

Listing 3

```
10 '
20 '  WSTOTTS - Translate an ASCII file to TTS
30 '           and send it to an output port
40 '
50 DIM TTS(130)'          Make room for the conversion array.
60 LINE INPUT;"Enter ASCII file name:",FILNAM$
70 OPEN "I",1,FILNAM$
80 GOSUB 600'           Initialize some variables.
90 IF EOF(1) THEN END
100 LINE INPUT #1,A$'    Get a line from the file
110 FOR P=1 TO LEN(A$)'   and read it 1 character at a time.
120 PC$=MID$(A$,P,1)
130 GOSUB 190'          Translate each character and send it to
140 NEXT P'               the typesetter.
150 GOTO 90
160 '
170 '  TRANSLATE AND SEND subroutine.
180 '   Check for special case before translating.
190 IF PC$=CHR$(&H7B) THEN 410'  Special expanded character?
200 IF PC$=CHR$(&HD) THEN RETURN'  Ignore carriage returns
210 IF PC$=CHR$(&HA) THEN RETURN'  and linefeeds.
220 PN=TTS(ASC(PC$))'              Else translate the character
230 GOSUB 290'                     to TTS and send it out.
240 RETURN
250 '
260 '  SEND subroutine.
270 '  Send the character to the typesetter, but first
280 '   adjust the shift mode if necessary.
290 P5=PN
300 IF P5>128 AND SHIFT=0 THEN SHIFT=1:P5=27:GOSUB 360:P5=PN-128:GOTO 360
310 IF P5<128 AND SHIFT=1 THEN SHIFT=0:P5=31:GOSUB 360:P5=PN:GOTO 360
320 IF P5>128 THEN P5=P5-128
330 '  This section assumes that SUPERSHIFT is in effect for next CHAR ONLY.
340 IF P5=27 THEN SHIFT=1'         Set SHIFT mode if code=SHIFT
350 IF P5=31 THEN SHIFT=0'         Set UNSHIFT mode if code=UNSHIFT
360 OUT &H44,P5'                   Then send it to typesetter.
370 RETURN
380 '
390 '  SPECIAL CHARACTER subroutine.
400 '  Process special expanded characters.
410 PTEMP$=""
420 P=P+1'                          Look at next character in line
430 PC$=MID$(A$,P,1)'                to see which special character.
440 IF PC$<>CHR$(&H7D) THEN PTEMP$=PTEMP$+MID$(A$,P,1):GOTO 420
450 PN=TTS(0)'  Make PN$ NULL just in case illegal character
460 IF PTEMP$="N" THEN PN=TTS(10)'  EN SPACE    (These are the special
470 IF PTEMP$="M" THEN PN=TTS(1)'   EM SPACE     characters used here.)
480 IF PTEMP$="T" THEN PN=TTS(2)'   THIN SPACE
490 IF PTEMP$="U" THEN PN=TTS(3)'   UNSHIFT
500 IF PTEMP$="S" THEN PN=TTS(4)'   SHIFT
510 IF PTEMP$="UR" THEN PN=TTS(5)'  UPPER RAIL
520 IF PTEMP$="LR" THEN PN=TTS(6)'  LOWER RAIL
530 IF PTEMP$="NL" THEN PN=TTS(8)'  EN LEADER
540 IF PTEMP$="AT" THEN PN=TTS(9)'  ADD THIN
550 GOSUB 290'                     Then send the character out.
560 RETURN
570 '
580 '
590 '  INITIALIZATION subroutine.
600 FOR P=0 TO 127:READ TTS(P):NEXT P'  Read the ASCII/TTS array
610 SHIFT=0'                            Initialize SHIFT mode to UNSHIFT
620 PN=31
630 GOSUB 290'                         Send UNSHIFT code to typesetter
640 RETURN'                              to initialize it, then return
650 '
660 '  DATA FOR ASCII TO TTS TRANSLATION
670 '
680 ' This is the TTS look-up table, indexed by ASCII value.
690 ' These are actually REVERSE TTS codes, which means that
700 '  the TTS codes have been flipped (8-bit becomes 1-bit,
710 '  7-bit becomes 2 bit, etc.) and shifted right 2 positions.
720 ' Eight-bit SET means that the character needs SHIFT mode.
730 '
740 DATA  0, 52, 32, 31, 27, 51, 59, 58, 61, 36
750 DATA  46, 0, 0, 0, 0, 0, 0, 0, 0, 0
760 DATA  0, 0, 0, 0, 0, 0, 0, 0, 0, 0
770 DATA  0, 0, 4, 184, 127, 23, 56, 23, 163, 34
780 DATA  177, 49, 23, 23, 38, 50, 39, 162, 45, 55
790 DATA  57, 48, 42, 33, 53, 60, 44, 35, 171, 43
```

(Listing 3 continued next page)

Another problem that occurs whenever computers send data to typesetters is that most small computers "talk" in ASCII, while most typesetters "talk" in TTS. In case you've never heard of TTS, it is a coding system used by Printers that was developed to simplify the coding requirements of the old "hot metal" machines. TTS stands for "TeleTypeSetter," (which is a trademark of Fairchild Graphic Equipment Corporation) and I'm sure that any ten Printers you ask will give you ten different stories about how it originated. Needless to say, it is not a convenient code to work with. Unlike ASCII and EBCDIC, where character positions can be "calculated" ("C"="A"+2) and characters are in a nice straightforward order, none of the characters in TTS is in any recognizable order. The most believable explanation of TTS coding that I have ever heard is that the codes were assigned to each character according to frequency of use to reduce the wear on parts in the early mechanical typesetters. So an "E" (which is the character that usually occurs most in text) received the simplest code, and so on. I have no proof of this theory, but my boss swears that it is so, and since he signs my paycheck EVERY week, I accept it as Truth.

There are many other problems with TTS coding. TTS does not recognize case. Instead, it uses "shift modes" to decide whether a character should be in upper or lower case. TTS coding is less efficient than ASCII for computers, because it may take two codes to represent a single character, whereas ASCII has a unique code for everything. Table 1 contains a sample ASCII to TTS conversion table. I call it a "sample" table because ASCII and TTS can NOT be directly translated. TTS, although consisting of only sixty-four unique codes, has many characters that are not represented in ASCII coding, so these characters must be arbitrarily assigned to some ASCII value. Similarly, ASCII has some characters that can't be translated into TTS, and which must also be arbitrarily assigned to some TTS value. Fortunately, when sending data from a small computer to a typesetter, only the ASCII to TTS translation must be used. A few typesetters accept ASCII coding, which eliminates all of these problems, but most of the "good deals" that you can buy speak "TTS".

To make things even harder, almost every brand of typesetter uses a different input coding structure. Although there are several "generic" typesetting languages (e.g. CORA), virtually every type or model of machine has its own "custom" commands that are incompatible with most other machines. Even identical machines sold to different customers, or sold at different times, may have subtle programming differences that make it impossible to get the same output for the same given input coding! The problem is very similar to the problem encountered by small computer users if they try to run CP/M, CDOS, or SDOS software interchangeably. Some versions are upward-compatible, some are downward-compatible, some are totally incompatible. Basically, this means that any software written to send data from a computer to a typesetter is going to almost always be "one of a kind." This also explains why there are so few commercially-available programs to send data from a CP/M based machine to a phototypesetter. Table 2 lists some of the commands available on a typical phototypesetter, Mergenthaler Linotype Corporation's LINOTRON 202. To list all of the commands available would require a small book.

Listing 3 (continued)

```
800 DATA  54, 0, 62, 173, 23, 152, 147, 142, 146, 144
810 DATA  150, 139, 133, 140, 154, 158, 137, 135, 134, 131
820 DATA  141, 157, 138, 148, 129, 156, 143, 153, 151, 149
830 DATA  145, 8, 56, 40, 41, 2, 34, 24, 19, 14
840 DATA  18, 16, 22, 11, 5, 12, 26, 30, 9, 7
850 DATA  6, 3, 13, 29, 10, 20, 1, 28, 15, 25
860 DATA  23, 21, 17, 0, 47, 0, 0, 0
870 END
```

Typesetting Conversion Programs

Once the "low-level" problems in the preceding sections have been dealt with and (hopefully) neatly tucked away into some simple driver routines to take care of translating and transmitting ASCII data to the typesetter, the "high-level" software must be written. In other words, you have the interface, and now you have to tell your computer how to arrange its data to "feed" it. I've found that "feeding" programs usually fall into 2 basic categories, DIRECT CODE CONVERSION, and WORD PROCESSOR CONVERSION.

DIRECT CODE CONVERSION is the easiest method. It simply involves translating a file from ASCII to TTS. The ASCII file already has all of the codes in it that are required to tell the typesetter everything it has to know, and the conversion program inserts no additional commands or formatting. This program is fairly simple to write, but is of little use to anyone who does not know your ASCII to TTS conversion codes and who is not familiar with the input coding for your typesetter.

Listing 1 is a sample file that might be used as input to a direct code conversion program, and listing 2 is the actual output produced by the typesetter. Listing 3 is a program, written in Microsoft BASIC, that will translate an ASCII file into TTS and send the data to a serial port. This program is just an example, so some parts (most notably, the serial port handshake routines) have been left out.

Notice that the program keeps track of which of the characters is upper or lower case in order to send the appropriate TTS shift code whenever necessary. Notice also the routines to check for special characters enclosed in curly braces which cannot be generated by a standard ASCII keyboard.

Many commercial typesetting services offer this type of conversion with a simplified version of the typesetter's commands that is a bit easier to use, but the typesetter commands still must be entered into the ASCII file before it can be converted.

WORD PROCESSOR CONVERSION uses much simpler input files. In fact, this conversion uses regular ASCII text or word processor created files for its input. For example, this article (which was written with WordStar) could be used as input to such a program. There is no need to insert commands in the input file because the conversion program decides where and what any commands should be and inserts those commands automatically into the file when it translates it to TTS and sends it to the typesetter.

In order to insert commands, the conversion program has to make certain assumptions about the data it is translating. For example, it assumes a certain character size, font, etc. This method of conversion is most useful when the input files are "routine" data, like newspaper stories and novels, which are always in the same general format (same size, spacing, etc.).

Unfortunately, word processor conversion has its limits. Since the computer is left to make all of the decisions about how the typesetter is going to print the data, the person who wrote the file has very little say about how his or her file will appear when it leaves the typesetter.

Some simple commands are often added to the conversion program to solve this problem, but in general, when you let the computer make decisions about what parameters to use, it will surely screw up somewhere, sooner or later. For example, tabular data (like columns of numbers in a financial report, or a cross-reference table) is very difficult to convert from a word processing format to typesetting commands because of the way that most typesetters are programmed. (Many typesetters ignore multiple spaces, too.) In addition, there are many other problems of the same nature.

In all but the simplest cases, files to be converted in this manner require some form of "human intervention" before they can be successfully typeset. Fortunately, the simplest cases are the most common. Here at CDP, we routinely translate entire newspapers (except pictures and advertisements) from ASCII word processing files and send them to the typesetters with no operator involvement at all. (Many of the advertisements fall into the "human intervention" category because of their "non-standard" requirements, like pictures of giant hands pointing to a reduced sale price, or photographs of lettuce, etc.)

Listing 4 is a sample word processing input file, and listing 5 is the output produced by that file, after conversion. The conversion program appears in listing 6.

This program is the simplest form of word processor converter, and could be used to handle only simple paragraphed text with headlines. Although it could just as easily have been written to recognize the start of a paragraph as four or five spaces at the beginning of a line, this program defines the start of a paragraph as the first line after a single blank line. Anything preceded by two blank lines is treated as a "headline" case.

Most "high level" software is written (cleverly enough) in some form of high level language. We've used everything from BASIC to PASCAL and C and even assembly language. Of course speed is essential, but the only thing a language really needs to do the job is the ability to communicate with whatever hardware interface is required to drive the typesetter. This usually requires the ability to do direct I/O. A compiler is nice, but an interpreter is ok. The ability to jump to machine language subroutines and to link in assembly language modules is handy, too, since table look-ups and I/O are best handled in machine code. Obviously, it would be very difficult to work with any language that couldn't manipulate ASCII strings. It is also a good idea to bear in mind that the less cryptic the language and the more well commented your program is, the easier it will be to modify when you have to make changes six months after you've finished writing it.

Some Closing Remarks To Cheer You Up If You Have To Do This

Manufacturers are slowly beginning to realize the advantages of standardized input coding, but little progress has been made, perhaps because manufacturers are afraid that using a standard input code would allow customers to buy machines made by the competition. Currently, converting to a new input coding scheme can be cost prohibitive, due to the very high cost of retraining operators and reprogramming or modifying existing input equipment, so typesetter users are often obliged to stick with the same manufacturer just to avoid this extra cost.

Interestingly, the CP/M operating system has been at least partly responsible in bringing about some of the changes in standardization that have been made. Perhaps forced to by the sheer weight of the number of CP/M users (or at least guided by it), some typesetter manufacturers have begun offering machines that are capable of running CP/M, and others are looking very closely at CP/M-based machines that can front-end their products.

It is probably just a matter of time until the typesetter manufacturers make available all of the software discussed here, along with a machine that will be able to use it.

Although all of the hardware information given here is factual (and in use at CDP Corporation, my employer), parts of the software information, and the views given of the typesetting industry, are strictly my own opinion. Virtually every Printer I know would argue with at least a few of the things I have said here, and I'm sure that all of the manufacturers would disagree with me about several of my observations, too, but the ideas given here are well tried and work very well in most cases. Even if this isn't exactly what you need, it should at least help you to understand and expect some of the problems that you might encounter when you interface your computer to "the ultimate printer."

References

Stevenson, George A., Graphic Arts Encyclopedia, Second Edition, 1979, McGraw-Hill

Mergenthaler LINOTRON 202 Operator's Manual, 1978, Mergenthaler

Telephone Typesetting User's Manual, 1981, CDP Corp.

Hardy, David J., Fred Lasich, Sue Hayman, et al., Emulating on the 202, 1978, CDP Corp.

(continued next page)

Listing 1

[P1OF1M20L12]This is a sample of a [F34]direct code conversion[F1] file. Notice how easily fonts and [A45P30]character[P6] sizes[P10] can be changed. This is the most versatile, but most difficult to use method, because it requires knowledge of all of the typesetter's commands.<

Listing 2

This is a sample of a direct code conversion file. Notice how easily *fonts* and charac-ter sizes can be changed. This is the most versatile, but most difficult to use method, because it requires knowledge of all of the typesetter's commands.

Listing 4

This is a sample word processing file. Notice that there are no commands in this file and that it is a completely normal run-of-the-mill word processing file. Word processor conversion techniques work best on this kind of file, because this file has no unusual requirements.

Listing 5

This is a sample word processing file. Notice that there are no commands in this file and that it is a completely normal run- of-the-mill word processing file. Word processor conversion techniques work best on this kind of file, because this file has no unusual requirements.

Listing 6

```
10 '
20 '  WPTOTTS - Translate an ASCII word processing file to TTS
30 '            and sends it to an output port
40 '
50 DIM A(128),STD(50),SOP(50),SOH(50)' Make room for the arrays
60 LINE INPUT "Enter ASCII file name:",A$
70 OPEN "I",1,A$
80 SHIFT=0:'                     Initialize SHIFT mode to UNSHIFT
90 '  First fill the arrays.
100 FOR X=0 TO 127                READ THE ASCII TO TTS CONVERSION ARRAY.
110 READ A(X)
120 NEXT X
130 X=0'                         Read the standard line prefix.
140 X=X+1
150 READ STD(X)
160 IF STD(X)<>0 THEN 140
170 X=0'                         Read the start of paragraph prefix.
180 X=X+1
190 READ SOP(X)
200 IF SOP(X)<>0 THEN 180
210 X=0'                         Read the start of headline prefix.
220 X=X+1
230 READ SOH(X)
240 IF SOH(X)<>0 THEN 220
250 NULLINE=0'                   This is the BLANK LINE counter.
260 QUOTE=0'                     This is the OPENING QUOTE FOUND flag.
270 IF EOF(1) THEN 1140
280 LINE INPUT #1,A$'            Get a line from the file.
290 L=LEN(A$)'
300 NOTNULL=0'                   <== MEANS LINE IS NULL
310 IF L=0 THEN 350'             If line is 0 chars long, then it's NULL.
320 FOR X=1 TO L'                Or, if line is all spaces, it's NULL
330 IF MID$(A$,X,1)<>" " THEN NOTNULL=1' <== MEANS LINE IS NOT NULL
340 NEXT X
350 IF NOTNULL=0 THEN NULLINE=NULLINE+1:GOTO 270' If null line, then increment
360 '                                     the null line ctr and do the next line.
370 IF NULLINE=1 THEN QUOTE=0:GOTO 560'  1 BLANK LINE preceeds a PARAGRAPH.
380 IF NULLINE>1 THEN QUOTE=0:GOTO 950'  2 BLANK LINES preceed a HEADLINE.
390 P=0'                         If not a special case, then send out
400 SHIFT=0'                     the STANDARD LINE prefix.
410 P=P+1
420 CHAR$=CHR$(STD(P))
430 IF CHAR$=CHR$(0) THEN 460
440 GOSUB 1220
450 GOTO 410
460 FOR P=1 TO L'                Then send out the line,
470 CHAR$=MID$(A$,P,1)
480 GOSUB 1290
490 GOSUB 1220
500 NEXT P
510 CHAR$=CHR$(27)'              Then send out an END-OF-LINE code.
520 GOSUB 1220
530 GOTO 250
540 '
550 '  SPECIAL CASE:  PARAGRAPH
560 P=0'                         First send out START OF PARAGRAPH prefix.
570 SHIFT=0
580 P=P+1
590 CHAR$=CHR$(SOP(P))
600 IF CHAR$=CHR$(0) THEN 630
610 GOSUB 1220
620 GOTO 580
630 FOR P=1 TO L'                THEN SEND OUT THE LINE
640 CHAR$=MID$(A$,P,1)
650 GOSUB 1290
660 GOSUB 1220
670 NEXT P
680 '    Then send out all following lines until a blank line
690 '     is found.  All lines until then are linked together by
700 '     changing the RETURNS to SPACES.
710 IF EOF(1) THEN 1140
720 LINE INPUT #1,A$'            Get the next line.
730 L=LEN(A$)'                   See if it's blank.
740 IF L=0 THEN 900
750 NOTNULL=0
760 FOR X=1 TO L
770 IF MID$(A$,X,1)<>" " THEN NOTNULL=1
780 NEXT X
790 IF NOTNULL=0 THEN 900
800 CHAR$=" "
810 GOSUB 1290
820 GOSUB 1220
830 FOR P=1 TO L
840 CHAR$=MID$(A$,P,1)
850 GOSUB 1290
860 GOSUB 1220
870 NEXT P
880 GOTO 710
890 '  Output a SPACE to link the previous line to the next line.
```

```
900 CHAR$=CHR$(27)
910 GOSUB 1220
920 NULLINE=1
930 GOTO 270
940 '
950 '  SPECIAL CASE: HEADLINE
960 P=0'                    First send out the START OF HEADLINE prefix.
970 SHIFT=0
980 P=P+1
990 CHAR$=CHR$(SOH(P))
1000 IF CHAR$=CHR$(0) THEN 1030
1010 GOSUB 1220
1020 GOTO 980
1030 FOR P=1 TO L'          Then send out the line.
1040 CHAR$=MID$(A$,P,1)
1050 GOSUB 1290
1060 GOSUB 1220
1070 NEXT P
1080 CHAR$=CHR$(27)'        Then send out an END-OF-LINE code.
1090 GOSUB 1220
1100 GOTO 250
1110 '
1120 '  DONE ROUTINE
1130 '    Send out a code to tell the typesetter that we've finished.
1140 CHAR$=CHR$(23)
1150 GOSUB 1220
1160 CHAR$=CHR$(10)
1170 GOSUB 1220
1180 END
1190 '
1200 '  SEND subroutine.
1210 '    Send out the character in CHAR$.
1220 P5=ASC(CHAR$)
1230 IF P5=127 THEN 1940
1240 OUT &H44,P5
1250 RETURN
1260 '
1270 '  TRANSLATE subroutine.
1280 '  Convert CHAR$ to TTS and send out shift code, if necessary.
1290 GOSUB 1320:'  TRANSLATE TO TTS AND GET SHIFT MODE
1300 IF SHIFT<>CSH THEN SHIFT=CSH:TEMP$=CHAR$:CHAR$=CHR$(P1):GOSUB 1220:CHAR$=TEMP$
1310 RETURN
1320 '
1330 '  ASCII TO TTS TRANSLATION ROUTINE.
1340 '    Enter:
1350 '      CHAR$ has ASCII character.
1360 '    Exit:
1370 '      CHAR$ has TTS character.
1380 '      CSH set to 0 if UNSHIFT, 1 if SHIFT, 2 if SUPERSHIFT.
1390 '      P1=TTS shift character.
1400 CHAR=ASC(CHAR$)
1410 IF CHAR>127 THEN CHAR=CHAR-127
1420 C=A(CHAR)
1430 IF C>=192 THEN 1490
1440 IF C>=128 THEN 1540
1450 CSH=0
1460 P1=62
1470 CHAR$=CHR$(C)
1480 RETURN
1490 C=C-192
1500 CSH=2
1510 P1=37
1520 CHAR$=CHR$(C)
1530 RETURN
1540 C=C-128
1550 CSH=1
1560 P1=54
1570 CHAR$=CHR$(C)
1580 RETURN
1590 '
1600 '  DATA for ASCII to TTS translation
1610 '
1620 DATA 0,0,0,0,0,0,0,23,0,0
1630 DATA 0,0,0,0,0,0,0,0,0,0
1640 DATA 0,0,0,0,0,0,0,0,0,0
1650 DATA 0,0,8,135,127,58,7,58,177,17
1660 DATA 163,35,58,58,25,19,57,209,45,59
1670 DATA 39,3,21,33,43,15,13,49,181,53
1680 DATA 0,0,0,173,58,134,178,156,146,130
1690 DATA 154,180,168,140,150,158,164,184,152,176
1700 DATA 172,174,148,138,160,142,188,166,186,170
1710 DATA 162,0,0,0,0,0,17,6,50,28
1720 DATA 18,2,26,52,40,12,22,30,36,56
1730 DATA 24,48,44,46,20,10,32,14,60,38
1740 DATA 58,42,34,0,0,0,0,0
1750 '
1760 '  The appropriate commands for the following prefixes
1770 '    would be inserted in the following DATA statements
1780 '    according to whatever you wanted to do when each was
1790 '    encountered.
1800 '
1810 '  Data for STANDARD LINE prefix.
```

(continued next page)

Operating Systems

Description	Version

These operating systems are available from Lifeboat Associates, except where otherwise mentioned.

Description	Version
CP/M-80 for:	
Apple II w/Microsoft BASIC	2.20B
Datapoint 1550/2150 DD/SS	2.21
Datapoint 1550/2150 DD/DS	2.21
Datapoint 1550/2150 DD/SS w/CYN	2.21
Datapoint 1550/2150 DD/DS w/CYN	2.21
Durango F-85	2.23
Heath/Zenith H89	2.2
iCOM 3812	1.42
iCOM 3712 w/Altair Console	1.42
iCOM 3712 w/IMSAI Console	1.42
iCOM Microfloppy (# 2411)	1.41
iCOM 4511/Pertec D3000 Hard Disk	2.22
Intel MDS Single Density	1.4
Intel MDS Single Density	2.2
Intel MDS 800/230 Double Density	2.2
MITS Altair FD400, 510, 3202 Disk	1.41
MITS Altair FD400, 510, 3202 Disk	2.2
Micropolis Mod I - All Consoles	1.411
Micropolis Mod II - All Consoles	1.411
Micropolis Mod I	2.20B
Micropolis Mod II	2.20B
Compal Micropolis Mod II	1.4
Exidy Sorcerer Micropolis Mod I	1.42
Exidy Sorcerer Micropolis Mod II	1.42
Vector MZ Micropolis Mod II	1.411
Versatile 3B Micropolis Mod I	1.411
Versatile 4 Micropolis Mod II	1.411
Horizon North Star SD	1.41
Mostek MDX STD Bus	2.2
Ohio Scientific C3	2.24
Ohio Scientific C3-B/74	2.24B
Ohio Scientific C3-C'(Prime)/36	2.24B
Ohio Scientific C3-D/10	2.24A
Ohio Scientific C3-C	2.24A
Sol North Star SD	1.41
North Star SD IMSAI SIO Console	1.41
North Star SD MITS SIO Console	1.41
North Star SD	2.23A
North Star DD	1.45
North Star DD/QD	2.23A
Processor Technology Helios II	1.41
by Lifeboat/TRS-80 Mod II	2.25C
by Cybernetics/TRS-80 Mod II	2.25

Hard Disk Modules

Description	Version
Corvus Module	2.1
APPLE-Corvus Module	2.1A
KONAN Phoenix Drive	1.8
Micropolis Microdisk	1.92
Pertec D3000/iCOM 4511	1.6
Tarbell Module	1.5
OSI CD-74 for OSI C3-B	1.2
OSI CD-36 for OSI C3-C'	1.2
SA 1004 for OSI C3-D	1.1
SA 4008 for OSI C3-C	1.3

Listing 6 (continued)

```
1820 DATA
1830 '
1840 '  Data for START OF PARAGRAPH prefix.
1850 DATA
1860 '
1870 '  DATA FOR START OF HEADLINE profix.
1880 DATA
1890 '  END OF DATA
1900 '
1910 '  QUOTE PROCESSING ROUTINE
1920 '     These were added to convert ASCII QUOTE characters to
1930 '     the appropriate opening or closing QUOTE marks in TTS.
1940 IF QUOTE=0 THEN 2000'      If no prev. quote found, then assume OPEN QUOTE.
1950 QUOTE=0'                   Else, assume this is a closing QUOTE.
1960 IF SHIFT<>0 THEN SHIFT=0:CSH=0:P5=62:GOSUB 1240' Tell that it's closing.
1970 P5=17
1980 GOSUB 1240'                TTS has only single quote marks, so send it
1990 GOTO 1240'                 twice to make a double quote mark.
2000 IF SHIFT<>1 THEN SHIFT=1:CSH=1:P5=54:GOSUB 1240' Tell that it's opening.
2010 QUOTE=1
2020 GOTO 1970
2030 END
```

Software Notes

Printer Or Console From BASIC-80

Bob Kowitt

I have been using Microsoft BASIC 5.x since it was first released two years ago. I had been using North Star's excellent BASIC for the three years before but I found the need to change, not only because Microsoft had what I considered to be a much more powerful language interpreter but, in addition, the new BASIC-80 allowed me to save the programs in a protected mode, to protect my programming from the end user.

However, the change forced a sacrifice on me. I had to give up two very convenient operations that were in North Star BASIC:

1) the multiple line user-defined function which operated in a similar manner to the procedures in PL/I and other high level languages, and,
2) the ability to choose at run-time whether to output to my CRT or printer.

I am still waiting for Microsoft to remedy the first loss but, while waiting, I decided to supply the second on my own. I found that BASIC-80 reads the CP/M vector table at load time to find the location of my CRT output routine and the LIST device output routine. The BIOS itself is never modified. Then when the programmer uses the command LPRINT instead of PRINT, BASIC-80 sends the output to the printer and not the CRT terminal.

I then reasoned that if I could redirect the CRT output to the printer by POKEing into the proper location of BASIC-80, I could accomplish my purpose under user control at run time.

Finding the location within BASIC-80 may be done using DDT or SID to look for the location that contains byte two of the console driver location and then a search of all locations found, to find which one is followed by byte 1 of the console driver location in Intel format. (When, oh when, will SID give us a multiple byte search?) After finding the BASIC-80 location, I experimentally POKEd the LST driver location into the CRT driver location and . . . EUREKA!! the printer took off.

This worked beautifully on my computer with BASIC-80 Version 5.0. Unfortunately, when I tried it with BASIC-80 Version 5.1, I realized that Microsoft's modification of BASIC-80 had moved my pointer location. Evidently, finding this pointer had to be done dynamically from the program to be fully transportable.

My original routine to do this was published in *S100 Microsystems* in the Mar/Apr 1981 issue and that's when I woke up to another problem. Actually, I was awakened by several phone calls and letters informing me that it didn't work. I was poking one byte to change the pointer location within the same memory page, since my CP/M BIOS had the console driver and the list driver on the same memory page. When the routine was implemented on a computer where the BIOS crossed a page boundary between the CRT driver and list driver, the computer was down and out for the count.

Modify again . . .

Of course! We need to find two bytes in the BIOS to be sure. Then if we cross a page boundary, we must POKE two bytes into BASIC-80. Since revision, I have transported programs to many computers and have had no problems.

I held off using the BASIC-80 compiler BASCOM until chaining was possible with COMMON variables but when that was available, new modifications in my printer/console selecting routines were necessary. Since, with compiled programs, there is no longer an interpreter pulling in pointers from CP/M, direct POKEs are possible into the jump vector table.

To explain the routines I use (in the order that they should be used):

Line 30 - Determine whether the program is to be used compiled or with the interpreter. (I am working on this to make it dynamic also.)
Line 560 - Search BASIC-80 for the location of the pointers.

TABLE is the location in the BIOS of the jump vector table for warm boot. This is found by reading location 01H and 02H.

PLOC1 & PLOC2 are the location of the pointer to the LST driver in the vector table.

CLOC1 & CLOC2 are the location of the pointer to the CRT driver in the vector table.

C1, C2,P1 & P2 are the pointers found by CLOC1, CLOC2, PLOC1 & PLOC2, that are to be POKEd into BASIC-80.

F1 & F2 are the locations within BASIC-80 that have the pointer to the CRT driver.

Line 380 - I have a form I use to outline any messages. This routine centers the line and outlines it with asterisks (or anything else that I might want).

Line 525 - Determine where you want the output sent. The default output is CRT and after selecting the printer it will be modified with P1 & P2.

Line 1060 - This line proved necessary because if you do a control C during the printout, the printing stops and you are left with all output going to the printer, including the message that you did a control C in a certain line. This can mess up your printing and, in addition, any input through the keyboard is echoed only on the printer, compounding the problem. Line 1060 polls the keyboard after every line and if an "E" or "e" is found, will restore the console, end the printout and branch to a correction if you write one. Put it in wherever you might think you would need it.

Line 620 - this is the meat and potatoes. If you are working in the interpreter, it pokes the proper values into the BASIC-80 pointer location and output goes to the printer or CRT, as selected previously. However, if you are using the program compiled (see compile switch at the top), you will actually poke into the CP/M BIOS and modify the console pointer there to point at the LIST device.

Line 615 - is the recovery line whenever you want CRT display. Go back and forth to provide interactivity in the midst of printing. Be sure you use this when you are finished printing to get back control at the CRT.

```
10      '        PRNCON.BAS          7/1/81
15      '
20      '        set compiler or interpreter=1
30 COMPILER=0 : INTERPRETER=1
40      '        *****************************
45 DEF FNY$(X$)=CHR$(ASC(X$+" ") AND &H5F)    'upper case functioi
50      '
135 GOTO 1000    '    to start of program
140     '
215     '        *****************************
375     '        emphasized centering routine
380 MK$="*"      ' alter design as you wish
385 MS$=MK$+"   "+MS$+"   "+MK$
390 M$=STRING$(LEN(MS$),MK$) :T=(76-LEN(M$))/2 : PRINT TAB(T) M$
395 GOSUB 405    'to normal centering routine
400 M$=STRING$(LEN(MS$),MK$) :T=(76-LEN(M$))/2 : PRINT TAB(T) M$ : RETURN
405 T=(76-LEN(MS$))/2 :PRINT TAB(T) MS$; : RETURN  'Centering routine
410     '        *****************************
415     '        select console or printer
525 MS$="(P)rinter or (C)onsole output ?" :GOSUB 380    ' centering routine
530 INPUT "",OT$
535 OT1=C1 : OT2=C2
540 IF FNY$(OT$)="P" THEN OT1=P1 : OT2=P2
545              RETURN
550     '        *****************************
555     '        search for output byte
560 TABLE=(PEEK(2)*256)+PEEK(1)
565 PLOC1=TABLE+13 : CLOC1=TABLE+10
570 PLOC2=PLOC1+1 : ~LOC2=CLOC1+1
575 P1=PEEK(PLOC1) : C1=PEEK(CLOC1)
580 P2=PEEK(PLOC2) : C2=PEEK(CLOC2)
585 FOR I=16600 TO 18000
590      IF PEEK(I)=C1 AND PEEK(I+1)=C2 THEN 600
595 NEXT
600 F1=I : F2=I+1 : RETURN
605     '        *****************************
610     '        subroutines for output choice
615 WIDTH 79  :IF COMPILER THEN POKE CLOC1,C1 :POKE (CLOC1+1),C2 :
   RETURN : ELSE POKE F1,C1: POKE F2,C2 :RETURN  ' enable console
620 WIDTH 132 :IF COMPILER THEN POKE CLOC1,OT1:POKE (CLOC1+1),OT2 :
   RETURN : ELSE POKE F1,OT1:POKE F2,OT2:RETURN  ' enable selected
999     '        *****************************
1000    '
1003    '      ********     S T A R T     ********
1005    '
1008 PRINT "Enter a test phrase to be printed 10 times   "; :
  LINE INPUT PHRASE$
1010 '
1020 GOSUB 525      'select output mode
1030 GOSUB 620      'poke selected value
1040 FOR K=1 TO 10
1050      PRINT PHRASE$
1060      S$=INKEY$        'never use CTL C -this polls for stop
1070      IF FNY$(S$)="E" THEN GOSUB 615 : END      'returns control to CRT
1080 NEXT
1090 GOSUB 615      'return control to CRT
1100 END
```

Software Notes

A Pseudo-INKEY Function For CBASIC2

Bill Burton

CBASIC is a popular language which has been widely used to create business or commercially oriented application software. One of the most important criteria by which such products are judged is the degree to which they have been 'idiot proofed'. This rather unfriendly sounding term means nothing more than writing programs in such a way that user errors, especially errant keystrokes, do not cause programs to abort unexpectedly. Whereas all programs should include appropriate error detection and recovery procedures, 'idiot proofing' becomes especially crucial when the integrity of a valuable data base is at stake.

Ironically, CBASIC programmers attempting to write serious applications, have faced, and have probably become accustomed to, special problems which result from the way in which CBASIC processes console input. A great many manuals describing packages written in CBASIC include prominent warnings about the possible consequences of entering either control C or control Z. (These will abort program execution without closing open files). In fact, entering either of these control characters is only dangerous if entry has been requested by an INPUT statement. In effect this means that the CBASIC INPUT statement should be avoided whenever possible. Code which allows the user to select a menu option provides an ideal example of when not to use the INPUT statement. The reasons are twofold. First, there is no good reason why a program should be allowed to abort accidentally from a menu and secondly, an extra keystroke (CR) is required to terminate the input. Consider the following menus:

```
ENTER - 1: TO UPDATE A FILE
        2: TO SORT THE FILE
        (OPTIONS 2 - 23)
        24: TO EXIT TO CP/M

ENTER - A: TO UPDATE A FILE
        B: TO SORT THE FILE
        (OPTIONS C - W)
        X: TO EXIT TO CP/M
```

The first of these approaches is adequate only if there will be fewer than ten possible selections (not counting choice 0:). The latter approach allows up to twenty-six selections, any of which can be invoked by a single keystroke.

(continued next page)

For the sake of simplicity let us consider how to process only three choices, A, B and C. The ideal solution is provided by the INKEY$ command of Microsoft'S BASIC-80. (Note: a similar command has been included in CB-80). BASIC-80 code to process three menu options using INKEY$ might appear as follows:

```
100 REM LINE 130 IS ONE LONG LINE
110 PRINT TAB(20) "ENTER OPTION (A,B,C) ";
120 A$=INKEY$
130 WHILE A$<>"A" AND A$<>"a" AND A$<>"B"
    AND A$<>"b" AND A$<>"C" AND A$<>"c"
140 A$=INKEY$
140 WEND
```

The beauty of INKEY$ is that the program will do absolutely nothing until the user has provided an acceptable response. Notice the TAB statement in line 110. Trying to duplicate the effect of the preceding code in CBASIC would be reasonably simple, using the CONCHAR% statement, except in cases where a TAB statement had been used to position the prompt line. CONCHAR% will trap invalid entry; however, when TAB statements are used, improper entry can easily corrupt the placement of the prompt line and the position of the cursor.

The example which follows illustrates one approach to solving this problem in CBASIC. An INKEY like function is simulated which will reject all invalid keystrokes. The positions of the prompt line and cursor remain unchanged with these exceptions: pressing the delete or backspace keys or their control key equivalents will move the cursor backward one character. A linefeed will move the prompt line. A carriage return will cause the prompt line to be redisplayed and may be used to restore the display when any of the other invalid keys (LF, BS, DEL) have been pressed. It is important to note that the console width must be set to zero (infinite width) and that a CONSOLE statement must be issued immediately thereafter. The TAB statement which one would normally use must be replaced by an equivalent defined function. You may wish to experiment and observe the results obtained when these restrictions are not observed.

```
REM     ========================================
REM     HOW TO TEST SINGLE CHAR. ENTRIES IN
REM     CBASIC, WHILE PRESERVING THE SCREEN
REM     ========================================

REM     ========================================
REM     INITIALIZE PSEUDOTAB AS VARIABLE T%
REM     NOTE: TRUE PSEUDOTAB VALUE = T% + 1
REM     IF T% = 19 THEN TRUE TAB VALUE = 20
REM     ========================================

        T% = 19

REM     ========================================
REM     B.8$ = 8 SPACES / B.32$ = 32 SPACES
REM     ========================================

        B.8$  = "        "
        B.32$ = B.8$ + B.8$ + B.8$ + B.8$

REM     ========================================
REM     SET INFINITE CONSOLE WIDTH : 110H=0
REM     CONSOLE STATEMENT BELOW IS REQUIRED
REM     ========================================

        POKE 110H, 0
        CONSOLE

REM     ========================================
REM     DEFINE PSEUDO-TAB AS FNP.TAB$
REM     ========================================

        DEF FNP.TAB$ = LEFT$ (B.32$, T%)

REM     ========================================
REM     REQUEST ENTRY:  'A','B' OR 'C' ONLY
REM     ========================================

        PRINT
        PRINT FNP.TAB$; "ENTER :"
        PRINT
        PRINT FNP.TAB$; "A: TO RETURN TO MENU"
        PRINT FNP.TAB$; "B: TO ENTER NEW DATA"
        PRINT FNP.TAB$; "C: TO EXIT TO SYSTEM"
        PRINT
100     PRINT FNP.TAB$; "ENTER OPTION A,B,C: ";

110     GOSUB 1000

        IF INKEY% = 13 THEN 100

        IF INKEY% < 65 OR INKEY% > 67 THEN \
        GOSUB 1100 : GOTO 110

        ON (INKEY%-64) GOTO 120,121,122

120     CHAIN "MENU.INT"
121     CHAIN "DATA.INT"
122     STOP

REM     ========================================
REM     WHEN ANY LOWER CASE CHARACTERS HAVE
REM     BEEN ENTERED, THE FOLLOWING ROUTINE
REM     FORCES TRANSLATION TO UPPER CASE.
REM     ========================================

1000    INKEY% = CONCHAR%

        IF INKEY% > 64 THEN \
        INKEY% = INKEY% AND 95

        RETURN

REM     ========================================
REM     DO NOT DISPLAY INVALID CHARACTERS
REM     N.B. LINE 1101 IS ALWAYS EXECUTED
REM     ========================================

1100    IF INKEY% < 32 THEN \
        PRINT CHR$ (32); CHR$ (8); CHR$ (32);

1101    PRINT CHR$ (8); CHR$ (32); CHR$ (8);

        RETURN
```

8080 Assembler Programming Tutorial: Other Instructions

Ward Christensen

There are five instructions which didn't fit into the categories previously covered. They are: NOP, HLT, STC, EI, and DI, which will be covered in this section.

NOP

NOP gets its name from No OPeration. The 8080 and Z80 value is simply 0. NOP is useful to simply leave a byte or more of room in a program, so that some other instruction may later be inserted.

Another use of NOP is to "zap" out an instruction. For example, if you suspect a certain subroutine is causing problems, you might overlay the CALL to it, which is three bytes, with three NOPs.

HLT

I briefly mentioned HLT in section five, the movement instructions. If you could use the MOV instruction with an operand of "M,M", it would generate 76 hex. Since such an instruction would be meaningless, the pattern of 76 hex in the 8080 and Z80, stands for HLT: halt the processor.

HLT is not frequently encountered in CP/M programming. There are seldom times that you want to completely stop execution. CP/M itself uses it, for instance to "get your attention" to signal an error when executing the LOAD command. When a problem is encountered in the HEX file, LOAD prints an error message, then issues a HLT. It is necessary to reset your computer to get out of it.

STC

STC sets carry. It is most frequently used in subroutines which want to indicate something to the caller of the subroutine. For example, a subroutine that reads a byte of data from the disk might set carry when it gets an end of file. Normally, ASCII files in CP/M are terminated with an EOF character: 1A hex. However, ED and some other editors, allow an ASCII file to end without any 1A characters. Thus, a program which calls the disk read subroutine, couldn't always check for the 1A, unless the disk read subroutine intentionally faked one up. It could however, do a Jump Carry (JC) to test for EOF. This makes the program shorter, because each call to disk read doesn't have to do a

```
CPI     1AH
JZ      eofroutine
```

to test for end of file: instead it can merely:

```
JC      eofroutine
```

You will see use of this technique in the upcoming programming examples.

INTERRUPTS

An entire tutorial could be dedicated to the subject of interrupt handling in the 8080. It would not be sufficient just to cover the interrupt related instructions, but to go into why they are needed, and how they are used. In fact, virtually all hobbyist microcomputer programming in assembly language can be done without the use of interrupts. For this reason I avoid "confusing the issue" by spending time talking about interrupts, except to go over the two instructions which control them:

EI DI

EI stands for Enable Interrupts. DI stands for Disable Interrupts. If your computer has no device attached to it which is capable of generating interrupts, then you need not even be concerned with these instructions.

When the 8080 is reset, interrupts are automatically disabled. In order to run a device using interrupts, they must be enabled, with an EI instruction. Similarly, after an interrupt occurs, interrupts are automatically disabled. This prevents a second interrupt happening before the program is ready for it. One interrupt routine may issue an EI when it is able to be interrupted, or may wait until it is completely done before issuing an EI. The 8080 specifically ensures that one additional instruction is allowed to be executed following the EI, before interrupts are actually enabled. Thus, an interrupt routine frequently ends with:

```
EI      ;ENABLE
RET     ;THEN RETURN
```

This concludes the 8080 instructions. Next month, I'll discuss some of the pitfalls of 8080 programming, and maybe I can save you time, by keeping you from making the same mistakes I did. The next section will include some useful subroutines, then finally a complete CP/M programming example.

The March issue was placed into the mail on February 25th. 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 April 1982, into the mail around March 26th. We will print each month the date of the previous issue's mailing and would appreciate your help in tracking the deliveries.

Notice

XLT86, A Review And How To Minimize 8080 to 8086 Translation Grief

Kelly Smith

'First there is a mountain, then there is no mountain, then there is...' -Donovan

Digital Research provides the budding assembly language programmer for the new 8086 CPU based systems, an [almost painless] means of converting the 'mountain' of 8080 software to usable 8086 assembly language form . . . but not without some anguish in the process.

It is my hope that you can minimize the time required for conversion of your CP/M-80 based applications, by the insight gained through my own conversion problems and subsequent solutions.

Careful Review of Your 8080 Assembly Language Source Code

First of all, carefully review your source code for the 'undesirable' aspects of the code. This means:

1-Absolutely no self-modifying code can be allowed (as if this doesn't go without saying, but some solutions to problems can be best taken care of by this method)! Why, you ask? Well, the internal architecture of the 8086 has a six byte 'look-ahead' instruction buffer, that can acquire instruction sequences *prior* to its actual execution; meanwhile (six bytes away), perhaps you are thinking that your little 'byte fiddle' will actually occur when in fact, the code has not been modified at all! This can make for some interesting (and hair-pulling) debug, because 'now-you-see it, now-you-don't'! Also, because of this same instruction buffer, you cannot write code that will assemble into (attempted) executable addresses that are within six bytes of the end of a 'segment' address. You have to code some very large programs to do this one, however.

Listing 1

```
;
;
;
        mov     cx,Word Ptr PHYSEC          ;get physical sector number
;
setsec: mov     Byte Ptr FUNC,11            ;BIOS Set Sector Function
        mov     Word Ptr BIOS_DESC,cx       ;pass sector #  to BIOS descriptor
        mov     Word Ptr BIOS_DESC+2,0      ;dummy fill DX descriptor with zero
        mov     dx,(Offset FUNC)            ;point to function parameter block
        mov     cl,50                       ;let CP/M-86 Function 50 do the work
        int     224                         ;this is a BDOS CALL??? Amazing...
;
; more code follows, and finally...
;
        dseg
;
; storage for 5 byte BIOS function descriptor
;
FUNC            rs      1                   ;BIOS Function Code goes here
BIOS_DESC       rs      2                   ;CX data goes here
                rs      2                   ;DX data goes here
;
```

Listing 2

```
FFFF =            true     equ    -1          ;define true
0000 =            false    equ    not true;define false
                  ;
FFFF =            stdcpm   equ    true        ;standard CP/M
0000 =            nstdcpm  equ    false       ;non-standard CP/M
                  ;
                           if     stdcpm
0100                       org    100h
                           endif

                           if     nstdcpm
                           org    4200h
                           endif
                  ;
0100 C32601       begin:   jmp    lagnaf
                  ;
                           if     stdcpm
0103 506C617920            db     'Play your cards right Honey and...$'
                           endif
                  ;
0126 C30000       lagnaf:  jmp    $-$
                  ;
0129                       end
```

2-Conditionals based on the Carry Flag response to the 8080 DAD instruction *may* cause bizarre results because there is no equivalent instruction for DAD in the 8086 CPU – only a simulation of the DAD instruction (and rather lengthy!).

3-Any reference to 'M' (memory pointed to by register pair H and L, for the 8080) takes on a whole new aspect, due to the segmentation addressing used with the 8086. The BX register (you know it as HL) *could* be pointing to the local 'Data Segment' or to any other segment in memory, depending on the prior state of the CS (Code Segment), DS (Data Segment), or ES (Extra Segment) registers. This 'bytes you on the assembly' (heh, heh!) when you're doing otherwise legitimate things like attempting to get data out of CP/M's disk parameter block (of what use to be in only one sixty four kilobyte chunk of 8080 addressable RAM), and the data is actually some place in the high segment 'boonies'! If you aren't careful, your code will be picking up trash data from just about any memory segment but the one you really want! Note, that our old 8080 friend 'M' is actually translated to an 8086 equate as 'Byte Ptr 0[BX]', and is read as "pointing to the byte in RAM by the low byte content of the BX register (BL)". Study 'M' and its relationship to the segment registers closely – but then maybe you have masochistic ideas about using DDT86 . . .

4-Let me guess, the code you want to translate examines address 1 (the old CP/M-80 'Warm Boot' vector) and does direct BIOS I/O by calling this address (with a vector offset) to 'short circuit' some internal features of CP/M-80. Well, forget it! None of our presumed address vectors are accessible by this means when using CP/M-86. Digital Research *did* provide the means to do it, however, with a Direct BIOS Call (Function 50); but if you have this throughout your existing 8080 source code, count on a heavy session with your favorite flavor of editor to set things straight again. In Listing 1 see my code for doing direct BIOS I/O.

5-Conditional assembly parameters have a nasty way of just flat disap-

Listing 3

```
 FFFF                  true    EQU     -1        ;define true
 0000                  false   EQU     not true;define false
                       ;
 FFFF                  stdcpm  EQU     true      ;standard CP/M
 0000                  nstdcpm EQU     false     ;non-standard CP/M
                       ;
                               ORG     100h

                       ;
)100 EB 23             begin:  JMPS    lagnaf
                       ;
)102 50 6C 61 79 20 79         DB      'Play your cards right Honey and...$'
     6F 75 72 20 63 61
     72 64 73 20 72 69
     67 68 74 20 48 6F
     6E 65 79 20 61 6E
     64 2E 2E 2E 24
                       ;
0125 B1 00             lagnaf: MOV     CL,0
0127 B2 00                     MOV     DL,0
0129 CD E0                     INT     224
                       ;
                               END
```

Listing 4

```
                ;
0100 44                  MOV     B,H
0101 4D                  MOV     C,L
                ;
                ;
                ; Subtract DE from HL
                ;
0102 7D         SUBDE:   MOV     A,L
0103 93                  SUB     E
0104 6F                  MOV     L,A
0105 7C                  MOV     A,H
0106 9A                  SBB     D
0107 67                  MOV     H,A
0108 C9                  RET
                ;
                ; Negate HL
                ;
0109 7D         NEG:     MOV     A,L
010A 2F                  CMA
010B 6F                  MOV     L,A
010C 7C                  MOV     A,H
010D 2F                  CMA
010E 67                  MOV     H,A
010F 23                  INX     H
0110 C9                  RET
                ;
                ;
                ; Shift HL right one place
                ;
0111 B7         ROTRHL:  ORA     A
0112 7C                  MOV     A,H
0113 1F                  RAR
0114 67                  MOV     H,A
0115 7D                  MOV     A,L
0116 1F                  RAR
0117 6F                  MOV     L,A
0118 C9                  RET
                ;
```

(continued next page)

pearing from the newly translated source code. I mean *gone*!

The latest public domain 'sorted directory display' program SD-42, is 'chock-full' of TRUE/FALSE switches that allow you to configure it to do everything but pour coffee! Just as a quick example, look at Listing 2, a portion of assembled 8080 source code. That little gem was derived from a major software effort named 'Programming Porno' by Mike Karas (must give credit where credit is due), and essentially demonstrates what is to follow when given a touch by XLT86, shown in Listing 3.

Well, not exactly what we had in mind now, was it! And XLT86 takes other 'conceptual liberties', as with that routine labeled 'lagnaf:' . . . but then again, it looks as if it's a crude setup waiting for an instruction modify anyway!

Improving Coding Efficiency

XLT86 will make every attempt to translate your original source code 'to-the-letter', where possible, sometimes with inefficient results. Take for example, the 8080 source in Listing 4, and the translation in Listing 5. Well, sad but true . . . it took twenty-five bytes of memory for the 8080 code generation, but forty eight for the translation to 8086 . . . gads! So how can we improve upon this? Look at Listing 6, as we now take advantage of the 8086 instruction set, by 'hand coding' for efficiency.

Well, that's twelve bytes of code. If we got rid of the returns (stupid to make them subroutines!) we could save three more bytes, decreasing the amount of RAM required, and speeding up the execution time. Your translation will be 'riddled' with garbage code similar to this, if you don't 'hand code' portions like this example.

Divide and Conquer

An additional aspect to consider before attempting translation to 8086 code is the code size that is to be translated. You can expect that if your 8080 assembler is going to generate anything more than six kilobytes of code for an eventual '.COM' file, that you will

Listing 5

```
0100 8A EF                      MOV     CH,BH
0102 8A CB                      MOV     CL,BL
                        ;
                        ;
                        ; Subtract DE from HL
                        ;
0104 8A C3              SUBDE:  MOV     AL,BL
0106 2A C2                      SUB     AL,DL
0108 8A D8                      MOV     BL,AL
010A 8A C7                      MOV     AL,BH
010C 1A C6                      SBB     AL,DH
010E 8A F8                      MOV     BH,AL
0110 C3                         RET
                        ;
                        ; Negate HL
                        ;
0111 8A C3              NEG:    MOV     AL,BL
0113 F6 D0                      NOT     AL
0115 8A D8                      MOV     BL,AL
0117 8A C7                      MOV     AL,BH
0119 F6 D0                      NOT     AL
011B 8A F8                      MOV     BH,AL
011D 9F                         LAHF
011E 43                         INC     BX
011F 9E                         SAHF
0120 C3                         RET
                        ;
                        ;
                        ; Shift HL right one place
                        ;
0121 0A C0              ROTRHL: OR      AL,AL
0123 8A C7                      MOV     AL,BH
0125 D0 D8                      RCR     AL,1
0127 8A F8                      MOV     BH,AL
0129 8A C3                      MOV     AL,BL
012B D0 D8                      RCR     AL,1
012D 8A D8                      MOV     BL,AL
012F C3                         RET
                        ;
```

Listing 6

```
                        ;
0100 8B CB                      mov     cx,bx
                        ;
                        ;
                        ; Subtract DX from BX
                        ;
0102 1B DA              SUBDE:  sbb     bx,dx
0104 C3                         RET
                        ;
                        ; Negate BX
                        ;
0105 F7 D3              NEG:    not     bx
0107 43                         inc     bx
0108 C3                         RET
                        ;
                        ;
                        ; Shift BX right one place
                        ;
0109 D1 EB              ROTRHL: shr     bx,1
010B C3                         RET
                        ;
```

have to partition your 8080 source code into [hopefully] logical 'chunks' that are targeted for less than six kilobytes. XLT86 will not translate even moderately large source files! What does this mean to you from a conversion standpoint? Back to your editor, only this time you must equate 'externally referenced' subroutine labels where your partitioned source code JMP's, CALL's or references RAM (LDA, STA, LHLD, SHLD, STAX, etc.) outside of each partitioned 'module'. Once partitioned and translated, then you can either use the ASM86 pseudo-op 'INCLUDE', or concatenate the whole mess back together again with PIP, remove all the partition references, and then assemble normally. 'INCLUDE' is the nice way to go however, as you can edit small portions of your new 8086 converted program as you confirm its proper operation. Yes, *confirm* its operation with DDT86 even though it appears to be running properly from what you see happening on the console. Any weird file I/O should be double-checked with Ward Christensen's DU program. I should know, as my 'finally debugged' version of FINDBAD.A86 went through 'all of the motions' of working correctly; the only problem was, that as each new file extent opened up for the [UNUSED].BAD file bad sector group allocations, there were no group numbers 'pumped' into the directory. (Maybe that's why it's called a 'Boob Tube'. Do not trust what you see happening on that tube in front of you!)

Conclusion

Digital Research's XLT86 appears to do about 80% of the job, with the remaining 20% left for you. Not bad I would say, considering what you would go through if the work had to be done 'from scratch'! The conversion of FINDBAD Version 5.2 (Universal Version) took me about forty hours, and now happily checks out both double density microfloppies and a 'Winchester' disk drive, both without knowing the 'personalities' of either. SD Version 4.1 took half that time from the experience gained in translating FINDBAD, by getting the 'funnies' out prior to translating.

The key to effective translation time is a careful review of your original '.ASM' file; time is better spent there than on the agonizing task of, re-edit, re-assemble, debug, re-edit, . . .

Software Notes

Macros of the Month

Edited by Michael Olfe

The reader response to our squealings for submissions has been overwhelming, mind-boggling, and totally *ne plus ultra*. The distinguished panel of macro judges was stymied, split and stumped. Only a tie could justly represent the quality of these submissions. So thanks for the submissions, gentle readers, keep up the flood of macros, and enjoy those below.

Mike Olfe

```
;PMATE Macro:  JUSTIFY and UNDO
;
;Author:   David Flory
;Written: Feb 23,1982
;Version: 1.Ø
;
;Both of these macros must be used in format mode to work.
;
;Macro to right justify text written in format mode.  The macro searches for
;lines that don't end in a <cr> (char 13) and need padding.  In format mode,
;PMATE breaks lines at spaces or hyphens.  PMATE then converts the space just
;before a word that would be in the margin column to a char 224 or 225
;(auto-indent) while it converts a hyphen to char 226 or 227 (auto-indent). A
;converted space can be in the margin column while a converted hyphen cannot.
;Consequently, lines ending in a hyphen require one less space of padding.
;@T/226 returns 1 for a converted hyphen and Ø for a converted space.
;
a                                 ;start at the beginning, the macro works down
[                                 ;Main loop
    [                             ;BEGIN Search loop
        @t=Ø{%}                   ;IF eof THEN exit Macro
```

(continued next page)

```
        l-m                            ;find the end of the current line
        @t=13{m^}                      ;IF <cr>THEN next line and continue Search
        @t/226+@x<@w{_}{m^}            ;IF it needs padding THEN Pad
      ]                                ;END Search loop
      eØs ^N $                         ;look to see if it can be padded
      @e-{l^}{l-m}                    ;IF not, continue Search ELSE find end of line
      [                                ;BEGIN Pad loop
        @w-@x-(@t/226)                 ;FOR number of spaces needed DO
        {
           eØs ^N $                    ;(find a place to put a space
           @e{l-m^}{32i}               ;IF can't THEN repeat Pad ELSE insert a space
        }                              ;)
      -l]                              ;END Pad
  l]                                   ;next line REPEAT Main loop
  ;
  ;
  ;
  ;Macro to restore right justified text to free format mode.  Blocks of spaces
  ;(presumably padded) in free format lines are stripped to single spaces
  ;leaving the obligatory double space after each period.
  ;
  z                                    ;start at the end, the macro works up
  [                                    ;Main loop
     [                                 ;BEGIN Search loop
        Øl-m                           ;backup to end of prior line
        @c=Ø{%}                        ;IF beginning of buffer THEN quit Macro
        @t=13{^}{_}                    ;IF<cr> THEN continue Search ELSE Strip
     ]                                 ;END Search
     [                                 ;REPEAT Strip loop
        eØs  $                         ;find a block of spaces to strip
        @e{_}                          ;IF can't THEN Search
        mt                             ;tag the end of the block
        eØs^N  $                       ;find its start
        @e{_}                          ;IF can't THEN Search
        @t=".{m}                       ;give periods their due
        m#d                            ;leave one space and delete the rest
        -m                             ;move onto space
     ]                                 ;UNTIL can't strip anymore
  ]                                    ;END Main loop
```

```
;    SPLITMOD. MAC /L66          SPLIT SCREEN EDIT MACRO PMATE
;    BY JOHN KANNAR 2/20/82
;
;         SPLIT SCREEN EDIT MACRO FOR PMATE
;
;         DISPLAYS CONTENTS OF THE TEXT BUFFER AND BUFFER 9 IN A SPLIT SCREEN
;         MODE; WITH 9 LINES OF THE TEXT BUFFER ABOVE AND 9 LINES OF BUFFER 9
;         BELOW; SEPARATED BY A DOTTED LINE.
;
;         IT USES BUFFER Ø AS A WORK AND DISPLAY AREA FOR THE SPLIT SCREEN.
;
;         IT USES BUFFER 1 AS A DISPLAY AREA FOR "HELP" INFORMATION.
;
;         EDITING OF EITHER "BUFFER" IS POSSIBLE BY ISSUING A "#" COMMAND.
;         THEN TRACE MODE OR PMATE IS ENTERED,IN WHICH EITHER
;         INSTANT COMMANDS MAY BE ISSUED OR INSERT MODE CAN BE ENTERED. JUST
;          USE NORMAL INSTANT COMMANDS TO MODIFY THE WORK AREA ASSOCIATED WITH
;          EITHER BUFFER: ABOVE THE DOTTED LINE FOR THE TEXT BUFFER OR BELOW THE
;          DOTTED LINE FOR BUFFER 9.  CAUTION: DO NOT MODIFY THE DOTTED LINE!
;          WHEN EDITING OF EITHER OR BOTH BUFFERS IS COMPLETED, IT IS NECESSARY
;          TO RETURN TO COMMAND MODE (IF INSERT MODE WAS ENTERED) AND TO
;          RETURN THE CURSOR TO THE "HOME" POSITION: THE "+" IN THE FIRST
;          POSITION OF THE DOTTED LINE WHICH SEPARATES THE 2 WORK AREAS FOR
;          THE TEXT BUFFER AND BUFFER 9.  WHEN THE "ESC" KEY IS THEN PRESSED,
;          THE ACTUAL BUFFERS ARE UPDATED AND CONTROL IS RETURNED TO THE SPLIT
;          SCREEN COMPARE MODE.
;
;
GWARNING:  THE SPLIT SCREEN EDIT ERASES ANY PRIOR CONTENTS OF BUFFERS Ø AND 1.
           IT COMPARES THE TEXT BUFFER (ABOVE) TO BUFFER 9 (BELOW).
           QUIT OR CONTINUE ( Q OR C)?$
@K=81[%]                   ; QUIT IF "Q" IS INPUT
B1K                        ; CLEAR BUFFER 1
B1E                        ;  AND READ HELP FILE
XISPLITMOD.HLP$            ;  INTO IT
BK                         ; CLEAR BUFFER Ø
ØYØ                        ; INITIALIZE LINE POINTER FOR TEXT BUFFER
ØV1                        ; INITIALIZE LINE POINTER FOR BUFFER 9
BØE                        ; SET BUFFER Ø AS THE CURRENT EDIT BUFFER
:A Z @LV9 A @9K            ; ERASE ALL PRIOR CONTENTS OF BUFFER Ø(THE WORK AREA)
BTEA @ØL                   ; POINT TO THE "CURRENT" LINE IN THE TEXT BUFFER
9BØD                       ;  AND MOVE 9 LINES TO THE WORK AREA
BØE                        ; NOW DISPLAY DOTTED LINE IN THE WORK AREA
I+.................................................................................
$                          ;  TO SEPARATE THE TEXT BUFFER DISPLAY FROM THE
                           ;  BUFFER 9 DISPLAY (NOTE: THE + MARKS THE HOME
                           ;  POSITION)
B9EA @1L                   ; POINT TO THE "CURRENT" LINE IN BUFFER 9
9BØD                       ;  AND MOVE 9 LINES TO THE WORK AREA
BØE                        ; SET THE CURSOR AT THE HOME POSITION ON THE DOTTED
-1ØL T                     ;  LINE AND DISPLAY THE PROMPTING MESSAGE
                           ;
G                          SPLIT SCREEN EDIT
              ENTER THE DESIRED COMMAND (OR H FOR HELP)$
                           ;
@K=12[BTE % ]              ; <CR> EXITS FROM COMPARE
@K=82[ -1VA1 ]             ; "R" BACK 1 LINE IN BUFFER 9
@K=67[ +1VA1 ]             ; "C" FORWARD 1 LINE IN BUFFER 9
@K=69[ -8VA1 ]             ; "E" BACK 8 LINES IN BUFFER 9
@K-88[ +8VA1 ]             ; "X" FORWARD 8 LINES IN BUFFER 9
@K=89[ -1VA1 -1VAØ]        ; "Y" BACK 1 LINE IN BOTH BUFFERS
@K=66[ +1VA1 +1VAØ]        ; "B" FORWARD I LINE IN BOTH BUFFERS
@K=84[ -8VA1 -8VAØ]        ; "T" BACK 8 LINES IN BOTH BUFFERS
@K=86[ +8VA1 +8VAØ]        ; "V" FORWARD 8 LINES IN BOTH BUFFERS
```

```
    SPLITMOD. MAC /L66        SPLIT SCREEN EDIT MACRO FOR PMATE

@K=73[ -1VAØ ]             ; "I" BACK 1 LINE IN THE TEXT BUFFER
@K=77[ +1VAØ ]             ; "M" FORWARD 1 LINE IN THE TEXT BUFFER
@K=85[ -8VAØ ]             ; "U" BACK 8 LINES IN THE TEXT BUFFER
@K=78[ +8VAØ ]             ; "N" FORWARD 8 LINES IN THE TEXT BUFFER
@K=65[ ØV1 ØVØ ]           ; "A" BACK TO START OF BOTH BUFFERS
```

(continued next page)

```
@K=9Ø[BTE Z @LVØ -9VAØ B9E Z @LV1 -9VA1 BØE]
                            ; "Z" FORWARD TO END OF BOTH BUFFERS
@K=72[B1E G                 PRESS ANY KEY TO RETURN TO SPLIT SCREEN$ BØE]
                            ; "H" DISPLAYS THE "HELP" INFORMATION
@K<35 JA                    ; FOR ANY OTHER CHARACTER EXCEPT "#" UPDATE WORK AREA
@K>35JA                     ; "#" ENTERS THE SPECIAL SPLIT SREEN EDIT MODE,BY
?                           ; USING THE TRACE FUNCTION OF MATE AND PMATE.  NOTE
                            ; ALSO THE PROMPTING MESSAGE WHICH WILL APPEAR IN
                            ; THE COMMAND AREA OF THE SCREEN DURING TRACE MODE:
;
;   INSTANT COMMANDS AND "INSERT" MODE MAY BE USED TO MODIFY EITHER BUFFER:
;    THEN RETURN CURSOR TO THE HOME (+) POSITION AND TO "COMMAND" MODE
;    AND ONLY THEN PRESS THE ESC KEY TO CONTINUE
;
                           ;UPON RETURN FROM TRACE MODE (OPERATOR KEYS "ESC"KEY)
BTE A @ØL 9K               ;  ERASE ORIGINAL 9 LINES IN THE TEXT BUFFER
BØ3 @LV9 A @9BTD           ;  AND INSERT THE UPDATED LINES FROM THE WORK AREA
B93 A @1L 9K               ;  AND DO THE SAME THING FOR THE BUFFER 9 UPDATES
BØE Z (@L-@9-1)V8 (-@8)V7 @7L @8B9D
@7L                        ;  FINALLY, RESTORE THE CURSOR TO THE HOME POSITION
JA                         ;  AND UPDATE THE WORK DISPLAY FOR COMPARE MODE
```

~~Software Notes~~

A Patch For WordStar On The Superbrain

Reported by Todd Katz and John Leroy

One of the nicest features of WordStar is its comprehensive INSTALL program, which allows the user to tailor the word processing package to both his terminal and printer. But one of the bad things about this feature is that it doesn't always seem to work quite right; this is the case with the Superbrain.

But 'Brain users need not despair. Where there's a problem there is [sometimes] a patch. In this case the user should "install" the program, selecting the printer and communications protocol after selecting the "+" terminal choice that is allocated to the Superbrain.

Then the question is asked: are modifications to Superbrain now complete? The answer is **NO**!

At this point the user is introduced to the WS patch program which asks the location in operating system to be changed. You respond with the location. The program confirms your choice and tells you the single hex value in that particular location.

Here is a typical session:

LOCATION TO BE CHANGED (0-END):
User answers: **264**
The program responds:
ADDRESS: 0264H OLD VALUE: C3 NEW VALUE:
Naturally you fill in the blank.

If you don't fill in the blank the value will be unchanged and the program will tell you the content of the next byte of operating system. If you answer with a 0 you will be given an opportunity to end the session.

As far as the Superbrain is concerned there is tailoring to do in three locations:

Location #1		Location #2	
		02A4	C3H
		etc.	E0H
0264	C3H		02H
0265	10H		C3H
0266	03H		F4H
			02H

Location #3 for the Superbrain DOS 3.0

Beginning at location 2E0 insert the following order of changes in hex:

CD,F4,02,3D,CD,F5,02,21,52,E2,3E,33,BE,C0,3E,FF, 32,18,E5,C9,AF,21,34,E4,06,18,77,23,05,C2,FA,02, 21,00,00,22,00,E4,22,14,E4,22,16,E4,22,12,E4,C9,EB, C3,0C,03

Here's the sequence for the Superbrain 3.1 DOS

CD,F4,02,3D,CD,F5,02,21,52,E2,3E,33,BE,00,3E,FF, 32,1D,E7,C9,AF,21,34,E4,06,18,77,23,05,C2,FA,02, 21,00,00,22,00,E4,22,14,E4,22,16,E4,22,12,E4,C9,EB, C3,0C,03

And here's the sequence for the Superbrain 3.3 DOS

CD,F4,02,3D,CD,F5,02,21,52,E2,3E,33,BE,00,3E,FF, 32,20,E7,C9,AF,21,34,E4,06,18,77,23,05,C2,FA,02, 21,00,00,22,00,E4,22,14,E4,22,16,E4,2212,E4,C9,EB, C3,0C,03

When you have finished, input a "0" (but not the quotes) and confirm the terminal and printer selections. At this point WS should be installed and you can forget about the INSTALL and WSU ".com" files.

One final note: this does not assume that some of the fancy operating systems being used with the 'Brain will work with this sequence. If all else fails try some of the other terminal choices. The IBM 3101 works rather well on some models.

Volume 79 and Comments

CP/M Users Group

Volume 79

DESCRIPTION: MODEM programs for PMMI, SMARTMODEM, Serial I/O.
Latest versions as of March 1, 1982.
Digital Research "MAC" macro assembler required.

NUMBER	SIZE	NAME	COMMENTS
		CATALOG.079	CONTENTS OF CP/M VOL. 079
			FOR THE Potomac Micro Magic Inc (PMMI) MM-103 MODEM (also serial I/O):
079.1	16K	MODEM7.DOC	Documentation for MODEM 7 and higher. Originally from CP/M UG Volume 47.
079.2	14K	MODEM7.LIB	MACLIB file for use with MODEM741.ASM.
079.3	66K	MODEM741.ASM	Assembler source for MODEM version 7.41.
079.4	10K	MODEM741.COM	Object code of above program.
079.5	4K	MODEM741.SET	Patching instructions for MODEM741.COM.
079.6	1K	MODEM7X.BUG	Bug report.
			FOR THE DC HAYES SMARTMODEM:
079.7	85K	SMODEM37.ASM	Assembler source for SMODEM version 3.7
079.8	26K	SMODEM37.DOC	Documentation for the above program.

Comments

The new CPMUG volume features the latest versions of two modem programs, with significant enhancements. The summary offered here is skeletal, but the documentation on Volume 79 itself is quite exhaustive.

MODEM7 uses the file transfer routines written by Ward Christensen in his CP/M file transfer program and is compatible with his program in single file transfer mode. Multi-file transfers are only possible between two systems running MODEM7.

This program has three functions:

1 - **Communications.** The program may emulate a terminal or echo data back to sender (act as a computer). Only one computer may be in the echo mode at one time. It's useful if you wish to communicate with somebody running the terminal portion of the program.

2 - **Program transfer.** While in Terminal mode, you can go into File Transfer mode. This will allow you to send the contents of an ASCII file over the modem. This routine does no error checking and there are no protocols specified between this program and the receiving computer other than that it should be ready to receive data via the modem. Using the a secondary option, more than one file and ambiguous filenames may be transferred.

3 - **Modem control (for PMMI Modem).** These commands let you set the baud rate, establish ANSWER or ORIGINATE modes, dial a phone number, and perform other necessary functions.

SMODEM37 was originally written by Ward Christensen, later updated by Jim Mills and Mark Zeiger. It is for 8080 or Z80 CPU's, CP/M-80 2.X, and an external Hayes' SmartModem. This version 3.7 includes significant enhancements to earlier versions. It has some special features, such as "autodial", which allows you to automatically dial numbers from several directories.

Bit Manipulation In PL/I-80

Michael J. Karas

The PL/I-80 programming language provides the programmer with a powerful high level language environment in which to develop sophisticated applications software for virtually any task that small computers can accommodate. Through practical experience I have found that one of the many data types available within the language is especially useful for data handling in controller or utility environments. The "BIT" data type allows a PL/I-80 program variable to assume the attributes of data in the lowest level format available within a computer and its associated memory system. This technical piece will describe the PL/I-80 bit variable data types in Part 1, showing examples of one program using them.

Bit data is a declared variable, consisting of a string of binary digits (ones and zeros), treated as a string. In other words, the PL/I-80 bit string is handled in a manner similar to the conventional programming language character string. The difference is that each bit string digit can only assume one of two values, binary one or binary zero. In addition each bit string digit only uses one bit of memory for storage. Bit strings in PL/I-80 can have a length from one (1) to sixteen (16) digits. To establish a variable name as a bit string it is declared as follows:

```
DCL BITSTR BIT(12);
```

This makes "BITSTR" a variable that is a string of binary digits 12 long. Bit string variables are stored in memory according to the following rules for PL/I-80. If the declared length for the string is from 1 to 8 digits, then one byte of storage is used for the variable. If the length is from 9 to 16 bits, then two bytes are allocated for the variable. Bit strings are left adjusted toward the left within the memory bytes allocated if the string length is less than a full byte (8 bits) or less than a full word (more than 9 but less than 16 bits). The bit position numbering for a bit string starts at 1 and continues up to a number equal to the declared length of the string. The following gives a programmer view of a BIT(9) string:

Bit Number:	1	2	3	4	5	6	7	8	9
	0	1	1	0	0	0	0	1	1

Lastly, for reference, it is important to note that bit strings of two bytes in size are stored in computer memory with the lower addressed memory byte corresponding to the higher numbered string bit positions. The above example BIT(9) string is stored in memory as follows:

	7	6	5	4	3	2	1	0 ← Hardware memory byte bit number
Memory Address	1	0	0	0	0	0	0	0
Memory Address+1	0	1	1	0	0	0	0	1

Note how the string shorter than 16 bits is padded on the right with binary zero digits to fill out the two byte memory storage allocation. Note as stated above that the position number nine is stored in the lower numbered byte of memory.

Bit string constants are denoted in the PL/I-80 programming language as quoted strings of ones and zeros. The above example is written in PL/I syntax as like:

```
'011000011'B
```

where B denotes that the string is a bit string as opposed to a character string. An assignment statement can set a variable to a bit string convenience simply as:

```
BITSTR='011000011'B;
```

The PL/I-80 compiler permits a certain degree of notational convenience for the expression of bit string constants. Alternate bases other than binary may be used. The normal letter B is followed by a digit from 1 to 4 to achieve data expression in base 2 (B or B1 binary format), base 4 (B1 format), base 8 (B3 octal format), or base 16 (B4 hexadecimal format). The characters or digits used in the expression of the string value must be valid for the base specified.

Base Format	Allowable digit values
B or B1	0 and 1
B2	0,1,2 and 3
B3	0 to 7
B4	0 to 9, and A to F

The following chart shows several examples of the alternate base notation and gives the equivalent "normal" base representation.

'1010'B1 is the same as '1010'B
'1010'B2 is the same as '01000100'B
'016'B3 is the same as '000001110'B
'101'B4 is the same as '000100000001'B
'A5'B4 is the same as '10100101'B

The short PL/I-80 program in Listing 1 presents an interesting example of simple bit manipulation. The function provides no real productive data processing but allows description of a number of bit functions.

Execution of the program in Listing 1 prints out the following text upon the CRT console device. Detailed explanation of program operation follows.

```
HEXBIT (1) 1010
HEXBIT (2) 1011
HEXBIT (3) 0100
HEXBIT (4) 0001
1011000010100000
0001000001000000
1011101010100000
FINAL 1011101000010100
```

The initial memory space allocation for the above program example for the HEXBIT array consumes four bytes of storage, one byte for each BIT(4) element. Remember that each time an individual bit string is less than 8 bits, a full byte is used. The memory image for HEXBIT after the four assignment statements place values in each array element is:

	7 6 5 4 3 2 1 0	← Hardware memory byte bit number
Memory Address	1 0 1 0 0 0 0 0	HEXBIT(1)
Memory Address+1	1 0 1 1 0 0 0 0	HEXBIT (2)
Memory Address+2	0 1 0 0 0 0 0 0	HEXBIT(3)
Memory Address+3	0 0 0 1 0 0 0 0	HEXBIT(4)

The two pointer assignment statements that follow set memory pointer variables "P" and "Q" to Memory Address and Memory Address+2 respectively. Since declared variables BITA and BITB are based variables, their location in memory is dependent upon the values of pointers P, and Q respectively. This produces the effect of overlaying the storage allocations for BITA and BITB variables directly over the memory where the HEXBIT array is stored. The overlay mechanism allows the programmer to manipulate storage in alternate ways depending upon the declared type for the variable name used to reference the overlay data area. The above memory map becomes as follows for reference to BITA and BITB variable names.

	7 6 5 4 3 2 1 0	← Hardware memory byte bit number
Memory Address	1 0 1 0 0 0 0 0	BITA POSITIONS 9-16
Memory Address+1	1 0 1 1 0 0 0 0	BITA POSITIONS 1-8
Memory Address+2	0 1 0 0 0 0 0 0	BITB POSITIONS 9-16
Memory Address+3	0 0 0 1 0 0 0 0	BITB POSITIONS 1-8

The statement:

```
    put edit(bita,bitb)(col(1),b,col(1),b);
```

results in binary base console printout of the first two printed lines of ones and zeros shown above.

The memory map changes for each additional assignment statement. The function "SUBSTR" referenced here means

```
    substr(bita,5,4)
```

using bit string BITA starting at bit position 5 for 4 bits of length.

The function may be used on either side of an assignment statement. On the left it means to replace the correspondingly designated bit positions with the expression on the right. The SUBSTR function on the right side on an assignment statement indicates to return the bit string value indicated by the starting position and length. The SUBSTR function is generally used within the language syntax to move around "pieces and parts" of bit strings. Note that it is the only real way to dissect a bit string in SUBSET G type PL/I like the version being described here.

After the first SUBSTR statement the memory map for BITA and BITB becomes:

	7 6 5 4 3 2 1 0	← Hardware memory byte bit number
Memory Address	1 0 1 0 0 0 0 0	BITA POSITIONS 9-16
Memory Address+1	1 0 1 1 1 0 1 0	BITA POSITIONS 1-8
Memory Address+2	0 1 0 0 0 0 0 0	BITB POSITIONS 9-16
Memory Address+3	0 0 0 1 0 0 0 0	BITB POSITIONS 1-8

The final version of the memory image that is printed following the text string 'FINAL' looks like this:

	7 6 5 4 3 2 1 0	← Hardware memory byte bit number
Memory Address	0 0 0 1 0 1 0 0	BITA POSITIONS 9-16
Memory Address+1	1 0 1 1 1 0 1 0	BITA POSITIONS 1-8
Memory Address+2	0 1 0 0 0 0 0 0	BITB POSITIONS 9-16
Memory Address+3	0 0 0 1 0 0 0 0	BITB POSITIONS 1-8

So much said about that example. For newcomers to PL/I-80 that small program really packs a lot in a few statements.

A short discussion is pertinent, concerning the means available in the PL/I-80 language for printing out bit strings with the PUT EDIT statement.

If BITA='1011101000010100'B as shown in the above example then the following PUT statements will have the results shown:

```
1) put edit(bita)(col(1),b);
/* b format floats to string length */
        prints → 1011101000010100

2) put edit(bita)(col(1),b(16));
/* b(16) specifies binary field of 16 print positions. */
        prints → 1011101000010100

3) put edit(bita)(col(1),b4(4));
/* b4(4) specifies hex format printing with 4 print positions */
        prints → BA14
```

Additional print formatting options include B2(n) for n print positions of base 4 print format and B3(n) for n fields of octal format printing. This feature means that for programmers us-

(continued next page)

ing PL/I there is usually no need to convert internal bit strings to ASCII format representation because the PUT EDIT part of PL/I will do it for you.

Next month I will present another example, and then finally present an enhancement package of assembly language routines that give more bit processing power to the basic language capabilities.

Listing 1

```
bits:   proc options(main);
        dcl
                hexbit(4)           bit(4),
                bita                bit(16) based(p),
                bitb                bit(16) based(q),
                (p,q)               pointer,
                i                   bin fixed(15);

   /* assign values to elements of hexbit array */

        hexbit(1)='1010'b;
        hexbit(2)='1011'b;
        hexbit(3)='0100'b;
        hexbit(4)='0001'b;

   /* fix pointers to base storage allocation for
      bita over hexBit(1) and hexbit(2) and also
      put bitb storage allocation over
      hexbit(3) and hexbit(4). */

        p=addr(hexbit(1));
        q=addr(hexbit(3));

   /* print out the contents of hexbit array */

        do i=1 to 4;
        put edit('HEXBIT (',i,')',hexbit(i))
                        (col(1),a,f(1),a,x(1),b);
        end;

  /* output bit patterns of bita and bitb
     overlay strings */

        put edit(bita,bitb) (col(1),b,col(1),b);

  /* move the four bit(4) elements of the array
     into a packed format within
     the bita variable */

        substr(bita,5,4)=substr(bita,9,4);
        put edit(bita)(col(1),b);
        substr(bita,9,4)=substr(bitb,1,4);
        substr(bita,13,4)=substr(bitb,9,4);
        put edit('FINAL',bita)(col(1),a,x(1),b);

        end bits;
```

Product Status Reports

New Versions

CBS
Version 1.33

This new version contains modifications required by a new companion product, THE FORMULA. The FILES file is no longer required in this version; related information is now stored in the actual file.

Microsoft COBOL
Version 4.6

With this new release will be a new version of L80; called LD80 (L80 revision 3.54), it uses disk space rather than the memory to build the program image. This should ameliorate the size limitations which occurred with L80.

In addition, this update includes the following enhancements:

1-Runtime routines which generated INDEXed files have been revised and generate files with a new format; this feature is designed to speed the processing time required for file organization.
2-The multiplication algorithm has been sped up.
3-The runtime library has been made into a common runtime system, to speed up linking and to allow larger programs to be linked. Program chaining should improve and programs should take up less disk space.
4-An interactive symbolic debugger called DEBUG.REL has been added.
5-Conditional statements (and IF statements) may now reside within IF statements.
6-When an OPEN EXTEND is performed on a non-existent SEQUENTIAL or LINE SEQUENTIAL file, COBOL creates the file, rather than reporting an error.
7-A RELATIVE KEY may now be a numeric field with USAGE DISPLAY.
8-A RELATIVE or INDEXED organization file detected as possibly damaged may be updated, if a file status item is defined. The file status item is set to 91 upon opening; if no file status item is defined, a DECLARATIVES section will be performed, or a runtime error will be reported.
9-The absence of a DELIMITED BY clause in an UNSTRING statement is now diagnosed.

10-The presence of an OPEN EXTEND statement for a file with RELATIVE or INDEXED organization is now diagnosed.
11-The compiler diagnoses the presence of a READ statement in a format incompatible with its ACCESS MODE.
12-The definition of file status items as two-byte alphanumeric fields is now enforced.
13-An AFTER (BEFORE) ADVANCING clause in a WRITE statement to a disk file is now diagnosed.
14-Relocatable object files are now smaller, because the compile no longer emits temporary global symbols to them.
15-New entry points have been added to CRT drivers.
16-COBOL syntax for the DIVIDE statement has been revised to reflect ANSI standards.
17-GO TO statements which have been ALTERed and which reside in an independent section are reset to their original designation when the section is reloaded.
18-EXIT PROGRAM is executed as an EXIT statement if it occurs in a main program.
19-The syntax of the header statement conforms to the ANSI standard; an empty USING list in the PROCEDURE DIVISION header is not required for subprograms without parameters.
20-The first segment of a program may be independent.
21-Source text may follow a COPY statement on the same source line.
22-Conforming with the ANSI standard, there is a new syntax to replace the alphabet-name clause of the SPECIAL-NAMES paragraph in the ENVIRONMENT DIVISION.
23-FIPS flagging can be specified in the command line to the compiler by using the F switch.
24-The V switch may be used in the command line to the compiler; it changes the meaning of USAGE COMP, so that it is interpreted as USAGE DISPLAY.

The following 4.01 bugs have been repaired:

1-The first few bytes of the CRT driver are no longer destroyed when a subprogram returns.
2-INDEXED files in a SORT statement's USING or GIVING list can now be used.
3-A MOVE of a data item to a justified field shorter than the source field no longer destroys data following the destination field.
4-COMP-3 subscripts are now converted correctly.
5-Error messages no longer result when a program contains consecutive sections with the same segment number.
6-Random value is no longer used as a prompt character in the SCREEN SECTION.
7-The P switch now works, and allocates 100 bytes of additional stack space for each switch in the command line.
8-When RELATIVE organization specifies a key not in the file, an error message is no longer returned; runtime now searches for the existence of a higher-valued key.
9-After an error return from an OPEN statement, sometimes the program would abort; this has been fixed.
10-The first two bytes of WORKING STORAGE are not now destroyed by overlay loading.
11-Large COBOL programs containing overlays can now be linked.
12-A scaled numeric item can now be correctly MOVEd to an alphanumeric item.

(continued next page)

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

Microstat
Version 2.08

This update features some minor speed improvements on several programs. Hypergeometric has been fixed for cases when the sample is less than possible occurrences. Now stepwise multiple regression prints out the proper variables.

T/MAKER II
Version 2.5.2

This version corrects a minor bug in the Tally function.

Bugs

Lifeboat CP/M-80 For The TRS-80 Model II
Version 2.25C

The READ-ME.DOC file defines "set inverse video" and "reset inverse video" incorrectly the second time through. The first description (which follows) is correct.

		Decimal	Hex
SET INVERSE VIDEO	=	27,41	1B,29
RESET INVERSE VIDEO	=	27,40	1B,28

FORTRAN-80
Version 3.43

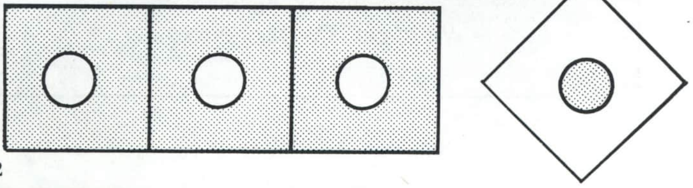
Linking Named Common does not work properly in L80 for this version; this bug has been fixed in version 3.44. It will work with PLINK, however.

Response

Another reader has responded to Howard O. Ehlers' letter in the April 1982 issue. Mr. Ehlers can obtain CP/M for the Cromemco line from: Intelligent Terminals Corp., 2320 Southeast Freeway, Houston, TX 77098 (Tel: 713/529-6696); or from Micah, 919 Sir Francis Drake Blvd., Kentfield, CA 94904 (Tel: 415/ 456-2262).

Clarification

On page 31 of the February 1982 *Lifelines*, in Ron Fowler's article on Using the CP/M-80 BIOS For Direct Disk Accessing, it is stated that Ward Christensen's DU.COM utility is free. However it is not free, but is available from CPMUG. (The latest version is on Volume 78.)



Sorry But . . .

We must raise our prices. *Lifelines/The Software Magazine* will begin its third year this June, and as we've grown, so have our expenses. As of June 15th, the following new prices will be in effect:

U.S., Canada, and Mexico:	$24.00 for twelve issues
All Other Countries:	$50.00 (air delivery)

So you June 1981 subscribers, send in your renewals. Back issue and dealer prices will remain the same.

Change of Address

Please notify us immediately if you move. Use the form below. In the section marked "Old Address", affix your *Lifelines* mailing label — or write out your old address exactly as it appears on the label. This will help the Lifelines Circulation Department to expedite your request.

New Address:

NAME

COMPANY

STREET ADDRESS

CITY STATE

ZIP CODE

Old Address:

NAME

COMPANY

STREET ADDRESS

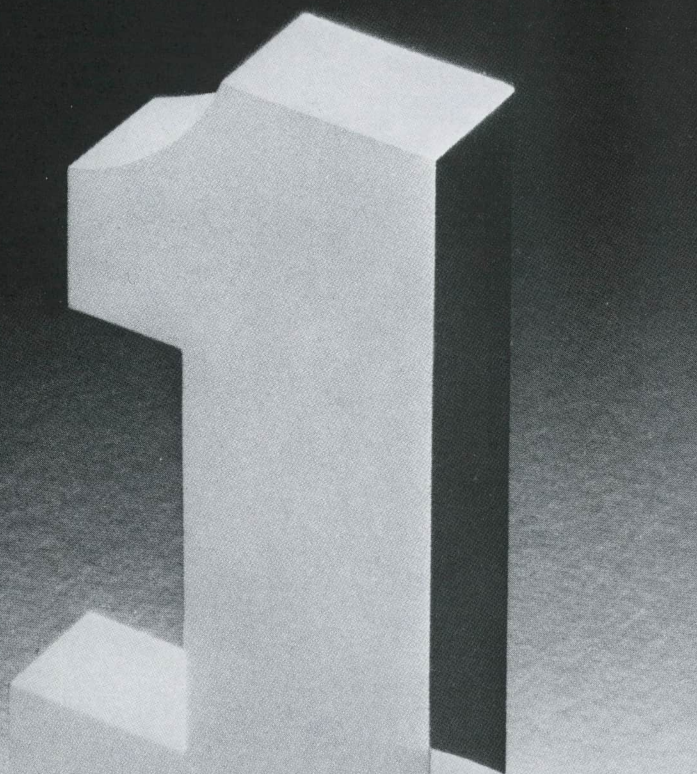
CITY STATE

ZIP CODE

We are dedicated to the achievement of a singular goal... to market fully supported software that sets standards of excellence.

Standards against which all software will be measured. Standards which require that we, as well as the OEM's and authors with whom we labor, constantly offer the state-of-the-art.

Our commitment to being Number One is too strong for Lifeboat to market anything less.



Lifeboat Associates

The standard for fully supported software.

1651 Third Avenue, NY, NY 10028. (212) 860-0300.
TWX 710-581-2524 (LBSOFT NYK).

VERSION LIST

March 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
MR	Memory Required

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

Product	S	P	MR	
ACCESS-80	1.0	8080/Z80	54K	
Accounts Payable/Cybernetics				Needs RM/COBOL. Runs w/CP/M-80, OASIS, UNIX
Accounts Payable/MC	1.0	8080/Z80	56K	For CP/M 2.2
Accounts Payable/Structured Sys	1.3B	8080	52K	w/It Works run time pkg.
Accounts Payable/Peachtree	07-13-80		48K	Needs BASIC-80 4.51
Accounting Plus		8080/Z80	64K	
Accounts Receivable/Cybernetics				Needs RM/COBOL. Runs w/CP/M-80, OASIS, UNIX
Accounts Receivable/MC	1.0	8080/Z80	56K	CP/M 2.2
Accounts Receivable/Peachtree	07-13-80	8080	48K	Needs BASIC-80 4.51
Accounts Receivable/Structured Sys	1.4C	8080	56K	w/It Works run time pkg.
Address Management System	1.0	8080		Requires 2 drives
ALDS TRSDOS	3.40	8080	32K	Needs TRSDOS. TRSDOS Macro-80
ALGOL 60	4.8C	8080	24K	
ANALYST	2.0	8080	52K	Needs CBASIC2,QSORT/ULTRASORT
APL/V80	3.2	Z80	48K	Needs APL terminal
Apartment Management (Cornwall)	1.0	Z80		Needs CBASIC2
ASM/XITAN	3.11	Z80		
Automated Patient History	1.2	8080	48K	
BASIC Compiler	5.3	8080	48K	
BASIC-80 Interpreter	5.21	8080	40K	w/Vers. 4.51,5.21
BASIC Utility Disk	2.0	8080	48K	
BaZic II	**03/03**			
Benchmark Word Processor	**2.2**			**Give Name & Model #'s of the video terminal)**
Benchmark Mail List	**1.1**			**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	48K	
BSTAM Communication System	4.5	8080	32K	
BDS C Compiler	1.45	8080	32K	w/'C' book
Whitesmiths' C Compiler	**2.1**	**8080**	**60K**	
BSTMS	1.2	8080	24K	
BUG / uBUG Debuggers	2.03	Z80	24K	
CBASIC2 Compiler	2.08	8080	32K	w/CRUN(2,204P, & 238)
CBS Applications Builder	**1.33**	**8080**	**48K**	**Needs no support language**
CIS COBOL Compiler	4.4,1	8080	48K	
CIS COBOL Compact	3.46	8080	32K	
FORMS 1 CIS COBOL Form Generator	1.06	8080		
FORMS 2 CIS COBOL Form Generator	1.1,6a	8080		
Interface for Mits Q70 Printer				CP/M 1.41 or 2.XX
COBOL-80 Compiler	**4.6**	**8080**	**48K**	
COBOL-80 PLUS M/SORT	4.01	8080	48K	
CONDOR II	2.06	8080	48K	
CREAM (Real Estate Acct'ng)	2.3	8080	64K	CBASIC needed
Crosstalk	1.4	Z80		
DATASTAR Information Manager	1.101	8080	48K	
Datebook-II	**2.04**	**8080**	**48K**	**Needs 80x24 terminal, N/A for CDOS, CP/M 1.4, MP/M**
dBASE-II	**2.3B**	**8080**	**48K**	
dBASE-II Demo	2.3A	8080	48K	
Dental Management System 8000	8.7A	8080	48K	Needs CBASIC
Dental Management System 9000	**1.08**	**8080**	**48K**	**Needs CBASIC**
DESPOOL Print Spooler	2.1A	8080		
DISILOG Z80 Disassembler	4.0	Z80		Zilog mnemonics
DISTEL Z80/8080 Disassembler	4.0	8080/Z80		Intel mnemonics,TDL extensions
Documate/Plus	1.4	8080	36K	
Documate/Plus/Demo	1.5			
EDIT Text Editor	2.06	Z80		
EDIT-80 Text Editor	2.02	8080		
Emulator-86	**1.0**			**An Emulator for CP/M-86**
FABS-I	**2.7**	**8080**	**32K**	
FABS II	**4.15**		**8080/Z80**	**48K**
FILETRAN	1.20		32K	1-way TRS-80 Mod I,TRSDOS to Mod I CP/M
FILETRAN	1.4		32K	Needs TRSDOS. 2-way TRS-80 Mod I,TRSDOS & Mod I CP/M
FILETRAN	1.5		32K	1-way TRS-80 Mod II,TRSDOS to Mod II CP/M
Financial Modeling System	2.0		48K	
Floating Point FORTH	2	8080/Z80	28K	
Floating Point FORTH	3	8080/Z80	28K	
FORTRAN-80 Compiler	3.43	8080	36K	
FPL 56K Vers.	2.6	8080	56K	
FPL 48K Vers.	2.6	8080	48K	

VERSION LIST

Product	S	P	MR	
General Ledger/Cybernetics				**Needs RM/COBOL. Runs w/CP/M-80, OASIS, UNIX**
General Ledger/MC	1.0	8080/Z80	56K	Needs CP/M 2.2 or MP/M
General Ledger/Peachtree	07-13-80	8080	48K	Needs BASIC-80 4.51
General Ledger/Structured Sys	1.4C	8080	52K	w/It Works Package
General Ledger II/CPaids	1.1	8080	48K	Needs BASIC-80 4.51
GLECTOR Accounting System	2.02	8080	56K	Use w/CBASIC2,Selector III
GLECTOR IV Accounting System	1.0	8080		Needs Selector IV
HDBS	1.05A	+	52K	
IBM/CPM	1.1	8080		
Insurance Agency System 9000	1.08	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/Z80	56K	Needs CP/M 2.2
Inventory/Peachtree	07-13-80	8080	48K	Needs BASIC-80 4.51
Inventory/Structured Sys	1.0C	8080	52K	w/It Works Package
Job Cost Control System/MC	1.0	8080/Z80	56K	Requires CP/M 2.2
JRT Pascal System	1.4	8080	56K	
LETTERIGHT Text Editor	1.1B	8080	52K	
LINKER		Z80		
MAC	2.0A	8080	20K	
MACRO-80 Macro Assembler Package	3.43	8080/Z80		
MAG/base1 (LMS)	2.0.1	8080	56K	Needs CBASIC, 2.06 or later & 180K/drive
MAG/base2 (IMS)	2.0.1	8080	56K	Needs CBASIC, 2.06 or later & 180K/drive
MAG/base3 (ADS)	2.0.1	8080	56K	Needs CBASIC, 2.06 or later & 180K/drive
Magic Typewriter	3	Z80	48K	
Magic Wand	1.11	8080	32K	
MAG/sam3	4.2	8080	32K	
MAG/sam4	1.1	8080	32K	Needs CBASIC
MAGSORT-C	**1.0**			**For CBASIC**
MAGSORT-M	**1.0**			**For MBASIC**
MAGSORT-M	**1.0**			**For Compilers — BASCOM, FORTRAN-80, PL/I-80**
MAILING ADDRESS Mail List System	07-13-80	8080	48K	
Mail-Merge	3.0	8080		
Master Tax	1.0-80	8080	48K	
Matchmaker		8080	32K	
MDBS	1.05A	+	48K	
MDBS-DRS	1.02	+	52K	
MDBS-QRS	1.0	+	52K	
MDBS-RTL	1.0	+	52K	
MDBS-PKG		+	52K	w/all above MDBS products
Medical Management System 8000	8.7a	8080		Needs CBASIC
Medical Management System 9000	1.1	8080		Needs CBASIC
Microcosm		Z80		CP/M 2.X or MP/M
Microspell	4.3	8080	48K	Needs 150K/drive
Microspell Demo	1.0			For Dealers Only
Microstat	**2.08**	**8080**	**48K**	**Needs BASIC-80, 5.03 or later, or CBASIC**
Microstat for Apple	2.0			
Mince	2.6	8080	48K	
Mince Demo	2.6	8080	48K	
Mini-Warehouse Mngmt. Sys.	5.5	8080	48K	Needs CBASIC
Money Maestro		8080/Z80	48K	CP/M 1.4 or 2.2
MP/M-I	1.0			
MP/M-II	2.0	8080	48K	Needs MP/M
MSORT	1.01	8080	48K	
Mu LISP-80/Mu STAR Compiler	2.10	8080		
Mu SIMP / Mu MATH Package	2.10	8080		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	2.2		44K	Also for MP/M
PAS-3 Medical	1.78	8080	56K	Needs 132-col. printer & CBASIC
PAS-3 Dental	1.64	8080	56K	Needs 132-col. printer & CBASIC
PASM Assembler	1.02	Z80		
Pascal/M	4.02	8080	56K	
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/Cybernetics, Inc.		Z80		Needs RM/COBOL
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)	2.2	8080	56K	Z80/8080

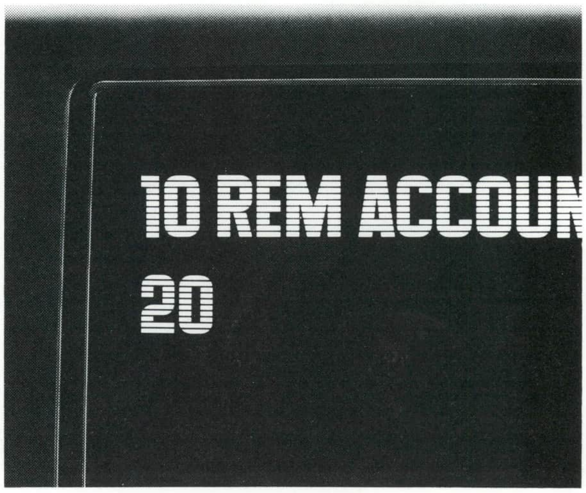
(continued next page)

VERSION LIST

Product	S	P	MR	
PLAN80 Demo	**1.1**			
PL/I-80	1.3	8080	48K	
PLINK I Linking Loader	3.28	Z80	24K	
PLINK-II Linking Loader	1.10A	Z80	48K	
PMATE	3.02	8080	32K	
POSTMASTER Mail List System	3.5	8080	48K	
Professional Time Acctg	3.11a	8080	48K	Needs CBASIC2
Programmer's Apprentice		8080/Z80	56K	Needs BASIC-80
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
Property Manager	1.0	8080	48K	Needs CBASIC
PSORT	1.3	8080		
QSORT Sort Program	2.0	8080	48K	
Real Estate Acquisition Programs	2.1	8080	56K	Needs CBASIC
Remote	3.01	Z80		
Residential Prop. Mngemt. Sys.	1.0	Z80	48K	
RM/COBOL Compiler				w/Cybernetics CP/M 2, OASIS, UNIX
RAID	5.0.2	8080	28K	
RAID w/FPP	5.0.2	8080	40K	
RECLAIM Disk Verification Program	2.1	8080	16K	
SBASIC	5.4	8080	48K	
Scribble	1.3	8080		
SELECTOR-III-C2 Data Manager	3.24	8080	48K	Needs CBASIC
SELECTOR-IV	2.17	8080	52K	Needs CBASIC
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/Z80	32K	Needs Word Processing Program
Standard Tax	1.0	8080	48K	Needs BASIC-80 4.51
STATPAK	1.2	8080		Needs BASIC-80 4.2 or above
STIFF UPPER LISP	**2.8**	**8080**	**48K**	
STRING BIT FORTRAN Routines	1.02	8080		
STRING/80 bit FORTRAN Routines	1.22	8080		
STRING/80 bit Source	1.22	8080		
SUPER SORT I Sort Package	1.5	8080		Max. record=4096 bytes
SELECT		8080/Z80	40K	
T/MAKER II	**2.5.2**	**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	
TINY C Interpreter	800102C	8080		
TINY C-II Compiler	800201	8080		
TRS-80 Customization Disk	1.3C	8080		
ULTRASORT II	**4.1C**	**8080**	**48K**	
Lifeboat Unlock	1.3	8080		Use w/BASIC-80 5.2
VISAM	2.3p	8080	48K	
Wiremaster	3.12	Z80		Needs 180K/drive
Wordindex	3.0	8080	48K	Needs WordStar
Wordmaster	1.07A	8080	40K	
WordStar	3.0	8080	48K	
WordStar w/MailMerge	3.0	8080	48K	
WordStar Customization Notes	3.0	8080		
XASM-05 Cross Assembler	1.05	8080	48K	
XASM-09 Cross Assembler	1.07	8080	48K	
XASM-51 Cross Assembler	1.09	8080	48K	
XASM-F8 Cross Assembler	1.04	8080	48K	
XASM-400 Cross Assembler	1.03	8080	48K	
XASM-18 Cross Assembler	1.41	8080		
XASM-48 Cross Assembler	1.62	8080		
XASM-65 Cross Assembler	1.97	8080		
XASM-68 Cross Assembler	2.00	8080		
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		Needs 50K/drive
Z80 Development Package	3.5	Z80		N/A-Magnolia,Superbr'n,mod.CP/M
ZDM/ZDMZ Debugger	1.2/2.0	Z80		For N'Star,Apple,IBM 8"
ZDT Z80 Debugger	1.41	1.41	Z80	N/A-Superbr'n,mod.CP/M
ZSID Z80 Debugger	1.4A	Z80		N/A-Superbr'n,mod.CP/M

+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.

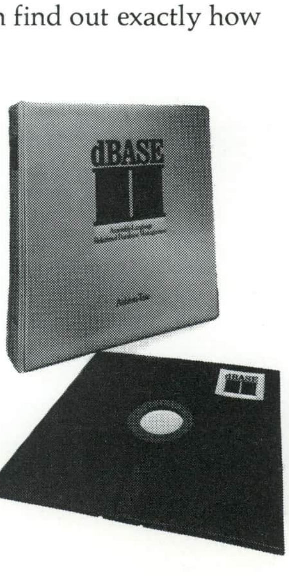
Call for our Dealer Plan and OEM run-time package prices, then take us up on our money-back guarantee. Send us your check and we'll send you a copy of dBASE II that you can exercise on your CP/M® system any way you want for 30 days.

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®

1651 Third Avenue / New York, N.Y. 10028

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

[redacted]