LIFELINES™

The Software Magazine™

$3.00 March 1983 Volume III, No. 10 (ISSN 0279-2575, USPS 597-830)

Fox & Geller asks dBASE II™ users: Which would you prefer?

Annual Report, 1982

Unit Sales and Dollar Volume

PRODUCT A, Quarterly Unit Sales, $39.95/unit net:

	1981	1982	
1st	821	1037	
2nd	510	600	
3rd	385	530	
4th	1020	1298	
	2736	3465	TOTAL UNITS
	$109,303.20	$138,426.75	TOTAL SALES

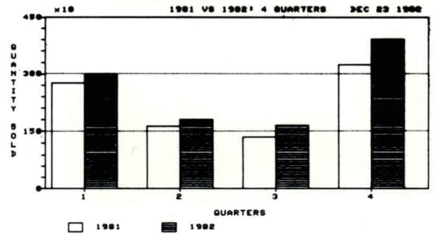
PRODUCT B, Quarterly Unit Sales, $24.95/unit net:

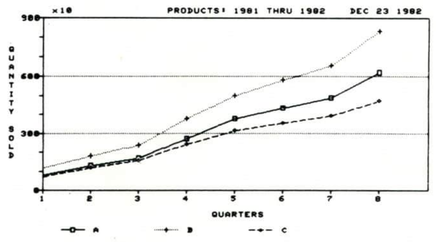
	1981	1982	
1st	1196	1240	
2nd	624	803	
3rd	590	752	
4th	1365	1810	
	3775	4605	TOTAL UNITS
	$ 94,186.25	$114,894.75	TOTAL SALES

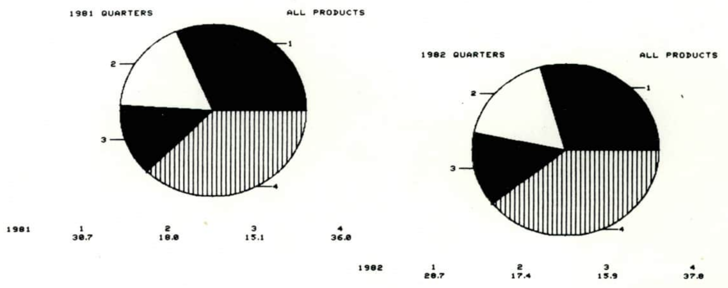
PRODUCT C, Quarterly Unit Sales, $49.95/unit net:

	1981	1982	
1st	745	697	
2nd	485	401	
3rd	380	365	
4th	851	812	
	2461	2275	TOTAL UNITS
	$122,803.90	$113,636.25	

OR







On the left, you have numbers which are important, but can be a bore to read. And it's easy to miss important trends embedded in those numbers.

On the right are graphic aspects of the same numbers. Graphics that are quickly understandable, visually dynamic, and all available from one package, **dGRAPH™** by **Fox & Geller.**

That's right. **dGRAPH** gives you the flexibility to produce bar charts, pie charts, line graphs, and combination piebar charts. All of them are **easy to do,** as easy as naming the kind of graph you want and what kind of data you want to graph. That's it. **dGRAPH's** unique AUTOGRAPH™ feature takes care of the rest in seconds. That includes loading data, computing scales, drawing grid lines, shading, overlay graphs (if any), and labelling. And the results are transferable to EPSON, OKIDATA, and a growing list of other popular printers so as to enhance your meetings and reports. You get all of the above capabilities plus powerful data manipulation features so you can graph various fields—all this for just $295.00.

So, dBASE II users, which would you prefer? We think **dGRAPH** is the obvious answer. Don't you?

Ask for **dGRAPH** by name at your local computer dealer. And while you're there, see **Fox & Geller's** full line of quality software. Software that's practical, reliable, and reasonably priced. Software that's created by Jeff Fox and Jacob Geller, individuals who stand behind every product that bears their names.

Use the Reader Service Card for more information. Or contact:

dGRAPH is also available for non-dBASE II users!

dBASE II is a trademark of Ashton-Tate. dGRAPH and AUTOGRAPH are trademarks of Fox & Geller.

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

FOX&GELLER

794-8883

LIFELINES™ The Software Magazine™

Publisher: Edward H. Currie
Production Manager: Harold Black
Art and Design Manager: Kate Gartner
Marketing & Circulation Manager: Patricia Matthews
Typographer: Harold Black
Software Consultant: Susan Sawyer
Managing Editor: Patricia Matthews
New Versions Editor: Lee Ramos
Technical Editor: Al Bloch
Technical Consultant: Emil Sturniolo
Cover: K. Gartner
Printing Consultant: Sid Robkoff/E&S Graphics

Lifelines (ISSN 0279-2575, USPS 597-830) is published monthly at a subscription price of $24 for twelve issues, when destined for the U.S., Canada, or Mexico, $50 when destined for any other country.

Second-class postage paid at New York, New York, and other locations. POSTMASTER, please send changes of address to Lifelines Publishing Corporation, 1651 Third Ave., New York, N.Y. 10028.

Opinion Editorial

Edward H. Currie

Tis All Done With Mirrors My Son ...

There is perhaps no more interesting realm to contemplate than that of the cognitive process. The intensity and pace of brain research has increased considerably in recent years and continues its inexorable march towards an understanding of what is perhaps to be the most intriguing of computers. Technology has been hard pressed to rival the physiological and psychological marvels of the brain but continues an inexorable march of its own leaving the casual observer to wonder what will be next revealed. Packing densities in the human brain are on the order of 10^7 circuits/cm^3 with switching speeds on the order 10 milliseconds. Memory density is believed to be on the order of 10^{15} bits/cm^3 which leads to an estimate of some 10^{10} switching elements in the brain.

The basic unit of time in the realm of of the microcomputer is the nanosecond. A convenient rule of thumb is that one nanosecond is to one second as one second is to thirty years. Typical speeds for interesting events in todays microcomputers are on the order of hundreds of nanoseconds. Now it might seem on the face of it that this is awfully fast and that there could be little reason for wanting to have things take place much faster! But consider the following, suppose that you wish to test all possible sequences of 25 instructions in a microcomputer just to be sure that there is no particular sequence which will send the processor into the twilight zone. Let's further assume that the typical instruction takes on the order of 1.6 microseconds (i.e. 1600 nanoseconds). Then all possible combinations of 25 instructions would take about 787,000,000,000 years! Assuming of course that we were executing 625,000 instructions per second over this period.

Thus it's relatively easy to see that executing instructions rapidly is very important in some cases. But electrons and holes do have mass , i.e., inertia, and therefore getting them to rapidly accelerate does require more effort the higher the switching speeds, i.e. accelerations. Technologists are therefore in search of techniques for increasing switching speeds and while the rest of us realize that fluid and mechanical computers are now little more than historical artifacts we assume that future computers will be electronic marvels. But perhaps not ...

The upper limit on switching speeds for transistors of the type used in semiconductor microprocessors is on the order of one nanosecond. In casting around for alternate approaches, optical devices come to mind. In particular, the optical transistor is now a reality with switching speeds on the order of one picosecond, i.e. 1000 times faster than its electronic counterpart. But perhaps even more exciting is the prospect of basic switching components which have more than two stable output states as compared to transistor switches which are either on or off (i.e. in a "zero" or "one" state). This might well result in the development of entirely new concepts of computer design and architecture. Furthermore these "optical transistors', let's call them optsistors, are capable of parallel operating modes as compared to the serial operations of their transistor counterparts. Thus not only can such devices do more things but they can do them simultaneously and in more ways. . . .

Thin-film technology in conjunction with laser technology could be combined to build computers of the future linked by optical fibers to open a realm as yet undreamt of by today's computer types. Perhaps computers of the future will be hybrids combining the best features of biology, optics, lasers and electronics. And as tomorrow's child asks how does it work the reply may well be 'tis all done with mirrors my son ...'

* * *

The rapid proliferation of microcomputers continues apace with historians beginning to make efforts to record in detail the happenings of recent years. Unfortunately much is being lost in the translation. Claims and counter claims abound as to who did what first, said what first and the origins of many important contributions are in real danger of being lost forever.

In the months ahead we'll cover in wide ranging discussions some of the more fundamental contributions to microcomputerdom and those responsible for them.

Most of those who have made noteworthy contributions are intelligent, sensitive people who if they have been recognized at all failed to get the proper credit. We'll attempt to identify such people for you and point out those who in some sense have been obscured but nonetheless have provided and still provide the foundation upon which the microcomputer world continues to build.

One such individual is Ward Christensen. Ward has done much to provide the foundation for the quality and quantity of public domain software and its free and frequent exchange by setting high standards for documenting of source code, providing a de facto standard for asynchronous telecommunications, spearheading the CPM Users Group (CPMUG), computerized bulletin boards, etc. Ward asked recently who was responsible for providing the first CBBS system and was told that it was some guy in Chicago. After a few moments reflection he realized that they were referring to none other than Ward Christensen !

Or what about Kelly Smith who at his own expense and with a considerable expenditure of his time has provided thousands of hours of access to his CBBS in Simi Valley, California. Kelly maintains 20 Megabytes of public domain software free for the asking. Not to mention a sizable contribution of his own software which he has insisted be made available to all those who wanted it.

And there are many more, some of whom have made small contributions and others large but each provided a vital link in the development of microcomputers and microcomputer software.

Kathy McMahon called the other

(continued on page 9)

Feature

Get a Better Performance Out of CB-80

Robert P. VanNatta

The goal of this article is to explore a number of possibilities for getting better performance out of CB-80 programs. It should not be assumed that the suggestions which follow will give similar results in other dialects of Basic. Some will and some won't!

Performance, when applied to a high level language such as CB-80, usually means four things: 1) Run time speed; 2) code size; 3) compiler behavior; and 4) memory management.

Most of this article will be addressed to the first two of these items. With respect to the third, if you have version 1.3 or later of CB-80, you may have considerable confidence that it works. The earlier versions had some significant problems but were replaced free to registered users. At last report, Digital Research had a version 1.4 in the works but hadn't decided on a release date or just how it would be different from version 1.3.

The most annoying bug in version 1.3 is its inability to allow the use of the same user-defined function twice in the same line and the accompanying failure of the compiler to trap this error.

Speed Is Not the Issue

A favorite saying of Gordon Eubanks with respect to his CB-80 is that "Speed is not the issue". This claim, wherever he makes it, seems to draw a catcall or two out of every crowd, and, if nothing else, the remarks here may serve to address some of those questions.

It is this writer's notion that Gordon is right, however. My reasons for this conclusion are essentially that, given the memory addressing capability of the 8080 processor (64k), and the disk capacity and printer performance now customary, a hardware feature is usually the limiting factor in a program, not the raw speed of the language. This is not to suggest that CB-80 is slow. Quite to the contrary, it is quick. It is, in fact, probably quick enough for almost any application, if you don't do something stupid so as to destroy its performance. I shall presently document a few "monkey wrenches" which, properly, should be left out of most programs. Obviously, no compiler is going to generate code which will run as fast as the best assembly language routines. If for no other reason, this will always be true because the compiler must generate code to take into account all the "worst case" situations. A compiler that blows up when the user tries to do some exotic thing with it is said to be buggy; whereas dedicated code that will perform a limited function well is said to be optimized.

If The Program Is Too Slow

Speed is, of course, relative. We all dream of instant number crunching; but that is, obviously, not possible. Speed should be considered a problem if any of the following standards are not met. Operator input loops should generally accept input as rapidly an an operator can enter it. Maximum delays during operator input routines ought not to exceed 2 seconds and these should be infrequent. Printer routines should produce data rapidly enough to keep the printer busy. Display routines should work at the speed of the hardware. Crunching routines which require little or no operator intervention must be completed with sufficient dispatch so as to assure that the operator is not late for his/her coffee break.

If your pet program fails to run fast enough to suit you, the first thing to do is to figure out why. The most common reason for slow programs is disk bind. A program is said to be disk bound when the disk operations are sufficiently frequent to affect adversely the overall performance of the program. Disk bind is much easier to diagnose than to fix. If you simply notice the correlation, or lack of it, between the disk activity light and the annoying delays, you have your answer. If disk bind is the problem, faster code is not going to solve the problem, as all forward progress in your program must stop during disk reads and writes. Solutions to this problem involve getting rid of either the disks or the accesses, and neither is language dependent.

Optimize Record Lengths

One thing that can be done to improve performance during disk reads and writes is (when using random access files) to assign record lengths in multiples or sub-multiples of 128 bytes. Thus 32,64,128, and 256 are excellent choices for random file record lengths. The reason for this is that the typical CPM BIOS blocks the disks in 128 byte sectors. Even though sector-spanning reads and writes are completely transparent to the user, the fact remains that if any record length is used other than a multiple of the sector size, almost every read or write will have to take place in two different sectors in two different locations on the disk. For example, if a record length of 127 bytes is selected, only two out of every 128 records will lie entirely within a single sector. Similarly a record length or 63 or 65 would result in about half of the records spanning sectors. A record length of 256 will get the record in exactly 2 sectors instead of scattering it over three.

The importance of record length optimization should not be overstated. It will not cure disk bind, and since most of the disk delay is accounted for in the seek and head settling times, the improved performance will not be as dramatic as you might expect at first blush. I mention it simply because it is not made all that clear in the CP/M documentation, and because the "Molasses in January" syndrome is usually caused by a combination of trivial things, not one big thing.

(continued on next page)

Find the Hot Spots

It is a poorly kept secret that in a typical computer program 10% of the code does 90% of the work. Identify that code and give it your attention. Nobody appreciates fast code that is never used.

Use Integer Variables

The biggest single thing that you can do for CB-80 code to improve performance is to find some real variables and convert them to integers.

As we have mentioned before, CB-80 is a native code compiler. If compiler toggle 'I' is set during compilations, the assembly language instructions generated will be interlisted with the basic code. Such an interlisting is very educational, even if you are not a proficient assembly language programmer.

It has been said over and over again that integer operations execute much more rapidly than real number operations. If you interlist some simple programs you will begin to understand why. Most common activities involving integers are processed by the generation of a few inline assembly language commands.

For example, a simple FOR-NEXT loop using integer variables such as:

```
FOR I% = 1 to 1000
NEXT I%
```

generates only nine 8080 instructions, and makes no calls to the library, Figure 1 (see page 37) (with the documentation added) is the code displayed by the compiler with the interlist (toggle I) set.

Note that there are only 6 instructions in the loop. Since this code doesn't look exactly like Ward Christensen has taught us to write it, mainly because it is a relocatable module and not executable code, I took the liberty of compiling and linking this program, and then examining it with DDT.

Finding the code wasn't all that difficult, since, if the linkage is done with toggle M set, the address of the first instruction in the module is reported. The actual code relating to this empty FOR-NEXT loop as found on disassembly with DDT is as follows:

```
0399    LXI     H,0001
039C    SHLD    0514
039F    JMP     03A9
03A2    LHLD    0514
03A5    INX     H
03A6    SHLD    0514
03A9    LXI     D,FC17
03AC    DAD     D
03AD    JNC     03A2
```

And, lo and behold, we find that the linker has converted the pseudo-assembly language into something that would even get the approval of Ward Christensen.

Now, I assume, there is someone who can think of a way to recreate the logic of a BASIC FOR-NEXT loop in less than 9 instructions, but it seems to this writer that the opportunity for improvement is severely limited, no matter who writes it. The only idea that this writer has for writing faster code than this would involve switching to a one byte data type and using 8 bit arithmetic. In theory, this would make a faster loop, but the whole idea is absurd because loop size would also be limited to one byte or 255. Specifically, my stopwatch is not programmed to detect the difference in execution speed between a DAD instruction and an ADD instruction in 255 repetitions or less.

What About Generating Code For Line Numbers

If you write programs that crash a lot, you probably appreciate a common feature present in many versions of BASIC that tells you what line you crashed on. CB-80 does not normally bother to give you this information; however, a compiler toggle is provided to force this information into the code. If this toggle is set, the interlisting will reveal two additional instructions for every logical line of code as follows:

```
LXI    H,1      ;where 1 is the line
                number
CALL  ?LNUM   ;library routine
                containing
                ;a SHLD (address)
                ; and a RET instruction.
```

The result, when compiled into the empty FOR-NEXT loop, is the generation of this extra code twice, once for each line. This compiler toggle is necessary for the ERRL function to work, but notice the price. A total of 8 instructions are added to a program which only had 9 instructions to start with. Four of these instructions are inside the loop and are thus critical. The hot spot in our loop has, accordingly, gone from 6 instructions to 10 instructions. Percentagewise, this is a very heavy performance penalty, and suggests that a program with line numbers compiled in is going to be a much different animal than one without.

What About Real Numbers in a Loop Index

The following loop looks similar to the integer loop except for the fact that it uses real numbers. The loop I used is as follows:

```
FOR A.REAL = 1.0 to 1000
NEXT A.REAL
```

The similar appearance to the casual programmer is very deceiving. Figure 2 (see page 37) shows the code generated for a real loop.[1]

The price of those six library calls is very dear. I made some effort to chase some of the them around the library with DDT. For the most part, I got dizzy trying to follow the flow, but I did satisfy myself that many of these library calls involve the execution of hundreds of instructions. Specifically, as nearly as I can tell the integer loop runs about 100 times faster than the real loop.

This does not mean that a real loop does not have its place. In order to assure an integer loop, the index, loop delimiters, and step should all be either integer constants or integer variables. They may be positive or negative. With respect to a real loop, all delimiters may be any legal real number. Likewise, they may be either constants or variables. Furthermore, they may be changed at any time. This means, for example, that if you need to run a FOR loop with the stepping rate set equal to the square root of the index variable, you can do it.

Obviously, not every optimization will result in a reduction of code space required by 40% and increase speed a factor of 100 times, but this gives idea of the things that can be done.

I would propose the following rule of

thumb: where speed is critical, go to great extremes to use integers wherever possible. Even if the ultimate result must be a real or string variable, use integer variables for portions of the calculations where possible, and then make a conversion at the last possible moment. Likewise, it should be noted that the use of constants wherever possible makes for faster code. For example, if the loop size in Figure 1 is changed from a constant to a variable the amount of code required for the loop will double.

Use Complex Statements

My comments on the use of integers should not be taken to imply that CB-80 does a bad job on string operations. It is just that integers go like lightning.

There are some things that can be done to help real number crunching and string jerking. This is principally accomplished by building complex nested functions. Consider the following program segment which is illustrative of string operations:

```
a$="lkj"
a=len(a$)
b$=a$+" "+d$
x$=left$(b$,a+10)
```

Although a bit inane, this code segment, when compiled as a public function, takes up 91 bytes of code consisting of 27 op-codes including 12 calls to the CB-80 library. If this code is wrapped in an integer based FOR-NEXT loop, my Radio Shack Model 16 (on the Z-80 Board) takes almost 10 seconds to execute that mess 1500 times.

When optimized that same code can be made into a single line complex statement as follows:

```
x$=left$("lkj"+" "+d$,len("lkj")+10)
```

This code will do the same thing. However, it will compile as a public function in 57 bytes of code containing only 14 in-line op-codes and 5 trips to the library. Execution time was measured at under 3 seconds for 1500 iterations in a FOR-NEXT loop. This figures out to about a 40% code reduction and a speed increase by a factor of over 3 times.

Curiously, I timed the same string routines in CBASIC at 27 seconds and 8 seconds repectively. My clocked times for MBASIC ver. 4.51 were 17 seconds and 10 seconds, and slightly longer with MBASIC with TRSDOS. The real shocker, though, was SBASIC. It is a native code compiling BASIC which I, unfortunately, bought the month before CB-80 was released. It required 56 seconds to execute the multiline version 1500 times and 30 seconds for the one line version. My profound conclusion is that, as with all BASICS, CB-80 will perform substantially better if efforts are made to nest the functions. The reason, at least in CB-80, for this increase in performance is that when functions are nested the results of one calculation are passed directly to the next function without a redundant memory write and read in between.

Beware of phony schemes to compact code

Not too long ago, I had an occasion to work with a college student whose assignment of the day was to optimize his program by reducing the number of PRINT statements to the lowest common denominator. He was seeking to accomplish this by using the word PRINT once, and then putting everything but the kitchen sink after it. One need only interlist the code of a PRINT statement once to find out that this doesn't accomplish a thing. The line

```
PRINT A;B;C;
```

compiles and generates exactly the same code as:

```
PRINT A; :PRINT B; :PRINT C;
```

The reason is that the print routine requires that the H-L register pair be loaded with a pointer followed by a call to one or two library functions depending on what is being printed. This has to be repeated as many times are there are items to print. It can be said, therefore, that the word PRINT is implied for each item printed, and nothing is to be gained by economizing on its use.

Another myth that needs to be exploded is the idea that the following:

```
WHILE X0%<100
      X0%=X0%+1 WEND
```

is somehow a more efficient alternative to a FOR-NEXT loop. It isn't! This is not to say that WHILE loops don't have their place. Just don't use them where a FOR-NEXT loop will work just as well.

The integer WHILE loop uses a total of eleven instructions compared to the 9 for the FOR-NEXT loop. Ironically, the extra two instructions are not where you would expect them to be. The compiler is clever enough to index X0% with an IDX instruction instead of using an add routine but the catch comes with the condition testing. The WHILE loop must handle a number of relational operators and so the DAD-JNC trick (see figure 1) used on FOR-NEXT loops isn't used.

When using an index counter always write it in the conventional form of:

```
X0%=X0%+1
  and not as
X0%=1+X0%.
```

The second form requires four instructions instead of the three required for the first version. Worse yet, however, is the following:

```
Y%=1 : X%=X%+Y%
```

This compiles into a total of seven instructions and will therefore take more than twice as long to execute as the first version. Of course, if you really like it slow, you can use REAL numbers and make three dives for the library to access that famous BCD math function.

Nested IF-THEN

The authors have bragged about their nested IF-THEN statements in CB-80. They take some getting use to. I have been working with CB-80 since it first came on the market about a year ago. Only now am I reluctantly becoming an advocate of their use. Nested IF-THEN's provide the opportunity to generate flow control statements with a complexity which challenges comprehension. (Translation: It makes a rat's nest!) Please review, for a moment, Figure 3A and Figure 3B. These two functions perform exactly the same job and will compile to within 6 bytes of the same amount of code. I wrote this particular function for the purpose of dealing with a name and address file which was in all upper case that I desired to print in lower case (except, of course, for the first letter of each word).

If you examine the function for contents, you will observe that it must test three conditions and toggle a flag in order to get the job done.

(continued on next page)

Such an activity is a traditional place for the use of the AND construct. The logic is straightforward. If all three conditions are true, do it; otherwise, forget it!

The failure of this approach from a performance standpoint is obvious. All three conditions are always tested before a decision is made. The more sophisticated approach is to test the conditions one at a time, and branch immediately if a condition fails. After all, what is the point in checking to see if a number is too large if you have already determined that it is too small? Nested IF-THEN's permit just this sort of flow control. Figure 3B tests the conditions one at a time and branches immediately. The results in terms of performance are dramatic. Function 3A requires the execution of no less than 40 assembly language instructions inside the loop to test the conditions and control the flag. By contrast, figure 3B requires that only 12 instructions be executed if the first test is failed, and a maximum of 19 instructions if all tests are true.

The price of efficiency is complexity.

It is my argument that it is worthwhile to squeeze 20 or 30 instructions out of a loop which is as tight as this one. As I mentioned above, the total amount of code is about the same; but only half of it is executed on any one pass. One of the most elegant things about figure 3B is the way that the flag toggle is buried. Figure 3A methodically tests every character to see if it is a space and then sets the flag as indicated. On the other hand, Figure 3B recognizes that there is no reason to even check for a space unless the character fails the first test indicating that it is 'out of range-low'. Similar efficiencies follow on through. The second test is for the flag. If the flag is set, we release it, and branch; otherwise, we conduct the high-range test. At this point, we get still another benefit of our sequential testing. We don't have to worry about releasing the flag on completion of the operation. The reason is obvious: you can't get to the third test, if the flag is set!

For lack of a better term, I have dubbed the procedure described above as 'multi-path flow control'. It is surely a familiar concept to accomplished assembly language programmers, but I suspect that the idea of playing hop-scotch in a rat maze is a bit novel to the BASIC programmer. Occasionally, I have seen the functional equivalent of nested IF-THENs generated with a handful of free standing IF-THEN statements coupled with a bunch of GOTOs which branch to still more IF-THENs and GOTOs. My comments here should not be construed as advocacy for such coding.

What About XOR

For those of you 'hep' on 'bit diddling' you will notice that I have chosen to implement the shift to lower case with an ADD procedure: **CHR$(character% + 32)** rather then the equally correct XOR procedure: **CHR$(character% XOR 32).** This is directly contrary to the recommendations in my Radio Shack Guide to Microsoft BASIC. The use of XOR under these conditions does not appear to be appropriate in CB-80, however. The compiler output for the XOR version requires four MOV instructions and two XRA instructions, whereas the compiler output for the addition is a single DAD instruction. The result is that the latter compiles in 5 less bytes of memory, and, although the execution of the sixteen bit addition is slower than the execution of the XRA instruction, it is surely not slower then four MOVs and two XRAs.

The lack of a one byte date type in CB-80, and the absence of a 16 bit XRA instruction in the 8080 instruction set, appear to dictate this result. Presumably, however, when CB-80 becomes CB-86 or CB-68, we will find a 16 bit EXCLUSIVE OR instruction implemented, and the avoidance of the XOR will no longer be justified. For example, the instruction set for the Motorola 68000 chip supports 'XORing' between any of its registers in the 32 bit (long word) mode.[2]

I should note, however, that the XOR does have a place. Specifically, if you want to go both ways, i.e. change lower case to upper case as well as change upper case to lower case, the XOR will do that. An XOR 32 will flip the bit in position 5 of the companion number. The effect of this when applied to the ASCII character set is to swap cases on any letter in the alpha range. I have disregarded the utility of generating an upper case character because of the existence of the built in UCASE$ function in CB-80.

GO TO

We have all been taught to avoid the use of the GOTO construct like the plague. Reasons for this range from claims that it is poor programming style to technical arguments that interpreter BASICs don't know where to go without rescanning the entire program in order to locate the matching label. Similarly I have read numerous works on strategies for arranging subroutines physically within the program so they can be quickly located during program execution.

The CB-80 compiler does not suffer from these operational limitations. Frequent use of GOTO in a program may be evidence of poor programming style, but it is not going to destroy the performance of the program. GOTO and GOSUB instructions generate JMP (label) and CALL (label) compiler output, respectively. The linker resolves the label ambiguity and assigns an absolute memory address making the GOTO construct the quickest way to get there from here. I have noted a tendency by programmers, including myself, to use what I will call a phony WHILE loop. It is phony in the sense that there is really no condition to test, but, rather, what is really desired is a backwards jump. Don't do it! All you accomplish is to require the testing of a condition that you don't care about. A JZ (jump zero) is not going to get you there quicker than a JMP. Similarly, I have noted some confusion as to the best way to get out of a WHILE loop. If you are only looking at the source code, it is tempting to think that the best way out of a WHILE loop is to force the condition to true and fall out the bottom. This is not so. The flow of control on a WHILE loop is controlled by various permutations of JMP instructions. There is no reason not to either enter or depart a WHILE loop with a GOTO instruction. Entry in the middle is especially useful if you have a routine that you want executed at lease once prior to testing for the condition. Likewise, if you are in the middle of a WHILE loop, and you are done with the loop, and know you are done, JMP out! Don't waste the time forcing your way out the bottom. In summary, insofar as the flow of control goes, as nearly as I can tell, all the flow control is compiled 'in line' and relies on familiar jump, jump conditional, and call constructs. There are no algorithmic searches, seeks, or library calls to get from point A to point B.

Similarly the loop constructs don't have any stacks or other support facilities that have to be reset on entry or exit. Thus WHILE and FOR looping constructs may be freely entered and exited at any location within the loop by use of any conditional or unconditional jump (IF-THEN, GOTO etc.)

Use Lots of Functions

The October issue of *Lifelines/The Software Magazine* carried an extensive article on the use of functions and that won't be repeated here, but there are a few points that should be emphasized. First, a function is a glorified subroutine. It is accessed by the program in the same fashion: a CALL and RET(urn) instead of a GOSUB and a RETURN. This process is so efficient that you can save code even if the particular routine is used only twice in the entire program. (You byte savers should also take note that the compiler generates a RET instruction out of the key word FEND so you need not conclude your function with a RETURN statement.)

Admittedly, if the function call requires a number of parameters to be passed, some extra code is going to be used loading them on the stack for passage; but oftentimes these parameters are constants which can be passed directly, thereby avoiding a clutter of assignment statements which often precedes a subroutine call.

The great appeal of the function, however, is that it forces structure and organization on the program, which ultimately has some code saving benefits. This is especially true if you are building overlays and follow some of the procedures I outlined in the October issue of *Lifelines* for moving functions to the root.

String Allocation

String operations are a bit mysterious and you must not let yourself get outsmarted by them. String allocation and garbage collecting are perhaps not the most exciting topics in the world, but at least a nominal understanding of their workings will save you some troubles. The crudest method of string allocation is a fixed allocation method. Under this approach, every string is assigned a permanent string location and enough memory to hold its maximum length. This method essentially doesn't work very well with string intensive programs on microcomputers, because the available string space simply won't support that luxury.[3]

Dynamic allocation is the buzz word associated with the concept of assigning memory space only as needed. Regardless of whether strings are declared, CB-80 does not assign memory space to a string until the first time that string is assigned a value. Then, it is only assigned the space that is needed. Internally, this is handled by the maintenance of 2-byte pointers pointing to the address of the string. The first two bytes at the string location will hold the length of the string and will be followed by the string itself. (The VARPTR and SADD functions return information about these things.)

The allocation is fairly straightforward, but where everything gets hairy is with the de-allocation. There is no dispute with the concept that string space ought to be released when it is no longer needed. The problem is 'how'? The Microsoft approach is to let the garbage accumulate until it becomes a problem, and then invoke a subroutine (while you twiddle your fingers) to make a pass through memory and clear away the garbage. CB-80 by contrast attempts to pick up its garbage as it goes. This method works fairly well, but there are ways that a programmer can trap garbage in memory; and, once it is there, it will stay there forever.

If you have a program that is having a problem running out of memory, one of the first things to do is to examine your memory for wasted space. The FRE function returns the amount of memory that is not being used. The MFRE function returns the amount of memory that is available for use. The difference between the two is waste. If the values returned by these functions are radically different, you had best isolate the cause and do something about it.

I have made no serious effort to document the various ways to load the memory with garbage, but I know of two that I will mention.

Beware of Re-executing DIM Statements

In CB-80, DIM statements are executable, and an array may be re-dimensioned at will. Certain precautions are appropriate, however. The quickest way that I have found to blow up the memory board is to re-execute a DIM statement without first doing some housekeeping.

A DIM statement, by definition, dynamically allocates space for an array. If the same variable has previously been dimensioned those dimensions will be clobbered. Stated another way, a DIM statement generates space for a series of pointers. Assuming that we are talking about a string array, when the particular element is first assigned a value, the string is placed in the dynamic string area and the address is placed in the pointer location associated with that element. Re-dimensioning does not release the string space assigned to the previous array; it merely overwrites the pointer area, thereby leaving the strings in memory forever. At the risk of seeming redundant, I repeat, "You cannot get rid of an array by re-dimensioning it". This merely cuts it loose in the middle of your memory without a paddle.

How to make an array go away

There is, of course, a correct way to recover the space assigned to an array. The process requires the express release of all the array elements prior to the re-dimensioning. As you will recall, CB-80 does not assign a memory location to a string until it has first been assigned a value. It is also true that if a string variable is set equal to another string variable which has not been assigned a memory location, then neither will have a memory location. Thus, if you select a dummy string variable which has never before been used and whip through a FOR-NEXT loop assigning each element in the array this null value you can recover all the dynamic storage area. If you then execute a DIM A$(0), you can get within two bytes of where you were before the array was first used (assuming that A$ was the array name). The form of the assign-

(continued on next page)

ment statement should be '<LET> A$(i%)=NULL$'. Do not use '<LET> A$(i%)=" " '. The problem with the latter statement is that it doesn't release the string; it merely makes it into a string of zero length. It may sound like double talk, but a string of zero length is actually two bytes long (the length pointer, you know). Any string may potentially be 32K in length because this length pointer actually uses the low-order 15 bits of the first two bytes of each string. Since most string operations require work space equal to the size of the string, don't expect to generate very many 32K strings in your 64K memory board.

Another Way To Butcher Your Memory

As mentioned in a recent issue of the *Digital Research News*, you can, if you try hard enough, overpower the dynamic allocation system by churning string variables of ever increasing lengths through memory. CB-80 requires that an entire string be stored in a contiguous location in memory. As a string grows in length, it is automatically relocated to another memory area if it won't fit in the previously assigned area. Each string relocation leaves behind potentially unusable memory. One way to reduce the segmenting is to reduce the number of relocations required. You can stabilize the string area with a pseudo-declaration. This is done by initializing all of the string variables that you intend to use actively with dummy values of a length equal to the maximum anticipated length of that string. The initialization will force all the strings so initialized to find an adequate location right off the bat. My own testing suggests that memory wastage can be reduced by as much as 15% by initializing the strings in advance. I have had some difficulty figuring out just what the worst case situation is for memory hashing. The problem seems minimal where the overall string length is low, because new strings will infill the holes left by the relocations. My testing procedures, which yielded the 15% improvement, involved a small number of strings which were repeatedly concatenated to ever increasing lengths until an OM (out of memory) failure occurred.

Conclusions

In summary, even if you don't understand or program in assembly language, compile your programs frequently using the assembly language listing toggle. Counting instructions is a very good way to get an idea of a place for improvement. If you see a trivial piece of source code that generates a page and a half of output, you know where to start. Complex, deeply-nested statements usually require much less code than simple one function per line coding. The reason for this is that the compiler can optimize nested functions, calculating the least significant one first. The result can then be the input for the next function until the entire statement is worked out. By contrast, if the functions are laid out one at a time, after each calculation the result must be stored and then retrieved for the next calculation.

There is no fixed rule identifying the 'proper' depth to nest functions. Function nesting forces a trade-off between understandability and efficiency. It can be assumed that if your statements are too complex, you will blow up the compiler. Unfortunately, no one, apparently including Digital Research, knows for sure just how complex *too* complex is. I have yet to cause a compiler explosion from an overly complex statement but I understand that it can be done and that Digital Research collects such examples in a file drawer somewhere, with the idea of either eventually documenting the maximum level of complexity possible, or using the data as a basis to justify modifying the compiler to accept even more complex statements.

How much can be saved with careful optimization?

The first question that should be answered before going on an optimization kick is whether it is really necessary. In my own experience, a twenty-percent reduction in code size has not been an unreasonable goal. The problem is not unlike beating dirt out of a blanket: the more you beat, the more you get, but...

Confession is good for the soul

It is said that confession is good for the soul. I don't know if that is so or not, but what follows is designed to discourage some sharp-eyed reader from asking something embarrassing, such as, "Why didn't you apply those rules of optimization to that function that you published in the October issue of *Lifelines/The Software Magazine?"*

The truth is that the 'flasher' function that I published in October is written around code that I have just had around for several years and have used just because it was there; and it worked. (Sharp-eyed readers will also notice a significant similarity between my flasher function and subroutine 345 in the public domain Osborne accounting programs.) By some coincidence, I chose to make it the research vehicle around which this article was written. It was only when I attacked that function with a broom and found that I was able to achieve an astounding 40% reduction in code size, while maintaining the same functionality, that I became inspired to write this article. The revised version appears as Figure 4 (see page 37).

The version as published in October requires a total of 238 bytes of code space and another 40 bytes of data area. The version presented as figure 4 will compile in 128 bytes of code and requires 22 bytes of data area. A two-byte pointer area is also required for each COMMON variable.

The changes

First, note that all possible variables are now integer. The original idea had been to use real variables for the loops because slow speed was the goal of the flash function. Ten thousand interations of an integer loop does as well as 120 iterations of a real loop in a lot less code. Secondly, notice that the redundant code was all abolished.

The third change is also typical of the sort of thing that must occur during optimization sessions. I pitched the entire guts out of the function and came up with a brand new algorithm to accomplish the flashing task.

There is, possibly, a better way than using the MOD function to create a flip-flop, but, if so, I have not had my nose rubbed in it.

A Final Note

If you are a tried and true CBASIC programmer, the MOD function will probably throw you for a loop, as it doesn't exist in CBASIC. It is new in CB-80. The form of the statement is:

MOD(x%,y%)

The value returned is an integer equal to the remainder after the division of x% by y%. In this sense, it works like the Microsoft version which would be: x% mod y%.

For those of you familiar with PL/I-80, a word of caution is in order. The form of the CB-80 statement is identical to the PL/I statement, but it doesn't work like the PL/I statement in that it appears to handle negative numbers differently.

(continued on page 37)

[1]I added the documentation, and, since the library calls are not documented by Digital Research, please regard my descriptions of the library functions as tentative guesses. Identification and documentation of all CB-80 library calls would be most useful to anyone who has the urge to write assembly language external functions for use with CB-80; however, this task is beyond the scope of this article. My description is based merely on casual observation.

[2]Digital Research has indicated an intention to move CB-80 to both the 8086 and the 68000. As of July 1982, work on the 68000 had not really started, for the reason that CB-80 is written in PL/M and Digital Research was still shopping for a PL/M compiler for the 68000. (I wonder if they checked with Lifeboat?)

[3]I may have overstated the case against fixed string allocation schemes a bit here. Fixed allocation schemes gobble memory in the same wasteful way that random access files gobble disk space. If the string length is uniform and predictable, and all of the strings will fit into memory at one time while still leaving a reasonable amount of work space, a fixed allocation method works well. Fixed allocation typically reserves space in the declaration block. Dynamic allocation allocates space only as needed during the runtime sequence. Airline overbooking is an example of dynamic allocation in action. It usually works, but also carries the seeds of failure with it.

Editorial (continued from page 2)

day to point out that The Software Magazine should provide more of a forum for entry level articles, reviews, etc. to provide an opportunity for those now entering the microcomputer realm to learn quickly the buzz words, basic concepts, etc. Kathy, an extremely competent systems analyst, has recently made the decision to extend her expertise to include micros. We've asked her to consider writing a monthly column for The Software Magazine targeted at specifically this area. She'll cover a wide range of subjects with a view towards providing insight to those who have been in search of an "entry point" into this fascinating field.

The Software Magazine has as a fundamental element of its charter an obligation to see to it that the readership is provided with comprehensive treatment of all aspects of the microcomputer software field. With Kathy's help we'll continue to meet that obligation and insure that the light at the end of the tunnel isn't a train.

Meet The First Program Generator That Really Works

Want to sidestep the tedium of coding and debugging hundreds of lines of BASIC? Want to compose sophisticated BASIC programs faster–up to 20 times faster?

Sure you do. So it's time you met The Programmer's Apprentice.

Quite simply, it is THE program development tool. For professionals and novices alike. It lets you generate easy-to-follow, fully debugged, and commented application programs in MBASIC™ or BASIC Compiler™. And here are some reasons it's so popular.

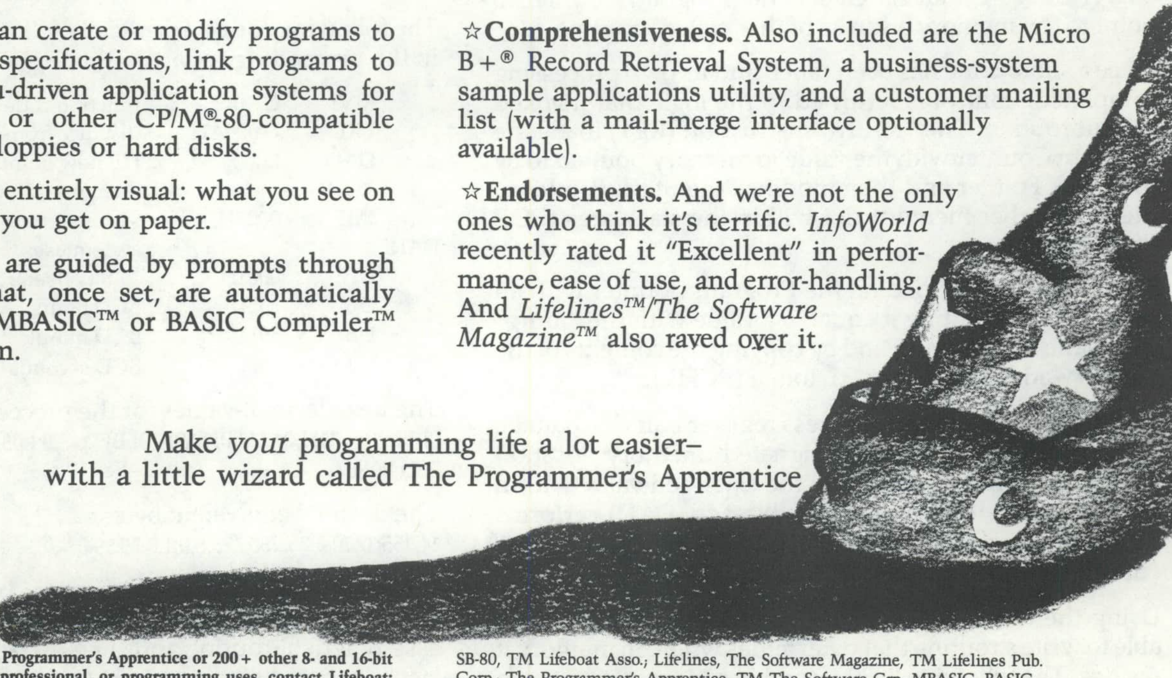
☆**Flexibility.** You can create or modify programs to code to your own specifications, link programs to create entire menu-driven application systems for use with SB-80™ or other CP/M®-80-compatible DOS, with either floppies or hard disks.

☆**Convenience.** It's entirely visual: what you see on the screen is what you get on paper.

☆**Ease-of-Use.** You are guided by prompts through field definitions that, once set, are automatically converted into an MBASIC™ or BASIC Compiler™ source-code program.

☆**Comprehensiveness.** Also included are the Micro B+® Record Retrieval System, a business-system sample applications utility, and a customer mailing list (with a mail-merge interface optionally available).

☆**Endorsements.** And we're not the only ones who think it's terrific. *InfoWorld* recently rated it "Excellent" in performance, ease of use, and error-handling. And *Lifelines™/The Software Magazine™* also raved over it.



Make *your* programming life a lot easier–
with a little wizard called The Programmer's Apprentice

For more information about The Programmer's Apprentice or 200+ other 8- and 16-bit programs for popular, business, professional, or programming uses, contact Lifeboat: 1651 Third Ave., N.Y., N.Y. 10028. Tel.: (212) 860-0300. TWX: 710-581-2524 (LBSOFT NYK). Telex: 640693 (LBSOFT NYK).

SB-80, TM Lifeboat Asso.; Lifelines, The Software Magazine, TM Lifelines Pub. Corp.; The Programmer's Apprentice, TM The Software Grp; MBASIC, BASIC Compiler, TM Microsoft, Inc.; CP/M, reg. TM Digital Res. Inc.; Micro B+, reg. TM Faircom.

Feature

Where Am I? When Did I Get Here?

Steven Fisher

High-level languages allow quick development of readable programs, but they are often too slow for input/output or bit manipulation. Programmers therefore "graft" machine-language routines for the time-critical portions of their high-level code. Since it is a hassle to figure out just where such a routine will reside in memory, it helps if such subroutines are location-independent and can figure out where they end up being put. This article explains how to make a subroutine answer the question 'Where am I?", fetching the system date and time while doing so.

There is a sixteen-bit register (counter) that tells the 8080/Z80 microprocessor where to look in memory for the next computer instruction. This is called the Program Counter (PC). Another sixteen-bit register points to the memory location used by the microprocessor for storing information. The stored data is rather like a stack of dishes in that the last dish placed on the stack is the first dish you take off the stack. This is called 'Last In, First Out" (LIFO) buffer management, and the top of the memory stack is pointed to by the Stack Pointer (SP). The "PUSH" instruction copies 16 bits of register information onto the top of the stack while "POP" removes it into a 16-bit register. A special-purpose instruction exchanges the information at the top of the stack with the contents of the memory-address register (HL); this instruction is "XTHL".

When the microprocessor performs a "CALL" instruction, it saves the current Program Counter value in the memory specified by the Stack Pointer, modifies the Stack Pointer to indicate the next memory slot within the stack area (PUSH PC), and then sets the Program Counter to point to the memory address of the "called" routine.

Once a subroutine has been called and its own processing completed, control is returned to the logic that invoked the subroutine. The "return" instruction (RET) loads the Program Counter with the value in memory pointed to by the Stack Pointer and then modifies the Stack Pointer to the next earlier memory slot within the stack area (POP PC).

Two other ways of loading the Program Counter with an address are by setting it equal to a value with the "jump" command (JMP hhhh), and by copying the contents of the memory-address register HL into it (PCHL).

The sixteen-bit memory-address register pair (HL) can be loaded directly from a designated memory location (LHLD), or immediately from the extended value within the instruction itself (LXI). A double-add (DAD) performs 16-bit addition to the memory address (HL), using the current contents of a register pair (B,D,H,SP).

Using the computer instructions just discussed, we are able to write a routine that determines where in memory it resides. The following 8080 assembly code returns its own location in the HL register pair:

```
FINDME: LHLD  0100H        ; get transient program start code
        PUSH  H            ; put it on the stack for later
        LXI   H,0E9E1H     ; LH="POP H ! PCHL"
        SHLD  0100H        ; put temporary routine in memory
        CALL  0100H        ; find out where next address is
RTRN:   XTHL               ; put this addr on stack, get old
        SHLD  0100H        ; restore program start code
        POP   H            ; HL points to RTRN address
        LXI   D,0009H      ; distance from RTRN to HERE
        DAD   D            ; HL now points to HERE
HERE:                      ; put your code here
```

The hexadecimal byte-values of the logic are:
2AH,00H,01H,0E5H,21H,0E1H,0E9H,22H,00H,01H,0CDH
00H,01H,0E3H,22H,00H,01H,0E1H,11H,09H,00H,19H

For those BASIC programmers, here it is in decimal:
42,0,1,229,33,225,233,34,0,1,205
0,1,227,34,0,1,225,17,9,0,25

Now that we have a routine that can find out where it is in memory, we can ask the system "What time is it?". Digital Research's MP/M operating system will fill a specified memory area with the date and time when you put a decimal 155 into register C prior to "calling" memory location 5. The date-time area is structured such that first there is a sixteen-bit binary number representing how many days have elapsed since December 31, 1977 (yes, Virginia, it is tacky). Next are three binary-coded-decimal (BCD) bytes for the hour, minute, and second using 24-hour notation (one PM is hour 13). The register pair DE (D) is used to point to the date-time area.

The following code immediately follows the label "HERE" in the preceeding code:

```
        MVI   C,9BH        ; function code to get date/time
        LXI   D,000AH      ; distance from HERE to DATE
        DAD   D            ; HL now points to DATE
        XCHG               ; exchange HL with DE
        JMP   0005H        ; system returns to calling program
DATE:                      ; system uses the following areas
        DW    0001H        ; 16-bit offset from 12/31/77
        DB    23H          ; BCD hours
        DB    59H          ; BCD minutes
        DB    00H          ; BCD seconds
```

The hexadecimal values for the preceeding are:
0EH,9BH,11H,0AH,00H,19H,0EBH,0C3H,05H,00H,01H,00H,23H,59H,00H

The decimal equivalent bytes:
14,155,17,10,0,25,235,195,5,0,1,0,35,89,0

If your program contains a string of bytes that reflect the values presented here, you would execute the string as an assembler subroutine and then extract the data from the last five bytes of the string. Gordon Eubank's CBASIC would do it thus:

```
getdate$ = \
    chr$(42)  + chr$(0)   + chr$(1)   + \
    chr$(229) + \
    chr$(33)  + chr$(225) + chr$(233) + \
    chr$(34)  + chr$(0)   + chr$(1)   + \
    chr$(205) + chr$(0)   + chr$(1)   + \
    chr$(227) + \
    chr$(34)  + chr$(0)   + chr$(1)   + \
    chr$(225) + \
    chr$(17)  + chr$(9)   + chr$(0)   + \
    chr$(25)  + \
    chr$(14)  + chr$(155) + \
    chr$(17)  + chr$(10)  + chr$(0)   + \
    chr$(25)  + \
    chr$(235) + \
    chr$(195) + chr$(5)   + chr$(0)   + \
    chr$(1)   + chr$(0)   + chr$(35)  + chr$(89)  + chr$(0)
datertn% = sadd(getdate$)       rem--- point to string
datertn% = datertn% + 1         rem--- skip string length byte
rem--- if CB80, previous line is "datertn% = datertn% + 2"
call datertn%                   rem--- run assembler routine
datertn% = datertn% + 32        rem--- least significant date
                                byte
dayoffset% = peek(datertn%)
datertn% = datertn% + 1         rem--- most significant date
                                byte
dayoffset% = dayoffset% + (256 * peek(datertn%))
year% = 1978
while dayoffset% > 1461         rem--- count down four years
    year% = year% + 4
    dayoffset% = dayoffset% - 1461
wend
if dayoffset% > 365 then \
    year% = year% + 1 :\
    dayoffset% = dayoffset% - 365
if dayoffset% > 365 then \
    year% = year% + 1 :\
    dayoffset% = dayoffset% - 365
if dayoffset% > 366 then \
    year% = year% + 1 :\
    dayoffset% = dayoffset% - 366
leapyear% = (0 = year% and 3)       rem--- 0 or -1
rem--- jul-to-cal algorithm by Steven Fisher CDP
if dayoffset% > (59 + (1 and leapyear%)) then \
    month% = \
        int%(float(dayoffset% + 32)/30.57) \
else \
    month% = 2 + (dayoffset% <= 31)
day% = dayoffset% - int%((30.57 * float(month%)) - 30.0 - \
    ((2 + leapyear%) and (month% > 2)))
 datertn% = datertn% + 1
hour% = \
    (0fh and peek(datertn%)) + \
    (10 * (peek(datertn%)/16))
if hour% >= 12 then \
    hour% = hour% - 12 :\
    ampm$ = "PM" \
else \
    ampm$ = "AM"
if hour% = 0 then \
    hour% = 12
datertn% = datertn% + 1
minute% = \
    (0fh and peek(datertn%)) + \
    (10 * (peek(datertn%)/16))
datertn% = datertn% + 1
second% = \
    (0fh and peek(datertn%)) + \
    (10 * (peek(datertn%)/16))
day$ = right$("00" + right$(str$(day%),2),2)
year$ = right$(str$(year%),2)
minute$ = right$("00" + right$(str$(minute%),2),2)
second$ = right$('00" + right$(str$(second%),2),2)
print using "##\/&\/& ##:&:& &"; \
    month%, day$, year$, hour%, minute$, second$, ampm$
```

Equivalent functions exist in other BASIC dialects. CBASIC had BCD math and structured code before MBASIC, and Digital Research no longer charges run-time license fees for CB80, so I have never bothered to become familiar with MicroSoft's version. You'll have to do MBASIC on your own.

MicroMoneymaker's Forum

Charles E. Sherman

Dealing With CP/M's Directory Bottleneck

Quite often your job as a consultant is to get the right hardware and software set up and working to suit your client's needs. There's a wide range of high-powered, highly sophisticated products to choose from. Lots of microcomputer hardware is powerful, professional, and reliable, and the CP/M family of off-the-shelf software offers programs of amazing power and sophistication. Hard disks and high-density floppies permit the convenient storage of hundreds or thousands of files. Yet for all the flash and dazzle, for all the capability and power, your client's hundreds or thousands of files must still be manipulated in a CP/M directory and filing system that hasn't changed much since micros were strictly for pioneers and hobbyists.

This situation reminds me of Gahan Wilson's cartoon of a giant aircraft factory where jet fighters are being turned out by the hundreds. High up in the rafters is a tiny, dark room where a wrinkled little old lady sits bent over and squinting through her rimless spectacles as she laboriously sews buttons on seat cushions by hand. Two anxious looking executives are hovering over her shoulder, and one says something like, 'You're holding us up again, Mom. Can't you sew any faster?'

Over a period of time, your typical client will have to anticipate that hundreds of files, at least, will be made and stored by various users, maybe all on one hard disk. Somewhere, way down the line, the client or the employees are going to have to root through screens full of filenames, all reading something like "PURCHMEM.D12," and figure how that's related to or different from "MEMOGRP.RPT." Or maybe Andy has quit in a snit and someone has to figure out what's where in that box of disks on his desk. And, say, can anyone remember the filename for that prospectus Anne made up last year? And now that our supplier is acting up, we've got to find that one letter where we specified quality and terms - it's somewhere between SPPLIER3.LTR and SPLIER27.LTR.

You can probably make up even better scenarios, but as a consultant you also have to solve such problems. It is, or should be, part of your job to recommend programs and procedures for the computer work flow. How will your clients identify, store, sort, and retrieve all the files their various users accumulate over the years? How will they identify the files which can be eliminated in order to liberate disk space? How will they identify the few files among many which have been modified and which therefore need to be backed up? And so forth.

For some clients and users the directory and file handling bottleneck under CP/M can become acute, yet the available CP/M software solutions have been something between nonexistent and fragmentary. Standing over there gloating and rubbing their hands are the purveyors of dedicated systems like Lanier and CPT which offer effective directories with keyword descriptions for each file and multiple sorts. More and more, it is only their good directories and elaborate support services which distinguish the dedicated systems from the micros. Vector Graphic's word processor, Memorite, also has an excellent directory. In the CP/M family, the word processors Benchmark and Superwriter have enhanced directory functions. There are a few directory aids in CPMUG, but none which I would put into the average client's business environment. In 1981, Advanced Micro Techniques of Foster City, CA, released an elaborate Library Data Base filing system called *MicroLIB* which is very powerful, but which requires the user to manually check files into and out of the library. It has some powerful features which we'll go into later, but it isn't what I've always wanted because it isn't anything like automatic.

For a long time I have been trying to nag some programmer friends into writing a CP/M directory and file-handling program which would satisfy my fantasies. I wanted it to overlay the CP/M operating system so it would work automatically. Whenever a file is created, the program would query if you wish to add additional directory information, and if so, it would put up a form into which you enter a key-word description. It should automatically enter the date if you have a clock, or take it manually if you do not. You should be able to search and sort through files by any combination of known information, including key words, date, user, and wildcards. Well, my team waited too long, and someone else has finally released a program which does all that and much more. Last fall MicroFusion of La Jolla, CA released *Trakmaster*, priced at $150. Since this is the first commercial program of its kind that I've been able to find, I thought you should know about it.

Trakmaster

When you first install the 26K *Trakmaster* (hereafter called TM) on any disk, you assign that disk an ID name and description, and TM creates its own directory file on that disk. You can modify CP/M to make TM autostart, otherwise you will have to type TM after each powerup or cold boot to get it going. On each fresh startup, TM queries for the user's name and the date. It will recall whatever name and date you give it until it is changed, so if neither item needs changing, you can pass over the query and go back to CP/M. Any redundant appearances of this interruption are little bother, as it only happens when you cold boot.

The TM directory keeps a range of information on each file which includes the disk name, user number, CP/M filename, application (8 characters), key words description (32 characters), user name, date created, last update, CP/M flags, and file size. This information can be sent to screen or printer, and can be displayed either in full, as in Figure 1, or partially, as in Figure 2. When the full list is sent to screen or printed at less than 132 characters, each line of the full display will occupy two lines.

All information on each file is automatically assigned by TM except for the application and description, which it asks the user to supply. When TM is on, it automatically monitors the CP/M directory, and whenever you save a file it will query you for two items of information about the file: application and description. If that information has already been entered, you can waive through the query and go back to CP/M.

You can search the TM directory either from TM or from CP/M. TM allows you to search along the normal CP/M naming and wildcard conventions, and also to search according to parameters in any of the file information categories illustrated in Figure 1. For example, you can ask for a display of all files created by John in an accounting application between 10/01/79 and 09/30/82 which are in reference to the Adams company, and so forth.

TM will keep a master file of all your TM directories, thus enabling you to have rapid access to all files even in very large disk libraries. You just copy all disk directories into a master directory file, and update either routinely or when reasonable to do so. TM will search all files in the master directory in just the same way as described above for single directory searches.

The directory functions are benefit enough, but *Trakmaster* also does wonderful things for your copy and backup operations. You can direct TM to copy files according to parameters set in any of the file information categories, just in the same way as described for directory searches. Thus, you can copy off a hard disk just those files made by Sue between any two dates about the Zenith Project. The powerful copy features become especially useful and important when backing up files from a crowded disk. You needn't take the time to backup all files every time, but instead you can merely instruct TM to backup those files which have been updated since the last backup.

Trakmaster is a magnificent step in the right direction, and as far as I know, it is the only commercially available program of its kind. *Trakmaster*'s design goals are clever and quite well thought out, but the implementation is only ordinary. It does everything it sets out to do, but not necessarily as smoothly or quickly as one would wish. It should, but does not, have provisions for taking the date from a computer clock. The screen and menu operations are not bad, but also not sophisticated. The major weakness is that TM's disk operations slow things down considerably whenever a file is saved. Using my super-fast Godbout/Compupro Disk 1 controller, it takes approximately 34 seconds to wind down out of Magic Wand when no new file information is added. This is about 30 seconds longer than it takes without TM butting in and that's too much. It will be even worse with the average disk controller. Because TM queries every save for application and description information, the 30 or more seconds of slow-down becomes annoying, and especially so when you have no new information to add. TM takes just as long to wind down when you have no news as when you put new information in. My final critical remark is for the documentation, which is merely adequate. Like the program, it does what needs doing, but that's all. Fortunately, it is just good enough to pass muster.

In spite of my criticisms, I think *Trakmaster* is a cleverly conceived and very much needed program. It could use some polishing, and I suspect that it will get it if the boss ever gets the time. Apart from being slow on the file saving operation, TM does do everything it sets out to accomplish, and that's a lot! Until something better comes along I have to recommend this one strongly. It is especially appropriate for hard disk systems, and it may be just the thing for those of your clients, whatever the size of their media, who will need to store and manipulate numerous files.

I get the impression that Microfusion is a one-man enterprise. I gather that the one man, who seems sincere and dedicated, is finding it to be a bigger job than he had ever anticipated to get his baby to market. It may be that his entrepreneurial experiences will be worth hearing about, if he'd care to share them, so I shall be checking it out on my annual Xmas trip South.

MicroLIB

Just a brief note about this one. *MicroLIB* is an elaborate directory data base which has some pass-word and encryption features which may be extremely useful in certain specialized applications. Files must be manually entered into the library, and they must also be checked out and replaced when they are worked upon. The MPUT command is a utility which puts files into the library, and the MGET command checks them out. To me, this means that the program is well suited for very high volume applications, where library maintenance can be routinized and afforded. Such large scale operations might very well want password protection for certain files, and even encryption for very private ones. Raish Enterprises, the authors of Quickey (see the December, 1982 issue), are also consultants to high volume users and they find *MicroLIB* to be very useful. *MicroLIB* is $295.

More Returns From The Readers

Trends detected in the early returns from readers (reported in the November, 1982 issue) have held up all along. Most readers are consultants of one kind or another, and only a few are entrepreneurs, although quite a lot more would like to be. A few readers requested reviews of business and financial software, and a couple wanted to be told how to get and stay rich. However, the majority of those who wrote said they wanted this column to focus on issues and reviews of software relevant to consultants and the electronic cottage industries. So be it.

Previews

Time for a trip to the hot ideas department. You folks haven't told me enough about your dreams, so I'll just have to tell you about mine. *Trakmaster* already ran all over one of them, and *Fancy Font* has dented the decorative features of another, but I still have a few virgin ideas left. If that gets boring, there are more program reviews in the offing. If things get desperate, I may fall back on my ancient legal lore and deliver some sage and practical advice about starting up a business.

(continued on next page)

Figure 1: Trakmaster full Directory Display on screen or 80 col. printer, each entry takes two lines

```
MicroFusion                                            Directory                                                 Thr 10-21-82 Page 1
Diskname   Un  D: Filename   .Ext  Application  Description                     Username  Create    Update    Flags    Size
HARDDISK   7   B: ACCTPABL   .CMD  Dbase        Account Payable command file    Bill      10-21-82  10-21-82  R/W DIR   8K
HARDDISK   7   B: ACCTPABL   .FRM  Dbase        Account payable form file       Bill      10-21-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: ADDRCHG    .LTR  Wordproc     Ltr Example for Mailing         Bill      08-19-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: ART        .MEM  Dbase        Article memory file             Bill      10-21-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: ARTICLE    .CMD  Dbase        Article command file            Bill      10-21-82  10-21-82  R/W DIR  16K
HARDDISK   7   B: BUG        .BAS  Basic        BASIC Test program "Bug"        Bill      08-24-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: CALL       .BAS  Basic        Autodial Pgm source             Bill      09-28-82  10-21-82  R/W DIR   8K
HARDDISK   7   B: CR-UNION   .ADS  Wordproc     C/R Union Ads                   Bill      10-19-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: DND        .BAS  Basic        Dungeon & Dragons game          ALL       08-23-82  10-21-82  R/W DIR  28K
HARDDISK   7   B: DNDMENU    .BAS  Basic        Dungeon & Dragons Menu          ALL       08-23-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: EXAMPLE    .TXT  Wordproc     Wordproc example text           ALL       08-01-82  10-21-82  R/W DIR  16K
HARDDISK   7   B: HARRAL     .INV  Invoice      Invoice Horral lost equipment   Bill      08-01-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: INSBANK    .LTR  Wordproc     Ltr to LJBT re Stop Maccibees   Bill      08-11-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: LEE        .INV  Invoice      Lee Equipment Invoice           Bill      08-01-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: MOM        .LTR  Wordproc     Letter to Mother re kittens     Bill      08-15-82  10-21-82  R/W DIR  16K
HARDDISK   7   B: MOM2       .LTR  Wordproc     Letter to Mom re wedding        Bill      10-01-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: MONOPOLY   .BAS  Basic        Monopoly game                   PETER     08-22-82  10-21-82  R/W DIR  28K
HARDDISK   7   B: PACMAN     .COM  System       Packman game (all terminals)    ALL       08-23-82  10-21-82  R/W DIR  20K
HARDDISK   7   B: PINOEVAN   .LTR  Wordproc     Ltr to Evan re Brown Account    Bill      08-01-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: REBASTOK   .LTR  Wordproc     Ltr To REBA re B&B Stock        Bill      08-23-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: RISETWO    .BAS  Basic        C/R Raccount source Pgr         Bill      08-12-82  10-21-82  R/W DIR  20K
HARDDISK   7   B: SETMAR     .BAS  Basic        Set margin on printer Pgm       ALL       08-23-82  10-21-82  R/W DIR   4K
HARDDISK   7   B: TMMANUAL   .FRQ  Wordproc     FM Manual Words to define       Bill      08-18-82  10-21-82  R/W DIR   8K
```

Figure 2: Trakmaster Partial Directory Display

```
MicroFusion                          Directory                          Thr 10-21-82 Page 1
   Diskname     Un   D: Filename    .Ext   Application   Description
   HARDDISK     7    B: ACCTPABL    .CMD   Dbase         Account Payable command file
   HARDDISK     7    B: ACCTPABL    .FRM   Dbase         Account payable form file
   HARDDISK     7    B: ADDRCHG     .LTR   Wordproc      Ltr Example for Mailing
   HARDDISK     7    B: ART         .MEM   Dbase         Article memory file
   HARDDISK     7    B: ARTICLE     .CMD   Dbase         Article command file
   HARDDISK     7    B: BUG         .BAS   Basic         BASIC Test program "Bug"
   HARDDISK     7    B: CALL        .BAS   Basic         Autodial Pgm source
   HARDDISK     7    B: CR-UNION    .ADS   Wordproc      C/R Union Ads
   HARDDISK     7    B: DND         .BAS   Basic         Dungeon & Dragons game
   HARDDISK     7    B: DNDMENU     .BAS   Basic         Dungeon & Dragons Menu
   HARDDISK     7    B: EXAMPLE     .TXT   Wordproc      Wordproc example text
   HARDDISK     7    B: HARRAL      .INV   Invoice       Invoice Horral lost equipment
   HARDDISK     7    B: INSBANK     .LTR   Wordproc      Ltr to LJBT re Stop Maccibees
   HARDDISK     7    B: LEE         .INV   Invoice       Lee Equipment Invoice
   HARDDISK     7    B: MOM         .LTR   Wordproc      Letter to Mother re kittens
   HARDDISK     7    B: MOM2        .LTR   Wordproc      Letter to Mom re wedding
   HARDDISK     7    B: MONOPOLY    .BAS   Basic         Monopoly game
   HARDDISK     7    B: PACMAN      .COM   System        Packman game (all terminals)
   HARDDISK     7    B: PINOEVAN    .LTR   Wordproc      Ltr to Evan re Brown Account
   HARDDISK     7    B: REBASTOK    .LTR   Wordproc      Ltr To REBA re B&B Stock
   HARDDISK     7    B: RISETWO     .BAS   Basic         C/R Raccount source Pgr
   HARDDISK     7    B: SETMAR      .BAS   Basic         Set margin on printer Pgm
   HARDDISK     7    B: TMMANUAL    .FRQ   Wordproc      FM Manual Words to define
```

Feature

The Z80 Instruction Set – Shakers & Movers

Kim West DeWindt

This month I am going to concentrate on a most useful group of instructions - the Load group. Various learned studies have calculated that load, or move, instructions account for fifty percent of all of the instructions used in a typical assembly program. It seems that programmers spend a lot of time moving information into a location where it can be manipulated.

In the following discussion, the term register refers exclusively to one of the eight bit registers (A,B,C,D,E,H, and L). The term register pair refers to one of the sixteen bit registers (BC, DE, and HL). The instruction stream is the flow of information that comes into the processor via the address in the program counter (PC). In the normal program flow bytes are read one at a time, and the PC is incremented, by one, after each instruction fetch. (Obviously jumps and calls alter this sequential flow, but we are not going to discuss them this month).

The Zilog Load mnemonic (LD) corresponds to a variety of Intel mnemonics:

Zilog	Intel
Load registers or memory	Move (MOV) reg. or memory
Load immediate data	Move immediate (MVI, LXI)
Load accumulator, direct	LDA and STA
Load accumulator, indirect	LDAX, STAX
Load reg. pair, indirect	LHLD, SHLD (note HL only)
Load index reg. (IX or IY)	- no equivalent -
Load SP from HL, IX or IY	SPHL

This section will also discuss exchange operations, push and pop, and the block move instructions. Although these are not officially in the load group, they do transfer data from one location to another. The relevant Z80 mnemonics and their Intel counterparts are:

Zilog	Intel
EX (exchange) - 4 flavors	XCHG and XTHL
PUSH reg. pair, IX, IY	PUSH reg. pair
POP reg. pair, IX, IY	POP reg. pair
Block moves	- no equivalent -

As I noted in a previous section, Zilog uses one mnemonic, LD, for all normal load instructions. (Due to Intel's existing opcode map, new Z80 instructions, like block move and the index register instructions, require two byte opcodes. Unique mnemonics emphasize this change in the opcode template.) The addressing mode (immediate, indirect, etc.) is specified by the operand field. The Z80 load instruction has more addressing modes than the 8080 move instruction. Additionally, some of the 8080's extended (16 bit) operations, limited to the HL register pair, are expanded to include DE and BC as valid operands. For example, you can now load the register pairs BC and DE from memory. In Intel mnemonics this capability would be expressed as LBCD (Load BC Double) or SDED (Store DE Double).

I am not going to go into the details of the Intel compatible instructions. By now, you all know that MVI A,55 puts the value 55 into the A register. I will list the Zilog opcode mnemonics, what they cause the processor to do, and an example of the Intel equivalent. That should keep everything nice and neat, providing a simple cross reference between Intel and Zilog listings. Instructions that are unique to the Z80, and that provide improvements over the 8080 abilities, will be explained more carefully. Coding examples, for these and other Z80 instructions, will appear in a later section of this tutorial.

The simplest load instructions move data in and amongst the internal eight bit registers. No outside addresses are generated; an internal three bit code selects two of the eight registers. Data is moved from the source register into the destination register. At the end of the instruction cycle, the contents of the two registers are equal. The contents of the destination register are destroyed, the source register is unchanged. The Zilog expression for this is:

```
LD    A,B        ;Load A with the contents of B
```

Intel's version:

```
MOV   A,B        ;Move the contents of B into A
```

Note that the use of the word load to describe this operation allows a written description of the instruction to list the registers in the same order in which they appear in the operand field.

The only double register addressing that exists allows the stack pointer to be loaded from HL, or one of the index registers. The 8080 can load the stack pointer only from HL (SPHL). Zilog shows this as follows:

```
LD    SP,HL      ;Load the stack pointer with the
                  contents of HL
```

Zilog has two special forms of the register load. One transfers data between the accumulator and the refresh register (counter). The other allows the programmer to read from and write to the Interrupt vector register, also via the accumulator. The 8080 does not have these special purpose registers, so it does not have the corresponding opcodes. In Z80 assembler code they appear as follows:

```
LD    A,I        ;Load A with the contents of the
                  Interrupt vector

LD    R,A        ;Load the refresh register (7 bits)
                  with the contents of A
```

Another simple data transfer moves a user-specified value into the addressed register (known as the move immediate by Intel buffs). One byte of data, nestled in the instruction stream, is written into the specified eight bit register. There are no internal machinations required to generate a data pointer. After reading the opcode, the program counter (PC) is incremented so that it now points to the next location in memory, the implanted data. Another memory read cycle fetches this data, and it is

(continued on next page)

relocated into a register. The data is not changed in its memory location, the previous contents of the register are lost. The proper expression for this instruction is:

```
LD     D,23            ;Load D with the value 23 (decimal)
```

This instruction can also be used to load any one of the Z80's register pairs, the stack pointer (SP), or either one of the index registers (It is up to the user to initialize, i.e. load, the index registers before using them. Their content is unknown after reset.):

```
LD     BC,3556         ;Load B with 35, and C with 56
LD     SP,0FFFFH       ;Initialize the stack pointer to the
                        top of memory
```

In Intel mnemonics, the eight bit version of this instruction is:

```
MVI    D,23            ;Move the value 23 (located in
                        memory, immediately after the
                        opcode) into D
```

Their mnemonic for the sixteen bit version is different:

```
MXI    BC,3556         ;Move the extended (16 bit) value
                        3556 into the register pair BC
```

Another simple form of addressing, called direct addressing, allows you to implant a fixed address in the instruction stream. Two bytes of data provide the sixteen address bits of a source or destination in memory. With the Z80, a direct address in the operand field can be paired with the accumulator, any of the normal register pairs, the stack pointer, or the index registers.The 8080 is limited to using the accumlulator, HL, and SP. For a Z80 example:

```
LD     (4955),A        ;Load location 4955 with the
                        contents of the accumulator
```

Normally, (4955) would not appear in a program listing. Instead, you would have a label (ex: Pferd) which is the label at address 4995. The assembler would fill in the appropriate numbers and the instruction would look like:

```
LD     Pferd,A
```

The Store A and Load A instructions are the Intel equivalents:

```
LDAX B                 ;The X implies the use of the BC
                        register as a pointer, rather than the
                        solo B reg.
STAX Pferd             ;Store A's contents at Pferd
```

Zilog's sixteen bit version of this instruction type looks very similar to their eight bit version:

```
LD     BC,(1248)       ;Load BC with two bytes, starting at
                        location 1248
LD     Pferd,DE        ;Load memory location Pferd, with
                        the sixteen bits in DE
```

The 8080 can only use direct addressing with the register pair HL. These are the familiar load and store double commands:

```
LHLD Pferd             ;Load HL from Pferd
SHLD Pferd             ;Store HL's contents at Pferd
```

Next in line is indirect addressing. This variation uses the contents of the HL register pair as a pointer into memory. The pointer can be directed at either the source or the destination of the load operation. The other half of the operand must be one of the eight bit registers, like this:

```
LD     A,(HL)          ;Load A with a value located in
                        memory. The register pair HL con-
                        tains the proper address.
LD     (HL),C          ;Load the addressed location (HL
                        contains the address) with the con-
                        tents of C
```

As in all load instructions, the contents of the source location (memory or register) are unchanged; the pre-instruction contents of the destination are overwritten. HL contains a complete sixteen bit address; no additional calculation is necessary before the address is shipped out to the address pins.

In the Intel instruction set this type of move operation is called a register to memory transfer, rather than register indirect addressing. The letter M replaces the Zilog notation (HL). The net result is that M stands for a the memory location pointed to by the contents of HL:

```
    MOV  A,M
or: MOV  M,C           ;Just don't forget to put the proper
                        address into HL.
```

A subset of this type of operation requires that the accumulator (A) be either the source or the operand. If this is the case, the address can be contained in HL, BC, or DE. This version appears as follows:

```
LD     A,(BC)          ;The source address is in BC
LD     (DE),A          ;Load the destination (address in
                        DE) with the contents of A
```

A more complicated form of address generation, indexed addressing, is unique to the Z80. (Actually, indexed addressing is unique only in this comparison of the Z80 and 8080 instruction sets. Almost all newer processors provide some form of indexed addressing.) In this mode, the index register (IX or IY) contains a base address. An eight bit offset is added to this base, the composite answer is a pointer to the source or destination. Any eight bit register can be the other half of the operand. The eight bit offset is data that is included in the instruction stream. Note that because it is necessary to fetch the offset and calculate the address, this type of instruction takes a little bit longer to process. (19 T states instead of the 9 T states required for the more mundane indirect addressing. A T state is one clock cycle - for the 4 megahertz Z80A, one T state is 250 nanoseconds long.) Also note that an offset is REQUIRED, even if the offset is 0. This is a two byte opcode, and there is no 8080 corrollary. The assembler format is:

```
LD     B,(IX+3)        ;Add 3 to the contents of IX and use
                        that value as an address pointer.
                        Load B with the value at that loca-
                        tion.
LD     (IY+0),88       ;Load the memory location, ad-
                        dressed by IY, with 88 – the source
                        could have been one of the eight
                        registers
```

Leaving the index registers for a moment, let us talk about swapping information. The exchange instructions allow the programmer to transfer information between registers without destroying the contents of either register. Two register pairs appear in the operand field, each can be considered a source and a destination. The Z80 uses special exchange instructions to get information into and out of the prime registers. This is the only way to communicate with these prime registers. A single command swaps the contents of all of the prime register pairs with the contents of the normal register pairs - very handy in interrupt handling routines. Later in this tutorial, I will give an example of a simple interrupt handling technique (context switching) which uses this instruction. The Z80 also has the 8080 style exchange instruction which swaps the DE and HL registers pairs. That mnemonic is:

```
    EX    DE,HL        ;swaps the contents of HL with the
                        contents of DE
```

This is the same function as Intel's:

```
    XCHG
```

The two exhange functions which involve the prime registers are:

```
    EX    AF,AF'       ;swap the contents of the prime flag
                        and A registers with the normal A
                        and F regs.
    EXX                ;exchange BC, DE, and HL with
                        BC', DE', and, HL'
```

Although the user can not manipulate the prime registers directly, one state of the Z80 operating environment can be swiftly saved. No need to push registers on and off the stack, trying to remember which byte goes into which register.

The exchange instruction can also be used with a limited type of indirect addressing. The top of the stack, pointed to by SP, can be exchanged with the contents of HL, IX, or IY. Swapping the TOS (Top Of Stack) with HL is Intel's:

```
    SPHL               ;exchange TOS with HL
```

Zilog mnemonics for that operation look like this:

```
    EX    (SP),HL      ;note that the notation (SP) is con-
                        sistent with other uses of indirect
                        addressing.
```

Or, to load one of the index registers, just modify the operand field to address the appropriate location:

```
    EX    (SP),IX      ;IY is also a valid operand
```

NOTE: It should be apparent that Zilog has made a concerted effort to standardize their opcodes and addressing notation. This simplifies the task of a programmer who has to pick up a listing, recognize the opcode, and then determine what type of data is being shuffled around, where, and how it is being addressed.

Transferring data through the stack is another way of getting data where you want it. Nothing fancy here, just the good old push and pop instructions. The only addition, evident in the operand field, allows the special index registers to be popped and pushed - just like everybody else:

```
    PUSH  AF           ;Push A and F onto the stack
    POP   IX           ;in this sequence, the contents of A
                        and F would end up in the IX regis-
                        ter.
```

Remember that F is the Zilog notation for the flag register. AF is the equivalent of Intel's PSW (Program Status Word). Other Zilog operands (for PUSH and POP) are BC, DE, HL and IY.

Last, and most certainly not least, we come to the block move instructions. The mnemonic base is LD (this is a form of the Load command). Letters are added to indicate the direction of the block move (the pointers are incremented or decremented), and to indicate that the instruction occurs once, or that it is repeated.

The block move instruction assumes that the register pair HL points to the start of the block to be moved (the source). DE must point to the future location of that block (the destination). BC contains a number equal to the length of the block that is to be moved.

The letter I indicates that HL and DE are incremented after each move. HL and DE are decremented if the mnemonic includes the letter D. BC, which is the count register pair, is decremented after every move within the series. The letter R means that the instruction is performed until BC is equal to zero. Even if the instruction does not loop, HL and DE are bumped (up or down) and BC is decremented.

There are four forms of the instructions. They are:

```
    LDI        ;Load, moving data from the ad-
                dress contained in HL to the ad-
                dress in DE. Increment HL and DE,
                decrement BC. Do not repeat
    LDIR       ;Do the same thing as the above in-
                struction, then repeat until BC is
                equal to zero.
    LDD        ;Load, from (HL) to (DE), but this
                decrement HL and DE. Decrement
                BC. Do not repeat
    LDDR       ;Load, decrement pointers, repeat
                until BC is equal to BC.
```

You can see that it is a very powerful instruction. In addition to compacting code, it can provide an increase in speed, depending on the type of operation. Later, in the coding examples, I will discuss some of the uses for this instruction.

That about wraps up data movement (excluding Input/Output). See you next month.

Call For Bug Reports

Help us keep you informed! Lifelines/The Software Magazine welcomes bug reports from readers. A product with a documented bug can still be valuable to the informed user.

We also welcome tips & techniques and will pay you $50 if we publish yours. Unpublished tips will only be returned if they are accompanied by a self-addressed envelope.

Feature

CP/M Interfaces to the Human Being

Bob Kowitt

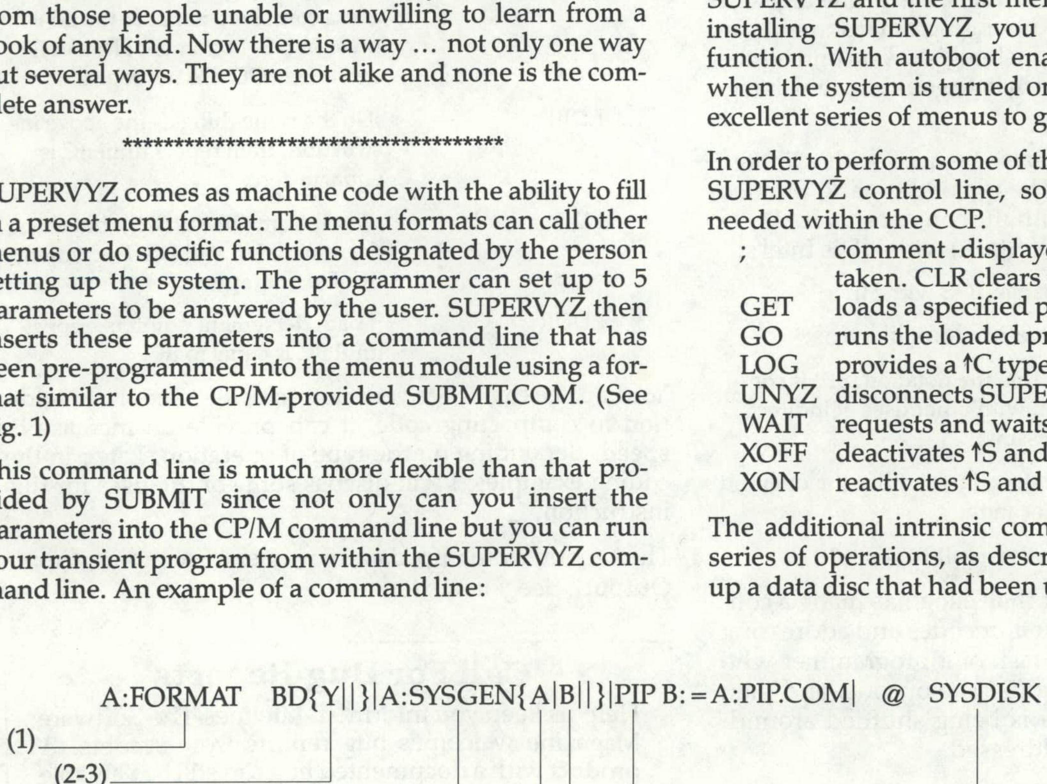
"I was reading the manual for about fifteen minutes before I realized that the book was upside-down." That's what my client told me after the first session spent reading the Digital Research manuals for CP/M. I had heard it before but not in quite such a picturesque way. One of the constant problems accruing to dealers providing CP/M to their customers has been the difficulty of understanding the usage of the CP/M built-in commands as well as the transient programs that are supplied by Digital Research. This is not to denigrate the operating system itself, but, these manuals were not written for the novice.

This problem has forced customers to call us, at all times of the day and night, to find out whether to "PIP a:=b:....." or "PIP B:=A:....."; "How do I rename my data file ?" ; "What do I back up when I want to back up.....and how ?" Add the multiplicity of questions of your choice and you will appreciate the amount of time taken on the phone teaching CP/M. I have, in self-defense, given copies of one of the books designed to make CP/M more easily understood. However, there has been no way to divorce myself from those people unable or unwilling to learn from a book of any kind. Now there is a way ... not only one way but several ways. They are not alike and none is the complete answer.

****

SUPERVYZ comes as machine code with the ability to fill in a preset menu format. The menu formats can call other menus or do specific functions designated by the person setting up the system. The programmer can set up to 5 parameters to be answered by the user. SUPERVYZ then inserts these parameters into a command line that has been pre-programmed into the menu module using a format similar to the CP/M-provided SUBMIT.COM. (See Fig. 1)

This command line is much more flexible than that provided by SUBMIT since not only can you insert the parameters into the CP/M command line but you can run your transient program from within the SUPERVYZ command line. An example of a command line:



```
A:FORMAT   BD{Y||}|A:SYSGEN{A|B||}|PIP B:=A:PIP.COM|   @  SYSDISK
```

This line in a menu option by the programmer will:

1) Load the format program to format the disc in the B drive double density (if your format program takes the command with this syntax)
2) Answer YES when the format program asks for verification.
3) Do the formatting
4) Load the system transfer program
5) Answer 'A' as the source
6) Answer 'B' as the destination
7) Do the transfer
8) Load PIP
9) Copy PIP.COM from A: to B:
10) Run a .SUB file called SYSDISK to copy a list of files to the new disc.

Epic Software's Steve Fisher, who wrote SUPERVYZ, has provided an overlay to the CP/M Console Command Processor (CCP) which he calls SUPERCCP. To load SUPERVYZ you type SUPERCCP, which in turn loads SUPERVYZ and the first menu. As an alternative, when installing SUPERVYZ you can enable the autoboot function. With autoboot enabled, the menu comes up when the system is turned on. Epic has also provided an excellent series of menus to get you started.

In order to perform some of the functions from within the SUPERVYZ control line, some addition features were needed within the CCP.

;	comment -displayed on screen but no action taken. CLR clears the screen
GET	loads a specified program into memory
GO	runs the loaded program
LOG	provides a ↑C type reset of all discs
UNYZ	disconnects SUPERCCP and SUPERVYZ
WAIT	requests and waits for a key-press
XOFF	deactivates ↑S and ↑Q
XON	reactivates ↑S and ↑Q.

The additional intrinsic commands allow the following series of operations, as described in the manual, to back up a data disc that had been used on the B drive:

```
GET PIP             (puts PIP into memory but does not
                    run it)
                    (displays instruction message)
;Place Source in A:, Blank in B: then press <RETURN>
WAIT                (waits for the carriage return)
LOG B:              (makes drive B: R/W)
GO B: = A:*.*[VO]   (does pip of all files)
;Replace program disc in the A: drive (another comment)
WAIT                (
LOG A:              (makes drive A: R/W)
```

This series of commands will not fit on the command line of the menu item so it should be placed in a submit file, with a name of your choice, to be called from the SUPERVYZ menu.

The on-board help is something that caused me a great deal of concern in the first version (since updated twice) of SUPERVYZ I used. It was written in PILOT, a computer assisted instruction (CAI) language. When a '?' was typed at menu selection time, the screen cleared, the PILOT interpreter was loaded and the help file for the application was displayed. The first time I ran it, I thought my computer had hung up and I ran for the reset button. The next time, I waited longer and finally after 23 line feeds, I had my help. This defect has been corrected in the latest version that, incidentally, also runs under some versions of MP/M (check before ordering). In this latest version, not only does the help facility run much faster but the entire package has been remodeled to run faster.

To create help files for his client, the programmer must write the help file with his own editor following the rules of the HELP language supplied by Epic. I was not given a copy of the new manual so I cannot comment on any instructions for setting up the help files that may exist within the manual. The instructions given on the disc, however, within the supplied help programs, are very complete. The language permits the programmer to provide for a help file that can be called at any stage in SUPERVYZ's operation, whether it is by the end-user or the programmer setting up the new menus or associated help files.

Creating menus takes planning. There are only 10 options plus the help call that are permitted on each menu but each menu can call other sub-menus so there is really no limit. An '0' response on a menu will recall the previous calling menu. During menu creation, not only can a command line be constructed but you can designate up to 8 files that are required on the disc, and where they must be, in order to perform the operation. If any of them are not available to the system at run time, an error message will be displayed.

I must say that Steve Fisher is constantly revising SUPERVYZ and the latest version 1.35 that I was sent for checking is far superior to the original package.

****

STOK PILOT is a programming language. It is a superset of PILOT and will run all programs written in PILOT. Commands have been added to allow checking of files, executing of other transient programs, loading and calling of machine language subroutines, a case switch, direct port I/O and many more. As an interface to CP/M it is totally flexible as long as parameters to the called program can be passed thru the command line but this is a limitation that, at present, must be realized. It is for this reason that one cannot do a SYSGEN, for example, without being aware of the location of the source and destination of the system being transferred. I mention this, in particular, since this article deals with interfacing the novice end-user with CP/M. I do not expect this person to know such things as the answer to the question:

```
SOURCE ON ?
DESTINATION ON ?
```

With STOK PILOT, I wrote a file transfer program that requests separate input of the name and type of the source file, name and type of the destination file, as well as the source and destination disc drives. I must offer a disclaimer here. It is not totally debugged but can be used as a template for the type of thing that can be done with STOK PILOT. Asterisks were allowed and error checking performed at each stage of operation. In addition, at ANY point, a '?' displayed a help file explaining the PIP operation. Admittedly, to the experienced user of CP/M, an effort far surpassing its result. But to the confused novice trying to get a system up for the first time, it removes some of the confusion of house-keeping. (See Fig. 2)

STOK PILOT's list of commands are impressive when compared to the subset used in CAI:

A:	Accept answer (like INPUT in basic)
ANE:	Accept answer no echo: - do not allow display of input
C:	Compute - addition and subtraction only - allows assignment of values to numeric variables
CALL:	Call assembler program at location 100
HCASE(X):	Jump to labeled routine depending on value of X
CH:	Chain another PILOT program
CLRS:	Clear screen
COMPILE:	Compiles the ASCII program for fast loading and running
CUR:	Full cursor positioning control
DEF:	Assign string variables
DI:	Disable the escape key
DRIVE:	Select drive
EI:	Enable the escape key
E:	End of a subroutine
END:	Go back to CP/M
ESC:	Defines the action to be taken upon using escape key
EXEC:	Execute CP/M commands
EXIST:	Check disc for existence of specific files
HOLD:	With this one, <RETURN> continues and 'R' jumps to specifed label
INMAX:	Defines the maximum length of string input allowed
J:	Jump to a label
LF:	Line feed function such as 'LF: 10' displays 10 line feeds
LOAD:	Load a machine language subroutine with filetype 'LOD' into location 100H.
M:	Match - compare input from Accept to these values

(continued on next page)

MC:	Same but include commas
OUT:	Output to I/O port
PR:	Print on list device
R:	Remark
SAVE:	Save the input text variable in memory
T:	Display text on console
TNR:	Display text with no return (useful for input)
U:	Use subroutine called by label
USER:	Switch to designated user
WAIT:	Same as ACCEPT but is timed by preset interval

A very strong feature of the language is that all of the commands presented above may be modified by a conditional statement. The condition can be an expression, the value of a variable or the yes or no result of a previous match. For example, if a previous question had to be answered with yes or no, the use subroutine command:

```
U:  *routine1
```

can be modified to

```
UY:  *routine1 (use routine1 if the answer was yes)
UN:  *routine2 (use routine2 if the answer was no)
```

This can provide the ability to output to the printer or console by the following:

(the line numbers are for reference – they do not belong in the program)

```
1-      *START
2-      CLRS:
3-      LF: 10
4-      TNR: Do you want this output to the printer?
5-      INMAX: 1
6-      A:
7-      M: Y , N
8-      JN:*START
9-      M: Y
10-     PRY:THIS IS AN EXAMPLE OF PRINTING
        WHERE YOU WANT IT TO GO
11-     TN:THIS IS AN EXAMPLE OF DISPLAY TO
        THE TERMINAL
```

Line 1:	Label for later reference
Line 2:	Clear the screen
Line 3:	Output 10 line feeds
Line 4:	Display question with no carriage return at end
Line 5:	Allow only 1 character reply
Line 6:	Accept answer
Line 7:	Allow a match of only a Y or N
Line 8:	If there is no match (no Y or N), jump back to label
Line 9:	Try to match with Y
Line 10:	If there was a match with Y, go to list device for print
Line 11:	Only if there was no match, display on console - leaving off the N in line 11 would have caused a console display under either condition.

In addition, several special characters are allowed in entry. The { and } are used in T: lines to turn on and off reverse or modified video depending on your terminal; and the caret (<) allows output of any special control characters you might need to send either to your printer or console.

STOK PILOT may be invoked in three modes: TEST, RUN or COMPILE. In test mode, the ASCII file is examined and any errors in syntax are reported. The program is not run. In compile mode, the compiler reads the line in the ASCII program that starts with the command COMPILE: and compiles the program as file type COM. If the ASCII program includes a command to load a machine language LOD type subroutine file, this file will be included in the compiled program. STOK PILOT reserves a 2K block at 100H for these machine language routines.

****

CP+

At first run, I was very disappointed with CP+. However, this initial disappointment was replaced in a short time with the realization that I was observing a disaster. It is at times like these that I would wish every seller and every user of an S-100 computer were a reader of Lifelines. We need impartial evaluation of software to help us select good from bad, useful from worthless and even good from better. I would be reluctant to part with $50 much less the $150 required. It comes with a very nice looking manual but doesn't promise to do very much. What it does promise to do, it does badly. The various interactions with CP/M are locked into a program written in the C language and not modifiable in any way by either the end-user or the dealer supplying him.

Only the barest CP/M facilities are utilized by CP+: PIP, STAT, and the intrinsics DIR, REN,ERA. To run anything else requires finding the menu item for entering a CP/M command and knowing what arguments to be passed and whether the program will accept arguments. One of the problems of a reviewer is looking at a program from the point of view of the person buying the software. It would not be fair to judge CP/M simplifying software from the point of view of someone thoroughly familiar with the operation of CP/M. We must "stupidify" ourselves and pretend we only know how to turn on the machine and type the name of a program. Fortunately, (very fortunately), I do not "stupidify" completely and I was able to realize that things were not happening as they should.

Let's stop and look at it from the end-user's point of view. If he buys his computer from any source that limits the assistance given to him (for example, mail order), the authors of CP+ claim that it can be up and running with NO programming necessary.

Upon opening the manual the buyer is in for a surprise. The pages are neatly divided with tabbed dividers for ease in locating the sections. But here's the surprise: Section I is called "Meet CP+". It consists of only 1/2 page. O.K! On to Section II called "Using This Guide". Again, only 1/2 page. It is only when we arrive at Section III, "Operating Concepts & Glossary" that I was able to find a use for the tabbed dividers. This section has 4 pages. If there is no need for a divider, why put on such a show over nothing? Could this be a portent of things to come ?

The installation procedure is simple and the manual written in a tutorial fashion with demonstration files on the disc. One can type CP+ and from then on is told one can run all of one's programs from there, doing whatever copying, etc. from within the CP+ environment. I found myself unable to do that. As I stated in the beginning of this article, the PIP problem and the making of files Read Only or Read/Write when desired is overwhelming to the beginner at the first try.

There was some thinking going on in the creation of CP+ and the author found some interesting problems that needed solving. For example:

1) allowing a description of each file in the directory

2) performing multiple renames with wild card extents

3) setting up a queue for copying, printing and erasing

Some of these problems were solved but most of them were not solved in my copy of CP+. Will they be in the future ?

The tutorial is directed at simplicity. For example, from the glossary:

The Cursor - The cursor is the moving marker that helps tell the user where the next action is to take place on the screen. It is usually a ■ or a ____, and may be blinking on or off.

The main menu, in addition to a help call, review commands, change user areas and quit to CP/M contains three sub-menu calls. You can select and run a program, get the print command menu or a file command menu.

All of the menus are the same format:

(SEE FIG. 3)

For some reason, the drawing of the lines around the menu has gone askew. All line are printed, and then the vertical lines are erased. This a minor bug in a program severely infested but is an indication of carelessness with something that could have been easily remedied. An examination of the Select and Run Menu yields a surprise right away. In the area designated for the File catalog (notice I said catalog not directory), is a wide column for File Description. A facility within CP+ allows you to create a catalog file on the disc. This facility reads the directory and invites you to type in a description of the directory entry. When you want to run a program, you can determine which program you want to run by its description, not by a file name limited to 8 characters and 3 in the file type. How often I have looked at filename such as "DU7GL.COM" and wondered what in blazes that was supposed to mean.

The (S)elect option allows you to page thru the catalog looking for the program to run. The (E)nter option permits entering the program to be run by its catalog number. When I tried doing this with PIP, I copied a short file from one disc to the other and waited for the return to the CP+ menu. I was disappointed to see the CP/M "A>" and had to re-enter CP+. That happened every time I tried to run a transient program from a menu or when I used the "Enter a CP/M command" mode from another menu. An unexplained error message "Updating the Print Queue" was displayed just before exiting to CP/M.

One of the unfulfilled promises is that of being able to mass rename files using wildcard attributes. By this means, one should be able to rename DEM1.TXT, DEM2.TXT, DEM3.TXT and DEM4.TXT by entering to the rename function DEM?.TXT as the old filename and DEMO?.TXT as the new. This is supposed to yield the four files DEMO1.TXT, DEMO2.TXT, DEMO3.TXT and DEMO4.TXT. Sorry, but it just doesn't work. The first file was renamed to DEMO and I had the option of deleting each already-existing file as CP+ tried to rename every other file DEMO. I would have ended up with one file called DEMO. That could have been a disaster had I not realized the problem myself and stopped the operation.

I tried the "'Copy System Files" menu. The option is copying the CP/M system files or CP+ system files. The first requires that you have SYSGEN on the disc. Sysgen is run, you must answer its prompts, it does its thing and back to CP/M......Reload CP+ again!!!! Copying the CP+ files returned to CP+ after copying and verifying each file copied. It used its own copy facility rather than PIP and was much slower.

One of the teasing prospects offered to the CP+ buyer is the hope of setting up a print queue or list of files to be printed and having it done, batch mode, while the operator is off having lunch. I tried it. I printed the first file, and the machine appeared to get trapped in a loop of formfeeds that did not end until I turned off the printer.

There are several parameters that I have used to judge each of these three packages, SUPERVYZ, STOK PILOT and CP+. The scale goes from 0 to 7.

P = supplied by programmer

	SUPERVYZ	STOK PILOT	CP+
Flexibility	5	7	1
End User Usefulness	7	7	2
Programming Level Needed	5	6	0
Fulfills Promise	7	7	2
Expandability	7	7	0
Speed	4	6	5
Manual Clarity	6	7	6
On Board Help (for programmer)	6	0	2
On Board Help (for end user)	P	P	2
Price	$95	$130	$150
Operating System:			
CP/M 1.4	no	yes	yes
CP/M 2.x	yes	yes	yes
MP/M 2.x	some	no	no

With this in mind, I would recommend SUPERVYZ as the most useful of the three. One surprise is that it is also the least expensive package of the three. Another benefit I discovered during the review process was that Steve Fisher is evidently a software hacker himself and unable to leave a good thing alone. There are more modifications and improvements in the works now and should be available by the time you read this. The programmer using STOK PILOT could provide most of the same facilities and his own unique format but at a greater effort on his part. The effort in programming the copy program illustrated is considerably more than that using SUPERVYZ's predesigned menus but the gain could be confidentiality of his source when providing compiled programs.

(continued on next page)

Available from:

Supervyz:	Epic Software Corporation 7542 Trade Street San Diego, CA 92121
Stok Pilot:	Stok Software Inc. 17 W. 17th Street New York, NY 10011
CP+:	Taurus Software Corporation 870 Market Street San Francisco, CA 94102

****

Figure 1

```
                 Epic Computer Corporation's SUPERVYZ Menu Definitions
To move within a field, use Right/Left Arrows or Control-D/Control-S.
To move between fields, use Up/Down Arrows or Control-E/Control-X.
To erase a character, use Delete; to erase a field, use Control-C/Control-U.
Pressing the Escape key completes entry of the selected Function or Menu.


Item (0-11) [6 ]   Function Title [System's Editor                       ]  Clear? [Y]
Command Line [A:ED $3:$1.$2{-A|23T}|ERA $3:$1.$$$                                  ]

      Optional Parameter Prompt          Optional Parameter Default       Length
 1 [What is the File Name?            ] [                              ]   [8 ]
 2 [What is the File Type (if any)?   ] [                              ]   [3 ]
 3 [ On which Drive?                  ] [A                             ]   [1 ]
 4 [                                  ] [                              ]   [  ]
 5 [                                  ] ]                              ]   [  ]

   Drive User  Filename     Ext    Require  Hide Drive User  Filename     Ext    Require  Hide
 6  [A] [0 ] [ED        ] [COM]     [Y]     [Y]|  [?] [? ] [*         ] [*    ] [     ] [ ]
 7  [ ] [  ] [          ] [   ]     [ ]     [ ]|  [ ] [  ] [          ] [     ] [     ] [ ]
 8  [ ] [  ] [          ] [   ]     [ ]     [ ]|  [ ] [  ] [          ] [     ] [     ] [ ]
 9  [ ] [  ] [          ] [   ]     [ ]     [ ]|  [ ] [  ] [          ] [     ] [     ] [ ]
```

Figure 2

```
R:      PCOPY.PIL
R:    this is a multiple copy program that calls PIP
ESC: *BYE
COMPILE: PCOPY
CLRS:
*START
R:      define the variables as blanks for entry into the
R:      first screen
DEF: $TODR
DEF: $FRDR
DEF: $TOFL
DEF: $FRFL
DEF: $TOTP
DEF: $FRTP
C:N=1
*NUSTART
R:      Get the entry form
U:CLRMSG
U:ENTRY
*FDR
CUR:22,10
INMAX: 1
A: $FRDR
U:CLRMSG
M: ,,
UY:ERRMSG
M: Z ,
ENDY:
JY:*FDR
M: , ? ,
UY:*HELP
JY:*FDR
*TDR
CUR:62,10
INMAX: 1
A:$TODR
M: ,,
UY:ERRMSG
JY:*TDR
M: , ? ,
UY:*HELP
JY:*TDR
R: ***************************************************
*FFL
CUR: 22,14
INMAX: 8
A: $FRFL
M: *,?
R:       establish a value for switch if asterisk or ?
R:       entered in file name
CY:N=0
U:*CLRMSG
M: ,,
UY:*ERRMSG
JY:*FFL
M: , ? ,
UY:*HELP
JY:*FFL
*FTP
CUR: 22,15
INMAX: 3
A: $FRTP
M: *,?
R:       establish a value for switch if asterisk or ?
R:       entered in file name
CY:N=0
M: ,,
UY:*ERRMSG
JY:*FTP
M: , ? ,
UY:*HELP
JY:*FTP
T:

R: ***************************************************
*TFL
CUR: 62,14
INMAX: 8
```

```
A:$TOFL
U:*CLRMSG
M: ,,
UY:*NULLTO
JY:*START
M: , ? ,
UY:*HELP
JY:*TFL

*TTP
CUR: 62,15
INMAX: 3
A:$TOTP
U:*CLRMSG
M: ,,
TY:
UY:*NULLTO
JY:*START
M: , ? ,
UY:*HELP
JY:*TTP

R:      this exec for when have to & from files entered
U:*OK
M: Y,OK
JN:*NUSTART
EXIST(N): $FRDR:$FRFL.$FRTP
JN:*ERR2
JN:*NUSTART
EXECY: PIP $TODR:$TOFL.$TOTP=
    $FRDR:$FRFL.$FRTP;PCOPY
END:

R:      this exec for when NULL entered as destination
R:      name
*NULLTO
U:*OK
M: Y,OK
JN:*NUSTART
EXIST(N): $FRDR:$FRFL.$FRTP
UN:*ERR2
JN:*NUSTART
EXECY: PIP $TODR:=$FRDR:$FRFL.$FRTP;PCOPY
END:

*ENTRY
CUR: 20,3
T:{ SINGLE/MULTIPLE FILE COPYING }
CUR: 16,6
T:ENTER {? } AT ANY ENTRY FOR HELP
CUR: 0,7
T: +++++++++++++++++++++++++++++
CUR: 0,9
T:{ enter z here to leave copy program }

CUR: 0,10
T:{ FROM } DISC DRIVE
CUR: 21,10
T: $FRDR
CUR: 0,14
T:{ FROM } FILE NAME
CUR: 21,14
T: $FRFL
CUR:0,15
T:{ FROM } FILE TYPE
CUR: 21,15
T: $FRTP

CUR: 40,12
T:{ (may be CR only for SAME Name & Type) }
CUR: 39,10
T: { TO } DISC DRIVE
CUR: 61,10
T: $TODR
CUR: 39,14
T: { TO } FILE NAME
CUR: 61,14
T: $TOFL
CUR: 39,15
T: { TO } FILE TYPE $TOTP
CUR: 61,15
T: $TOTP
CUR: 0,17
T: +++++++++++++++++++++++++++++
E:

*HELP
CLRS:
LF: 4
T:You must enter a file drive for both the {from} drive
T:and the {to} drive.
T:
T:You must enter a file name and file type for the {to}
T:filename and type.
T:
T:However, the {from} filename and type may be
T:entered with a {CR} alone,
T:if you want to keep the same file name.
T:
T:To copy all files with the same filename (the part to the
T:left of the dot),
T:enter an asterisk { * } for the filename.
T:To copy all files with the same file type (the part to the
T:right of the dot),
T:enter an asterisk { * } for the file type.
T:
T:THEREFORE, to copy all files from the disk, enter { * }
T:for both the file-
T:name and file type.
T:
T:      Now press {RETURN> } to continue.
HOLD:
CLRS:
R:      re-enter the entry form and return to same point
U:*ENTRY
E:

*ERRMSG
CUR: 20,20
T: { YOU MUST HAVE AN ENTRY }
E:

*ERR2
CUR: 20,20
T: { $FRFL.$FRTP NOT FOUND ON DRIVE $FRDR }
```

(continued on next page)

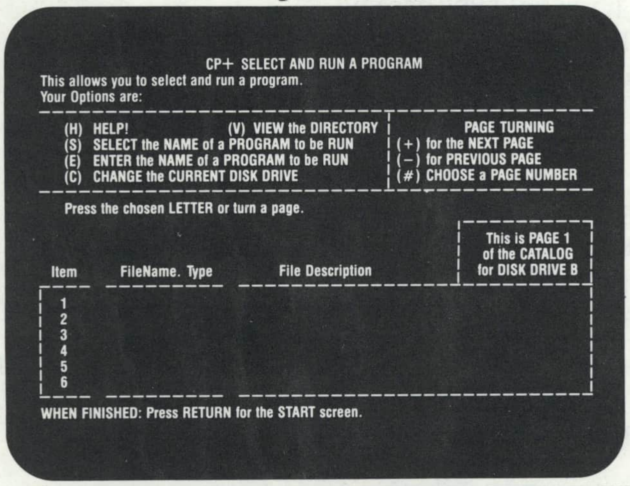
```
E:

*CLRMSG
CUR: 20,20
T: E:

*OK
CUR: 27,20
TNR:{ O.K. TO COPY ? }
INMAX: 1
A:
E:

*BYE
CUR: 27,20
T:{COPY ABORTED }
END:
```

Figure 3



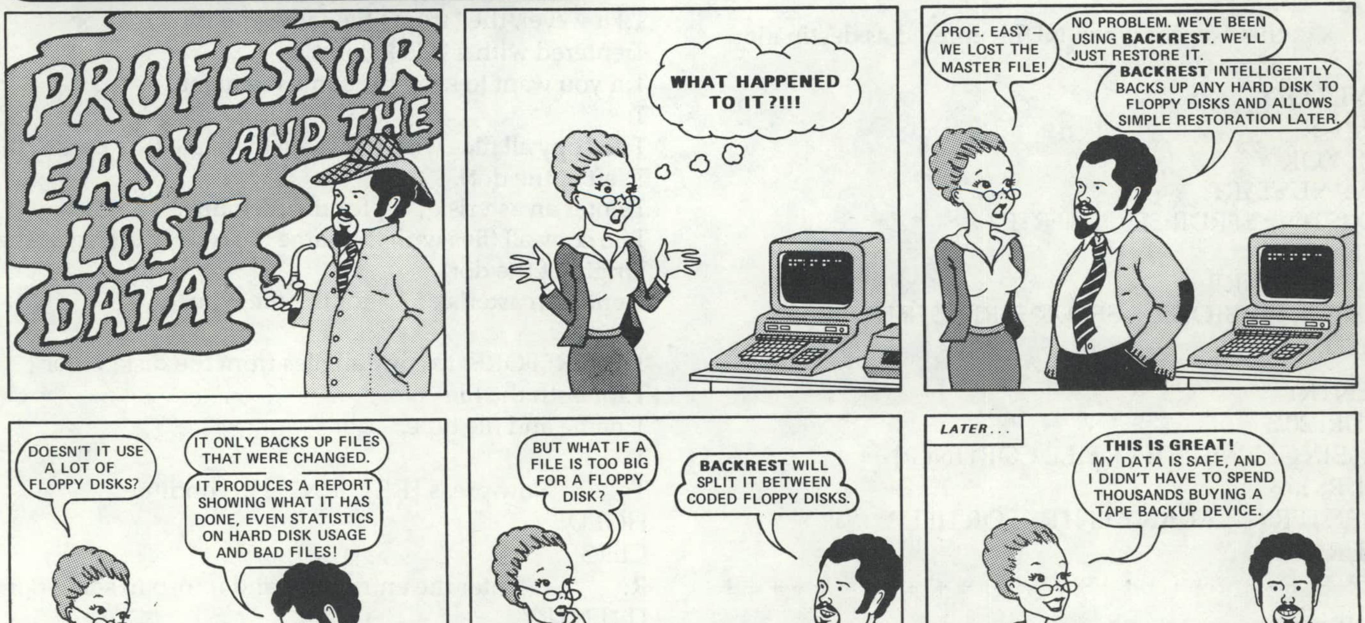
```
                         CP+ SELECT AND RUN A PROGRAM
This allows you to select and run a program.
Your Options are:
-----------------------------------------------------------------------
   (H)  HELP!                  (V) VIEW the DIRECTORY |      PAGE TURNING
   (S)  SELECT the NAME of a PROGRAM to be RUN        | (+) for the NEXT PAGE
   (E)  ENTER the NAME of a PROGRAM to be RUN         | (-) for PREVIOUS PAGE
   (C)  CHANGE the CURRENT DISK DRIVE                 | (#) CHOOSE a PAGE NUMBER
-----------------------------------------------------------------------
   Press the chosen LETTER or turn a page.
                                                       This is PAGE 1
                                                       of the CATALOG
 Item     FileName. Type          File Description    for DISK DRIVE B
   1
   2
   3
   4
   5
   6
WHEN FINISHED: Press RETURN for the START screen.
```

A PROFESSIONAL SYSTEM
AT A P.C. PRICE
$2995
TURN-KEY S-100 SYSTEM
FEATURING:

Integrand 10 slot enclosure	Teletek Systemaster SBC
2 8″ D.D., D.S. Drives	2 Parallel & Serial Ports
ADDS 3A Viewpoint Terminal	CPM™ 2.2 Installed

Full Teletek line available. Multi-user & Turbodos™ options can be added. Other S-100 products, printers, peripherals, personal computers, and CPM™ software products available at 15-20% above wholesale cost. Full service and repair. Workshops and classes held regularly.

TOTAL ACCESS
SUITE 202, 2054 University Ave.
Berkeley, California 94704
415-540-8066


PROFESSOR EASY AND THE LOST DATA
WHAT HAPPENED TO IT ?!!!
PROF. EASY – WE LOST THE MASTER FILE!
NO PROBLEM. WE'VE BEEN USING BACKREST. WE'LL JUST RESTORE IT. BACKREST INTELLIGENTLY BACKS UP ANY HARD DISK TO FLOPPY DISKS AND ALLOWS SIMPLE RESTORATION LATER.
DOESN'T IT USE A LOT OF FLOPPY DISKS?
IT ONLY BACKS UP FILES THAT WERE CHANGED.
IT PRODUCES A REPORT SHOWING WHAT IT HAS DONE, EVEN STATISTICS ON HARD DISK USAGE AND BAD FILES!
BUT WHAT IF A FILE IS TOO BIG FOR A FLOPPY DISK?
BACKREST WILL SPLIT IT BETWEEN CODED FLOPPY DISKS.
LATER...
THIS IS GREAT! MY DATA IS SAFE, AND I DIDN'T HAVE TO SPEND THOUSANDS BUYING A TAPE BACKUP DEVICE.

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

Complete 8 inch CP/M format disk and manual retails for $99.95. N.Y. residents please add sales tax.

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

Dealer inquiries invited.
CP/M is TM of Digital Research

Feature

Review of Micro Resources Washington Version 3.2

Charles H. Strom

There are several utility programs that Digital Research has included on the CP/M distribution disk that are designed to do directory and file manipulation. These programs are not the most user friendly or convenient programs, and we have seen many alternatives developed over the past several years both in the public domain and proprietary arenas. One of these, WASH, is a program that is distributed by Micro Resources, 2468 Hansen Court, Simi Valley, Ca. 93065. At the the time of writing, the price for WASH was $49.95. I obtained the program on 8" single density IBM standard format diskette, and I assume the various popular 5.25" formats are available as well.

Two files are supplied on the distribution diskette - WASH.COM, the utility itself, and WASHINST.COM, an installation program. The manual accompanying the program is photo-offset and 22 pages in length. The manual covers installation using WASHINST in the case where the target terminal is listed in the menu (whereby the installation is trivial) as well as for any terminal not covered in the menu. My Zenith Z-19 was included on the menu, but it is a relatively simple procedure to customize WASH for an unsupported terminal. All that is required is that cursor addressing and clear screen functions are available on the CRT. The author even had the foresight to allow padding by sending a user-specified number of nulls (00 hex characters) after a cursor control sequence so as to allow use on terminals requiring a pause at very high baud rates. I am presently using WASH at 19.2 Kbaud with no display problems whatsoever. Another often-overlooked capability included is to use ANSI-type cursor addressing as an alternative to ASCII addressing.

WASH is capable of running in non-video mode for use on a hardcopy terminal as an alternative installation-time option, but the program is clearly designed as a screen oriented utility, and that is where it really shines. See figure 1 for a typical display of the file directory while running WASH. The header displays the current drive and USER number (the latter may be disabled at installation time if running CP/M version 1.4 or if USER capability is not desired). The remaining free space on the disk is also displayed; the program uses CP/M's disk parameter block (DPB) to automatically adjust for allocation group size, so this is automatically calculated regardless of disk format. The U command can interrogate any drive for remaining space as well.

The file listing consists of four columns of files on a 24 by 80 terminal and shows both visible and system files ($DIR or $SYS attributes as set by STAT.) There is a file pointer (two arrow heads) that can be advanced forward by the space or carriage return keys, backward with the B key (case does not matter) or can "zip" ahead 10 files with Z. It is thus very easy to immediately get to a specific file on the disk.

Two of the available commands operate either on one file or on a group of files that have been "tagged" by the operator. When tagged, there is a visual cue of a "#" symbol next to the file. Tagged operations include deletion (the operator is queried, if he so requests, on a file-by-file basis to prevent catastrophes) and mass copy to a specified disk/user combination. There is no re-tag function, which might be useful if a user wanted to copy the same set of files to more than one destination disk/user. Other single file commands allow for deletion (with query), copy to a destination disk/user, file view on the console device which can be frozen with ↑S á la CP/M itself or aborted with any other keypress, and output of a file to either the LST: or PUN: logical device. The view function (as do both punch and list) conveniently strips the high order bit from each character before display so WordStar files, for example, may be clearly seen.

Additional commands include report of the size of the currently pointed-to file, display of version number/author, exit to the CCP via warm boot, and login of a new drive/user combination. This latter very powerful command enables the user to call the program and then do any necessary manipulations on any other disks without the necessity for WASH to be resident on disk. The program is more or less foolproof in that there are complete, easy to understand error messages. One kink that annoyed me is that if I tried to delete a read-only file, I got the all too familiar "BDOS" error and was dumped unceremoniously back to the operating system. The author tells me that indeed there is no provision for deleting or renaming R/O files because, in essence, he does not use this feature of CP/M and therefore never really considered this a necessary capability; it may be included in a future version. Another suggestion for future inclusion is that SYSTEM files (those not normally visible through the DIR command) be flagged as such. Under the current version, there is no such flagging and all files are displayed at all times. This is admittedly a very minor quibble. I would also like to see a command that will re-tag a group of files as previously mentioned; in this way I could follow a mass copy to an archive file with a mass deletion of the files from a working disk, for example.

The WASH manual is supplied as a twenty-two page photo offset set of sheets stapled together. Not very fancy, but it is indeed the best example of a manual I have yet run across in CP/M software. There is an introduction, an installation section, complete descriptions of all commands, and discussion of all possible error messages and what they mean. Granted it is easier to produce a grade A manual for a program of lim-

(continued on next page)

ited scope such as WASH than it is for a complicated word processor, for example, but after seeing so many examples of omission, misdirection and downright illiteracy in CP/M applications programming manuals (with incidentally, no relation to the price charged), it is gratifying to see that there are people in the microcomputer world who pride themselves on being able to communicate with the user.

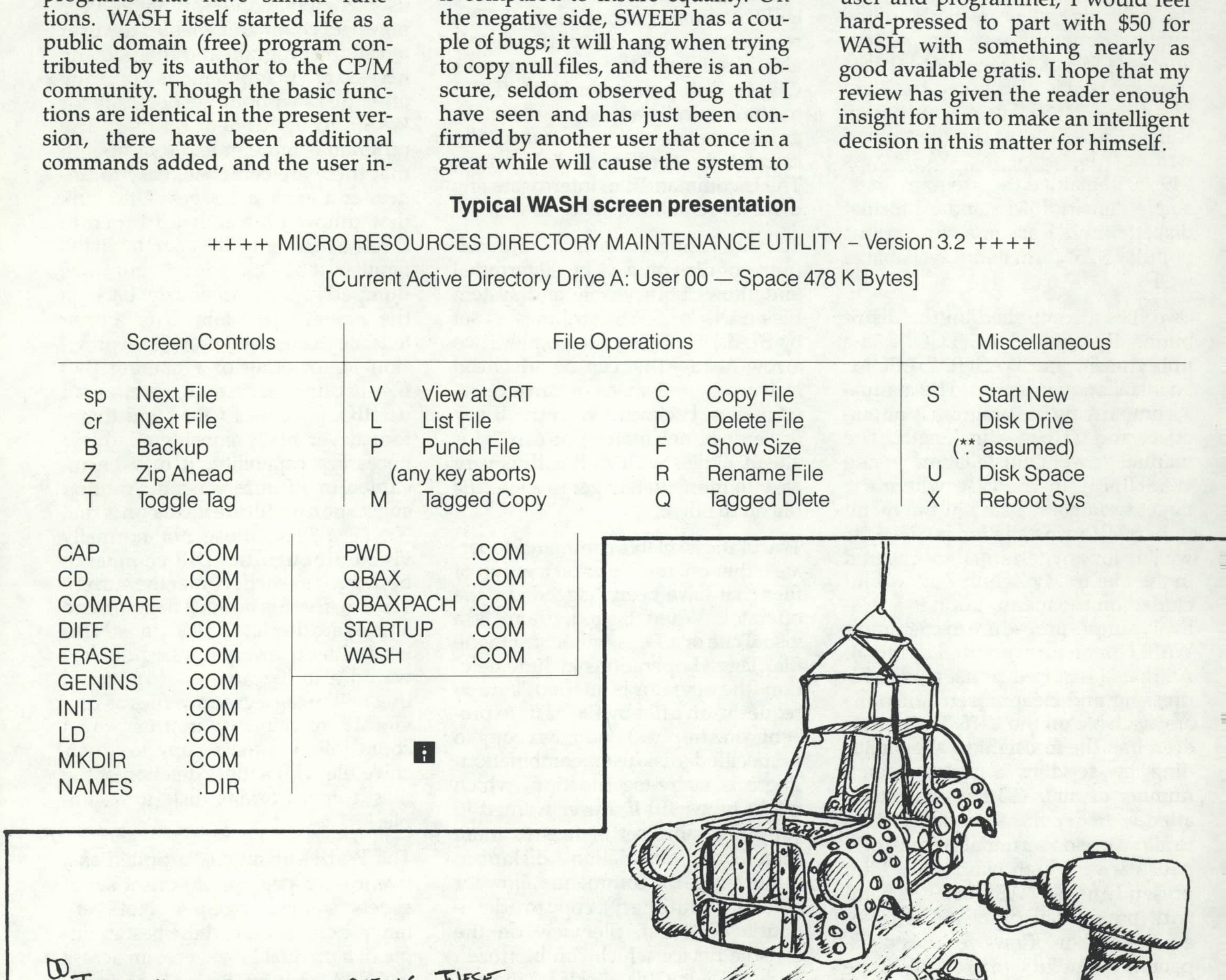
The final discussion appropriate for a review of a program such as WASH is a comparison with public domain programs that have similar functions. WASH itself started life as a public domain (free) program contributed by its author to the CP/M community. Though the basic functions are identical in the present version, there have been additional commands added, and the user interface is much friendlier. The other public domain program with similar capabilities is called SWEEP, written by Robert Fisher. SWEEP is written in PL/I, and is therefore several times larger than WASH. It takes up more space on a disk and takes longer to load, and does not have the screen oriented display of WASH. The functions are slightly more complete; there is support for read-only files (see my complaint above) as well as an optional copy with verification. In this case a CRC (Cyclic Redundancy Check) is calculated for the file both on source and destination disks and is compared to insure equality. On the negative side, SWEEP has a couple of bugs; it will hang when trying to copy null files, and there is an obscure, seldom observed bug that I have seen and has just been confirmed by another user that once in a great while will cause the system to hang in the midst of a multiple CRC file copy operation. The biggest advantage of SWEEP is that it is free! If I had my choice of either SWEEP or WASH for free or for that matter of purchasing each one for $50, I would choose WASH because of its display formatting and smaller size (and faster execution speed.) If I were a novice CP/M user, I would also lean towards WASH because of its easier to understand operation, the excellent manual, and the vendor support that is understandably absent in a public domain program. However, considering myself a seasoned CP/M user and programmer, I would feel hard-pressed to part with $50 for WASH with something nearly as good available gratis. I hope that my review has given the reader enough insight for him to make an intelligent decision in this matter for himself.

Typical WASH screen presentation

```
       ++++ MICRO RESOURCES DIRECTORY MAINTENANCE UTILITY – Version 3.2 ++++
            [Current Active Directory Drive A: User 00 — Space 478 K Bytes]

     Screen Controls   |            File Operations              |   Miscellaneous
 ----------------------+-----------------------------------------+------------------
  sp   Next File       |  V   View at CRT      C   Copy File     |  S   Start New
  cr   Next File       |  L   List File        D   Delete File   |      Disk Drive
  B    Backup          |  P   Punch File       F   Show Size     |      (** assumed)
  Z    Zip Ahead       |  (any key aborts)     R   Rename File   |  U   Disk Space
  T    Toggle Tag      |  M   Tagged Copy      Q   Tagged Dlete  |  X   Reboot Sys

 CAP       .COM   |  PWD       .COM   |
 CD        .COM   |  QBAX      .COM   |
 COMPARE   .COM   |  QBAXPACH  .COM   |
 DIFF      .COM   |  STARTUP   .COM   |
 ERASE     .COM   |  WASH      .COM   |
 GENINS    .COM   |-------------------
 INIT      .COM   |
 LD        .COM   |
 MKDIR     .COM   |
 NAMES     .DIR   |
```



Tips & Techniques/Unbuffering Your Input

Robert P. Van Natta

Most of us have come to appreciate keyboard buffering as a BIOS feature. Where present, the keyboard buffer permits you to type ahead of the computer. This means that if you have an input sequence that is interrupted by disk action, for example, you can input the entire sequence at once, and it will be read from the buffer as needed.

There are occasions, however, when it is convenient to blow the buffer away and provide a clean slate. This is often the case with respect to error processing. If you have an input sequence that permits the operator to get ahead of the machine and something runs amok, you will want your program to branch to an error routine which will ring the bell, flash the screen, or otherwise protest. Once the error is detected, oftentimes the remainder of the keyboard buffer will simply wind up being more errors, due to the unexpected branching caused by the first error. This may mean incessant bell ringing or screen flashing.

A more reasonable approach is to have a keyboard buffer discharge routine built into the error trap. If this is done, then you can leave the error trap with a clean slate and greatly reduce the likelihood of an immediate return.

In CB-80, there are two functions that are very useful in accomplishing this task. They are CONSTAT% and INKEY. The CONSTAT% function tests the console status to see if a character has been entered but not read. INKEY will read a character and will not echo it to the screen.

Thus you can blow out the input buffer with the routine set forth as follows:

```
    WHILE CONSTAT%
          dummy%=INKEY
          FOR delay%=1 TO 1000
          NEXT delay%
    WEND
```

This example, however, introduces several novel concepts which need further explanation. The first line is quite unusual in itself. By definition the CONSTAT% function returns a -1 if a key has been entered and not read and a 0 if the console is ready. The same program logic could be obtained by writing "WHILE CONSTAT%=-1" or "WHILE CONSTAT%<>0"; however, I deliberately chose the version set forth to illustrate what is referred to in the CB-80 manual as a 'logical expression.' If you are familiar with Pascal, you will probably be tempted to use the word 'boolean' to describe how this works, but for reasons not clear to me the CBASIC/CB-80 documentation conspicuously avoids the word.

The point is, regardless of what you call it, that whenever you are testing an expression to see if it is a −1, it is unnecessary to include the '=−1' portion of the equation in your source code. This is equally true of IF ..THEN statements. You may, for example, quite properly write, 'IF a% THEN STOP', which will do the same thing as the more familiar 'IF a%=−1 THEN STOP.'

Perhaps, I have digressed too much. Getting back to the original example, if a key-stroke is in the buffer, CONSTAT% will return a −1 and you will enter the loop. Next the INKEY function will read one character out of the buffer. It will not be printed on the screen, but its value will be assigned to dummy%. As its name implies, dummy% is a temporary variable with no particular purpose other than to provide a resting place for the value returned by the INKEY function.

The next line is, of course, a delay loop designed for the single purpose of wasting some time. The need for its presence, is, more than anything, the inspiration for this article. I first wrote this program segment without this delay loop, and then spent the next two days trying to figure out why it didn't work. The particular example was developed on a Radio Shack Model 16 using Lifeboat CP/M version 2.25d. I suspect, however, that the problem is not unique to this equipment, and that is why I write. The problem is that the CONSTAT% function returns a misleading response if it comes too soon after the INKEY function.

I would be going quite a way out on a limb if I were so brash as to claim to know why this is so. However, I am satisfied that it is not a bug in CB-80, as the code generated by CB-80 appears to be correct. I surmise, therefore, that it is a limitation in the implementation of the keyboard buffering scheme.

The problem, if it exists in your system can be demonstrated by the following CB-80 program:

```
print "hit several keys at once"
a%=inkey
for stall =1 to 400
next stall
while constat%
        a%=inkey
        print constat%
        for wait%=1 to 1000
        next wait%
        print constat%
wend
```

Ordinarily, you would expect this program to print a column of '−1's down the screen until the buffer was empty and then print two zero's. Pickles and Trout CP/M will do just that. However, Lifeboat CP/M for the Model II/16 will print an alternating column of 0's and −1's, thus demonstrating the unreliability of CONSTAT% which immediately follows an INKEY.

The problem identified here is equally applicable to the use of the CONCHAR% function. It may likewise be demonstrated in CBASIC V. 2.07. However, with CBASIC it is just barely demonstrable in the most Spartan loop, and to my observation sometimes works correctly and sometimes not.

(continued on next page)

How much delay is needed?

For those of you that have been with CB-80 for a while you will recall versions of it prior to version 1.3, in which the INKEY function failed to work at all. As I understand it, the cause of this early failure was a 'documented bug' within CP/M function 6. Version 1.3 of CB-80 solved this problem by bypassing function 6 and using direct console I/O through the BIOS.

Although I don't rule out the possibility of a connection between the erratic behavior of the CONSTAT% function and the INKEY function, the connection is not obvious (at least to me).

The amount of absolute time required for CONSTAT% to get its act together is very minor. In the example, I show a CB-80 integer loop with a size of 1000. I selected this size on the basis that:

1) a loop size of 300 always produced the wrong response.

2) a loop size of 400 produced erratic results.

3) a loop size of 500 appears to work reliably.

A word of caution is appropriate here, however. If you are inclined to attempt to duplicate my results, understand that the print statements inside the WHILE loop require considerable time themselves, and must be removed for accurate testing. The testing problem is therefore somewhat akin to checking a refrigerator door light to see if it really shuts off when the door is closed.

As you know from an earlier article of mine about CB-80 in *Lifelines/The Software Magazine*, CB-80 only executes six instructions to complete an iteration of an integer FOR loop. If you would care to multiply 6000 times the number of instructions per second that the Z80 processor will handle, it should be obvious that the time delay is not going to ruin the performance of your program.

Conclusions

The example that I used to begin this article is almost a textbook example of the correct way to implement a WHILE loop. Likewise, the uses of CONSTAT% and INKEY are very appropriate. It is unfortunate that a delay must be inserted to make it all work correctly. I believe, however, that the overall concept of using this routine to blow down the input buffer is a good one as there are often critical places in a program (i.e. points of no return) where it would be best if the operator could not anticipate the future.

From my vantage point on the flanks of Mt. St.Helens, I have no idea how pervasive the problem described here is among CP/M implementations, but it is my notion that if it appears in Lifeboat CP/M for the TRS80 MOD II/16, it probably appears elsewhere as well, and should be a consideration in any applications program writing. ■

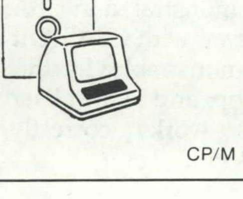
Make life easier with...

WASH™

the latest in an easy to use CP/M directory maintenance utility that replaces a dozen older programs with its menu driven file handling capabilities like:

- DIRECTORY DISPLAY
- FILE VIEW AND PRINT
- FILE RENAME
- FILE DELETE
- FILE COPY
- FILE AND DISK STATISTICS
- TAGGED GROUP COPY AND DELETE

WASH is fully compatible with CP/M user area directories and is delivered with a user friendly installation program to adapt WASH to your 24 x 80 console. WASH is available on 8" single density diskette for $49.95 plus 6% sales tax.



Contact:
MICRO RESOURCES
2468 Hansen Ct.
Simi Valley, CA 93065
(805) 527-7922

CP/M is a trademark of Digital Research

Clarification

Charles Klug, Sales Support Specialist of Televideo Systems, Inc. has informed Lifelines that the following information given in Charles Sherman's Tips & Techniques column on Televideo 950 (Jan. 1983, Vol. III, No. 8, p. 34) is inaccurate. "If it doesn't have revision level 2.0 or higher, write or call Televideo and they will send you free replacements."

Below is an excerpt from a letter from Mr. Klug stating their policy concerning replacement of firmware.

The article... indicated all users could call TeleVideo and receive the latest revision firmware for their terminal.

That is true, if, there is a problem validated by TeleVideo in using our terminals and it relates to a firmware problem. However, we do not otherwise replace firmware.

Should the user care to obtain the latest firmware it can be ordered through their dealer.

Users of TeleVideo terminals are welcome to call our Product Support Group if they are experiencing problems in making our product operate or require more information on our terminal products.

Feature

An Introduction To 8086 Programming

John Blanton

Overview

With the advent of the new generation of 16-bit home computers, typified by the IBM PC, there is a growing interest in the sixteen-bit processors on which the new systems are based. Those who have been with the home computer movement since its advent seven years ago are thoroughly acquainted with such eight-bit devices as the 6502, the 6800 and the 8080/Z80, and much home computer conventional wisdom is based on these simple, eight-bit microcomputers. The new sixteen-bit processors, however, differ from their predecessors in more than just the number of bits in the accumulator. Often more powerful instruction sets are available, there are additional registers and addressing modes and even the system architectures are bolder and more ambitious. Understanding and programming the 16-bit microprocessors will require mastering these new architectural features.

The 8088, on which the IBM PC is based, is functionally identical to the 8086, the main difference being that the 8088 has an eight-bit external data bus while the 8086 has a sixteen-bit external bus. The 8086, developed by Intel, is an expansion of the 8080 architecture, also developed by Intel. While the Zilog Z80 processor is also an offshoot of the 8080, it is essentially an 8080 with additional architectural features and an expanded instruction set. The 8086 does not incorporate the unique Z80 features, such as the alternate register set and the IX/IY registers, instead it is a unique expansion of the 8080, both in data width and in architecture. Much of the new architecture incorporates features previously available only with mini-computers, and it is primarily those novel aspects of the 8086 that will be discussed in this article.

Programmers already familiar with the 8080 or the Z80 should find this article useful in getting started in 8086/8088 programming, however it is not expected that one can get a thorough education in 8086 programming just by reading this. Further study should be pursued by acquiring and mastering one or more of the books devoted entirely to this subject. There are several to choose from, and four are cited here: *The 8086 Family User's Manual* by Intel gives a complete, technical description of the 8086 (and associated devices) plus full details of the instruction set and descriptions of Intel's program development tools (such as the ASM-86 assembler and the LINK-86 linking loader). The *iAPX 86,88 User's Manual*, also by Intel seems to be a more recent version of the previous book, and either book should provide very comprehensive coverage for the serious user. Written at a lower level but considerably more readable than the Intel manuals is *The 8086 Primer* by Stephen P. Morse from Hayden Book Company. *The 8086 Book* by Russell Rector and George Alexy from Osborne/McGraw-Hill is along the same lines as the Morse book but at a higher level and with much more detail.

Leaving those volumes to cover the fine details, this article will only hit the high spots of 8086 programming, generally paying more attention to architectural considerations, while barely touching on the subjects of program development (editors, assemblers and linking loaders). Covered will be matters of structure and use of the 8086 registers (plus correspondence between 8086 registers and 8080 registers), the principles of memory segmentation provided by the 8086 architecture, the 8086 addressing modes, the status flags and some discussion of programming techniques for the 8086.

Registers

While the 8080 has only eight 8-bit registers and two 16-bit registers the 8086 has thirteen 16-bit registers plus a flag register (which could be a 16-bit register, except there are not sixteen flag bits). The 8086 registers can be logically considered in five separate groups:

1) general registers
2) pointer/index registers
3) segment registers
4) program counter
5) flag register

There are four registers in each of the first three groups, and the remaining two have one register each. The following paragraphs will discuss the programming considerations for each group and will describe the correspondence between these registers and the 8080 registers.

The *general* register group consists of the AX, BX, CX and DX registers. The approximate correspondence to 8080 registers is as follows:

AX	-	A
BX	-	HL
CX	-	BC
DX	-	DE

Note that while the 8086 AX register is a 16-bit register, its 8080 counterpart, the A register is only an 8-bit register. The AX register, like the others in this group, can be treated as either a 16-bit register or as an 8-bit register pair. For example, just as the 8080 BC register pair can be accessed as separate B and C registers, the CX register can be accessed as the 8-bit registers CH and CL (high order and low order, respectively). The following table shows the 8-bit register pairs which make up each of the 16-bit general registers.

16-BIT REGISTER	HIGH BYTE	LOW BYTE
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

(continued on next page)

The general purpose registers are the working data registers of the 8086. Adds, subtracts, and's and or's are performed on data here. These registers tend to be more symmetrical with respect to arithmetical operations than do the corresponding 8080 registers. While the 8080 requires that the A-register be the destination for arithmetical and logical operations, 8086 instructions exist, for example, for adding any two 16-bit or any two 8-bit general purpose registers and storing the result in either. The asymmetries that *do* exist are of interest, and, although all the peculiarities can not be treated here, some highlights will give readers an idea of what to expect.

The following table shows the relationship between certain operations and the general registers implicit in those operations:

OPERATION		IMPLIED REGISTER
Word Multiply, Word Divide, Word I/O	—	AX
Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic	—	AL
Byte Multiply, Byte Divide	—	AH
Translate	—	BX
String Operations, Loops	—	CX
Word Multiply, Word Divide, Indirect I/O	—	DX

In particular, the MUL (unsigned, integer multiply) and the IMUL (signed, integer multiply) implicitly use the AX, AH, AL and DX registers, depending upon which form of multiply is coded. In the case of a byte multiply the source operand is multiplied by the AL register, and the 16-bit result is placed in the AX register, with the most significant byte in the AH register and the least significant byte in the AL register. If a 16-bit multiply is to be performed (using the 16-bit versions of the multiply instruction) the source operand is multiplied by the AX register, and the 32-bit result is placed in the DX and the AX registers, with the most significant half in the DX register.

A similar situation exists with the DIV (unsigned, integer divide) and the IDIV (signed, integer divide) instructions. For the byte version of the divide instructions the byte source operand is the divisor and it is divided into the *sixteen-bit* dividend in the AX register, and the byte quotient is returned in the AL register, while the byte remainder is returned in the AH register. For the 16-bit form of the divide instructions the 16-bit source operand is divided into the *32-bit* dividend formed by the DX and AX registers (the DX register being the most significant part). The 16-bit quotient is returned in the AX register, and the 16-bit remainder is returned in the DX register.

The decimal adjust instructions AAA (ASCII adjust for addition), DAA (decimal adjust for addition), AAS (ASCII adjust for subtraction), DAS (decimal adjust for subtraction), AAM (ASCII adjust for multiply) and AAD (ASCII adjust for divide) all imply either the AL register or both the AH and AL registers.

As with the 8080, a particular register is implicit with input/output operations. The AX register is implied as the data destination/source whenever a 16-bit input/output instruction is coded. The AL register is, likewise, implied when a byte I/O instruction is encoded. Also, as with the 8080, the source/destination port may be named directly by the instruction (by coding the port address as immediate data) or the instruction can implicitly designate the DX register to contain the 16-bit value of the port address.

The XLAT (translate) instruction implicitly uses the AL and BX registers. The AL register is summed with the BX register to produce a 16-bit address. The byte value found using this address is then loaded into the AL register.

The CX register is used as a count register when string operations are coded with the *repeat* prefix. In this case the string operation is repeated until the CX register goes to zero.

The *pointer/index* register group consists of the SP (stack pointer), BP (base pointer), SI (source index) and DI (destination index) registers. While these registers can participate in the basic arithmetical operations (exceptions noted above), these registers are not meant for general computation and data manipulation but are used for computing addresses to point to data. The SP register is an exact parallel to the 8080 stack pointer. It contains a 16-bit address, which is automatically accessed for computing the location in memory of data to be stored or retrieved by push and pop operations, and it is automatically decremented or incremented as required. The remaining three registers in the group are a little more interesting, since they have no counterparts in the 8080 (or the Z80). The BP register serves as a 'base' from which relative data addresses are calculated. The SI and DI are auto-incrementing/decrementing registers used for accessing strings of sequential data. The particulars of these applications will be discussed later in the section on addressing modes.

The four *segment registers* of the 8086 provide the means of dynamically altering the physical address space of the processor to any 16-byte boundary within the available physical memory address space. How this is accomplished will be discussed in the section on memory segmentation to follow. The four registers involved are:

CS	CODE SEGMENT
DS	DATA SEGMENT
SS	STACK SEGMENT
ES	EXTRA SEGMENT.

While the members of the pointer/index register group are usually interchangable with the general registers as sources and destinations of data for data move, arithmetical and logical instructions, the segment registers are not. Special instructions are used for referencing data in these registers, and even then the registers are not treated equally by the instruction set. For example, there exists an instruction for moving data from memory or register to a segment register, but this instruction cannot be used to move data to the CS register. Additionally, there is a special instruction, LES (Load Register and ES from Memory) which loads sequential memory data into a specified register and into the ES register.

The *program counter* of the 8086 functions like that of the 8080. It contains the address of the next instruction to be fetched from memory, except that for the 8086 this 16-bit address is not the physical memory address for the function fetch, but is a 16-bit value used (in conjunction with the 16-bit contents of the Code Segment register) in computing the 20-bit physical memory address. Also like the 8080 program counter, the 8086 PC register is not available for general move and mathematical operations.

The *flag register* of the 8086 functions like the corresponding register of the 8080 with notable exceptions: 1) the 8086 flag register is a virtual sixteen bits long and contains extra processor status flags within the additional eight bits; 2) there are instructions for manipulating the 8086 flag register which have no counterparts in the 8080. The standard 8080 flags (carry, parity, auxiliary carry, zero and sign) are retained in the 8086 flag register, and they are in their traditional places. The following table shows the location of the 8086 flags within the flag register:

X X X X O D I T S Z X A X P X C.

The X's in the table indicate unused bits in the register and give rise to the thought that there may be some future expansion that makes use of these extra bit positions. As with the 8080, these unused bit positions are only manifested when the flag register is pushed onto the stack. Even then undefined values are placed into bits on the stack corresponding to the undefined bits in the flag register. Otherwise, the remaining letters stand for:

O	OVERFLOW
D	DIRECTION
I	INTERRUPT ENABLE
T	TRAP ENABLE
S	SIGN
Z	ZERO
A	AUXILIARY CARRY
P	PARITY
C	CARRY.

Familiarity with the standard 8080 flags is assumed here, and only the unique 8086 flags will be discussed in detail.

The overflow flag, when set, indicates a magnitude overflow in signed binary arithmetic. As such, it is the exclusive-OR of the carries into and out of the high order bit of the destination operand following an arithmetic operation. For example, when −1 (hexadecimal FFFF) is added to −4 (FFFC) there is both a carry into the sixteenth bit and a carry out. The exclusive-OR of these two carries is zero; there is no overflow. This is as one would expect, because the result, −5 (FFFB) is correct.

The direction flag is not a status flag in the sense of telling the results of some just-completed operation; instead, it is a control flag, whose setting determines the way an operation will be performed. The thing that it controls is the direction of scanning for string operations. When the direction flag is set then the SI and DI registers will be autodecremented, and strings will be processed from back (high order memory) to front (low order memory). When the direction flag is zero the SI and DI registers are autoincremented during string operations. There are two 8086 instructions for setting up the direction flag. They are CLD (clear direction flag) and STD (set direction flag). No other flags are affected by these instructions.

The interrupt enable flag, like the direction flag, does not indicate the results of some operation. It reflects the current state of the interrupt enable flip-flop, and it is cleared by the CLI (clear interrupt) instruction and set by the STI (set interrupt) instruction.

The trap flag also enables a processor state – the single-step mode – which is useful for program debugging. The trap flag is set by pushing the flag register onto the stack, setting the appropriate bit in the data at the top of the stack and popping the top of the stack back into the flag register.

Memory Segmentation

There are several reasons why memory segmentation is desirable: 1) *Total Address Space*. Since the 8086 is *only* a 16-bit processor the various data registers hold sixteen bits, and data transfers to and from memory are in 16-bit chunks; so it is most convenient to be calculating 16-bit addresses all the time. However, if the entire physical memory is limited to a 16-bit address space, then the maximum memory for a system would be 65,536 bytes (remember, the 8086 still addresses on byte boundaries, not 16-bit word boundaries). Segmentation provides a means of increasing the physical address space without requiring address pointers to be more than sixteen bits long. 2) *Modularity*. It is often desirable to segregate logically independent units (such as program sections, data sections and program stack areas) from one another, so they may be maintained independently. Memory segmentation, as implemented in the 8086, allows such units to be allocated their own, separate, logically and physically independent memory areas. 3) *Memory-Tasking*. Memory segmentation allows independent program tasks to be loaded simultaneously in memory with memory assignments being determined only at load time and completely independent of compile time address assignments.

As mentioned before, this facility is provided by the segment registers. These 16-bit registers provide implicit base values used in the calculation of physical memory addresses for the different processor operations. They provide the 8086 with the capability of performing true memory mapping, and they ultimately allow the processor to address a total of 1,048,576 bytes of physical memory. It works like this:

Whenever the processor requires to access memory, whether for an instruction fetch, for a data word/byte access or for a stack operation, it first calculates a 16-bit address according to the addressing mode coded for the particular instruction. This 16-bit address, however, is not used to directly find a location in physical memory. Instead, the processor next calculates the required physical memory address by adding the 16-bit address to *sixteen times* the contents of the segment register implied by the operation being performed. For example, if it is required to add the byte value at physical address 1,000,048 a pro-

(continued on next page)

grammer may, after assuring himself that the DS register will contain the value 62,500, code an ADD instruction that will result in a 16-bit address value of 48. The 16-bit value 48 (hexadecimal 0030) is added to the 20-bit value 1,000,000 (16 times 62,500, or hexadecimal F4240) to obtain the ultimate physical address of 1,000,048 (hexadecimal F4270).

Although it may seem like a lot of bother to manage all this, in reality most of the burden is taken off the programmer by carefully-designed assemblers, which perform all the immediate address calculation and keep track of what is where.

Each of the four segment registers is used for calculating memory addresses for a separate kind operation.

The Code Segment Register is used in calculating physical memory for every instruction fetch. The net effect is that the contents of the program counter are added to sixteen times the contents of the CS register to produce the physical memory address from which the next executable instruction is fetched. In actuality the 8086 processor is "pipelining" instruction fetches during sequential program execution. It is fetching bytes from program memory in advance of its current requirements but still based on the current PC value. Whenever a jump operation is executed the next fetch address is computed from the current PC contents, the pipeline is purged and prefetching to fill the pipeline resumes from that point on (until another jump is executed).

The Data Segment Register is used for direct memory references for data, such as adding memory to a register or incrementing memory, but there are specific exceptions as noted in the following two paragraphs.

The Stack Segment Register is used as a base for all stack operations. Also data addresses computed using the BP Register (see "Addressing Modes" later on) can use the Stack Segment register instead of the Data Segment Register.

The Extra Segment Register is used in conjunction with the DI Register in computing addresses for string operations.

These default segment register assignments can be superseded by the programmer through the use of the Segment Override prefix. More will be told about prefix bytes later.

Addressing Modes

The ability to access data (and instructions) in many and varied ways provides much of the power of modern 16-bit computer architectures. Programmers only accustomed to working with 8-bit machines, such as the 8080, may not be aware of the need for additional address calculation schemes. Those having experience with minicomputers wonder how they could live without them. Examine the options:

The 8080 provides *register implicit* (the data address is one of the processor registers, and the particular register is specified by the instruction itself), *direct* (the 16-bit memory address is in the two bytes following the instruction), *immediate* (the required data immediately follows the instruction in program memory) and *register indirect* (a register contains the 16-bit memory address required). The Z80 additionally provides *program relative* (an 8-bit immediate value locates the required address with respect to the current instruction) and *indexed* (an 8-bit immediate value is summed with the contents of a 16-bit index register to provide the 16-bit memory address). What more is required?

As an example of something lacking in the above addressing modes, take the case of a subroutine that must access the contents of a data table, whose starting address is passed to it by the calling routine. Since the subroutine must calculate displacements into the table, it is not feasible to use the Z80 index registers, since the relative displacements in that case must be precoded into the instruction as immediate values. What is usually done in this case is to have a section of code to first calculate the physical data address using the table address that was passed to the subroutine and then to place the calculated address into the HL register and finally to use the register indirect addressing mode to access the data. From the standpoint of machine efficiency it would be better to have an automated procedure for accessing such data. It is with such requirements in mind that the more advanced processor addressing modes are designed.

While discussing the various addressing modes of the 8086 it is helpful to have some insight into how these modes are coded into 8086 instructions. To begin, consider the 8086 instruction format:

[PP] II [MM] [DD...].

Each of the capital letters in the preceding indicates four bits (a hexadecimal digit) of program code. The square brackets around a set of letters indicate that the feature is not always present. PP is the prefix byte (discussed later). II is the basic instruction byte, and it is the only feature that is not optional. MM is the mode byte, and it is this feature that sets the addressing mode for the particular instruction. The reason the mode byte is not always present is that some instructions, such as CLI (clear the interrupt flag) do not access data and do not require that a data address be generated. Additionally, some instructions perform specific operations on register data and for efficiency are coded into single-byte form with the register specified by certain bits within the instruction. The DD bytes are variously immediate data and address displacements that are tacked onto the end of the instruction when the previous instruction bytes indicate that they are needed.

For instructions that do access data the instruction and displacement bytes have the following form:

```
INSTRUCTION    MODE
BYTE           BYTE

XXXXXXXW       MMRRRAAA
```

The X's are the basic bits of the instruction. W is a bit used in many instructions to differentiate between the byte form of the instruction and the word form. MM stands for the two mode bits (called "mod" for short hereafter). RRR stands for the three register bits (called "reg'), and AAA stands for the three-bit register/memory (r/m) field of the instruction. The following illustrates the meaning of these bit patterns:

W = 0	Instruction references byte data.
W = 1	Instruction references word data.

mod = 00	Specifies memory addressing with no displacement bytes. r/m specifies the particular addressing mode.
mod = 01	Specifies memory addressing with one displacement byte. r/m specifies mode.
mod = 10	Specifies memory addressing with two displacement bytes. r/m specifies mode.
mod = 11	Specifies register addressing. r/m specifies a register.
reg	Specifies which register is to be used. reg selects one of eight 8-bit registers or one of eight 16-bit registers depending on the W bit.
r/m	The three-bit value of r/m selects one of the eight addressing options.

Space limitations prevent giving exhaustive details of addressing mode selection, but the serious reader can pursue the matter to completion in the previously-referenced literature, particularly in Rector/Alexy. However, descriptions of the various addressing modes will be found useful.

There are two kinds of addressing of interest. First of all there is program memory addressing. Secondly there is data addressing, and data addressing can reference three different sources for data. Data references can be to processor registers, to computer memory and to input/output ports. In the following program memory addressing and data addressing are discussed separately:

Program Memory Addressing

Program relative addressing uses an 8-bit or a 16-bit immediate data item as a signed binary displacement with respect to the current instruction address. This is done by summing the displacement algebraicly with the PC register (the 8-bit displacement is sign-extended if negative). The CS register is not altered, so only references with respect to the current program segment can be made (no jumps to another another program segment).

Direct addressing uses two 16-bit immediate values to reload the PC register and the CS register to produce an effective jump to another program segment. The first 16-bit immediate value is loaded into PC, and the second is loaded into CS.

Indirect addressing uses any one of the standard memory or register addressing modes (to be discussed next) to extract either one or two 16-bit address values from a register or from memory. In the case of a single 16-bit address the value can come from either memory or from a register, and this address is loaded into the PC register to produce an intrasegment transfer. In the case of dual 16-bit address values only a memory reference is allowed. The first 16-bit value retrieved from memory is loaded into PC. The second 16-bit value is retrieved from memory sequential to the first, and is loaded into CS to produce an intersegment transfer.

Data Addressing

Register implicit addressing mode specifies the register of interest directly within the instruction byte. Some instructions, such as the multiply instructions, use always the same set of registers for data sources and destinations. Other instructions, such as the decrement 16-bit register instructions, have a separate form for referencing any one of the AX, BX, CX, DX, SP, BP, SI or DI registers, and the register of interest is encoded by three bits within the instruction byte.

Immediate memory addressing mode references the data item immediately following the instruction (and addressing mode bytes, if any) in program memory. The data value must be encoded into the program at assembly time and may be an 8-bit or a 16-bit value.

Direct memory mode addressing employs a 16-bit address value, which is coded into program memory following the instruction, to compute the memory address of the data. This 16-bit value is added to the DS register contents (times 16) to produce the physical memory address. Therefore the data item referenced will be with respect to the currently defined Data Segment.

Direct, indexed memory addressing mode employs an 8-bit or 16-bit immediate value (coded in program memory directly following the instruction), which is algebraically summed with either the DI or the SI register to produce a 16-bit address value. This value is then added to the DS register contents (times 16) to compute the physical address of the data reference.

Implied memory addressing mode is like direct, indexed memory addressing, except that no 8-bit or 16-bit displacement is provided. The SI or DI contents are added to the DS contents (times 16) to produce the physical memory address.

Base relative addressing mode is an extension of the previous three addressing modes. The contents of the BX register are added to the preliminary address, as calculated by one of these modes, then the resulting 16-bit address is added to the DS *or* SS contents (times 16). Allowing either the DS or the SS register to be specified allows addressing relative to either the current Data Segment or the current Stack Segment.

Instruction Prefixes

Something has been mentioned before about instruction prefix bytes. There are three such prefix bytes (two have multiple forms). The processor encounters a prefix byte at a point in its cycle where it would logically expect to encounter an instruction, therefore these are really one-byte instructions. However, they do nothing by themselves, but only affect the processing of sequential instructions.

The *LOCK* prefix asserts the processor "lock" status for the duration of the instruction following the LOCK prefix. This is useful in multiprocessor applications, and it prevents a companion 8086 processor, having access to shared memory, from accessing that memory during the "locked" instruction cycle.

(continued on next page)

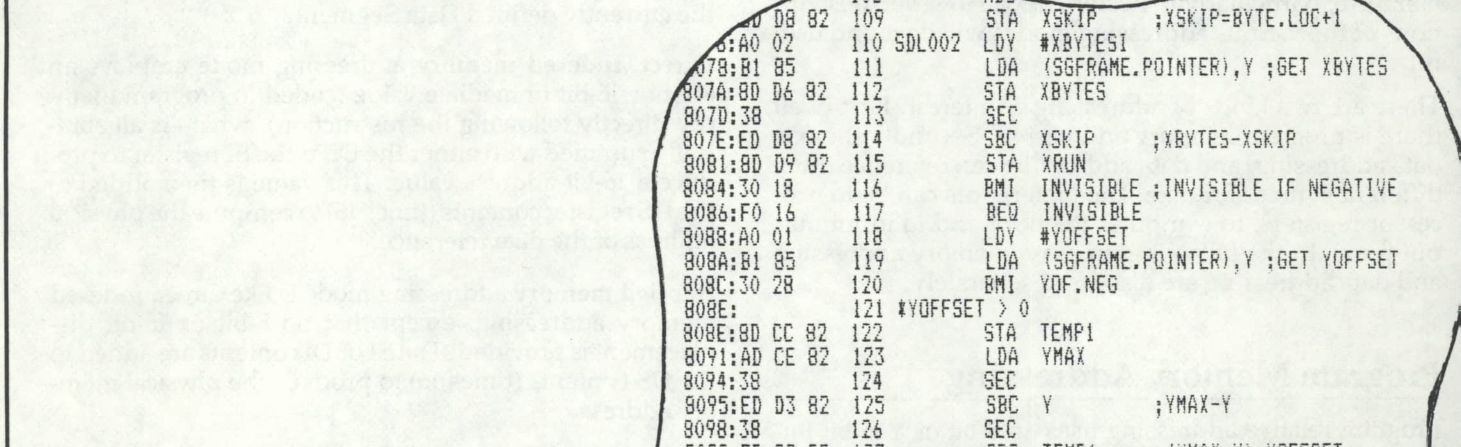
The *REP* prefix causes the string instruction immediately following to be repeated until the CX register has been decremented to zero. There are two forms for REP, and they are interchangable if the succeeding instruction is MOVS (move byte or word from memory to memory), LODS (load AL/AX from memory) or STOS (store AL/AX into memory). However if the following instruction is CMPS (compare memory with memory) or SCAS (compare AL/AX with memory) then, depending on which form of REP is coded, the string operation will be terminated on *zero* or *non-zero* result.

The *SEGMENT* prefix has four different forms – one to specify each of the four segment registers. The implied segment register for computing a *data* address for the following instruction is specified by the SEGMENT prefix. This allows the default segment register for the instruction to be overridden by the programmer and allows, for example, a data item to be retrieved from the current code segment.

Summary

The 8086 is obviously a much more sophisticated processor than the 8080 (and even the Z80) with about three times as much architectural detail to be mastered. The additional addressing modes of the 8086 and the symmetry of operation of the data registers are ideally suited for use by high level languages, such as Pascal and FORTRAN. However, assemblers available for the 8086 help to ease the programmer's burden of keeping up with the elaborate addressing schemes provided, and a serious programmer will not shy from using the 8086 in this mode.

The preceding is meant to give an overview of the subject matter, and much detail indispensible for actually programming the 8086 has been omitted for the sake of brevity. Once again, the serious programmer is encouraged to acquire one or more of the books referenced above (along with a set of proper program development aids) and to move up into the world of 16-bit programming.


E=mc²
COMPUTERS ARE SO POWERFUL BECAUSE
THEY ARE SO MUCH FASTER THAN US
©Rick '82

```
                            ADD
 D D8 82                    STA  XSKIP      ;XSKIP=BYTE.LOC+1
6:A0 02          SDL002     LDY  #XBYTESI
078:B1 85                   LDA  (SGFRAME.POINTER),Y ;GET XBYTES
807A:8D D6 82               STA  XBYTES
807D:38                     SEC
807E:ED D8 82               SBC  XSKIP      ;XBYTES-XSKIP
8081:8D D9 82               STA  XRUN
8084:30 18                  BMI  INVISIBLE ;INVISIBLE IF NEGATIVE
8086:F0 16                  BEQ  INVISIBLE
8088:A0 01                  LDY  #YOFFSET
808A:B1 85                  LDA  (SGFRAME.POINTER),Y ;GET YOFFSET
808C:30 28                  BMI  YOF.NEG
808E:            *YOFFSET > 0
808E:8D CC 82               STA  TEMP1
8091:AD CE 82               LDA  YMAX
8094:38                     SEC
8095:ED D3 82               SBC  Y          ;YMAX-Y
8098:38                     SEC
8099:ED CC 82               SBC  TEMP1      ;(YMAX-Y)-YOFFSET
809C:            *UNSIGNED COMPARE: YOFFSET>YMAX-Y?
809C:B0 06                  BCS  SDL003     ;JMP IF VISIBLE
809E:A9 00       INVISIBLE LDA #0           ;SGFRAME NOT VISIBLE
80A0:8D DC 82               STA  VISIBLE.FLAG
80A3:60                     RTS             ;DATS ALL...
80A4:AD D3 82    SDL003     LDA  Y
80A7:18                     CLC
80A8:6D CC 82               ADC  TEMP1      ;Y+YOFFSET
80AB:8D DA 82               STA  YLOC
80AE:A9 00                  LDA  #0
80B0:8D DF 82               STA  YSKIP
0B3:4C CC 80                JMP  SDL004
```

Product Status Reports

The new software products and new versions described below and on page 36 are available from their authors, computer stores, software publishers, and distributors. Information has been derived from material supplied by the authors or their agents, and *Lifelines/The Software Magazine* can assume no responsibility for its veracity. Software of interest to our readers will be tested and reviewed in depth at a later date.

New Products

ACTIVE SOFTWARE MARKETING

Electrical Panel Schedule Program is a CP/M program for electrical power engineers that simplifies compliance with Article 2000 ('Branch 1 and Feeder Calculations') or the National 1 Code. It automates panel schedule record keeping and calculations while significantly reducing labor costs for engineering and drafting.

The operator can add, change or delete loads, breakers and load descriptions for each panel (including subpanels or feeds) and can be printed in a form suitable for publication with final construction documentation. Variations from "standard" calculations are allowed.

A complete system including computer, terminal, printer, word processor and the PANEL software is $4995. PANEL software alone for any CP/M computer is $1500.

BUSINESS SOFTWARE CORP.

Group Benefits Shoppers™, a software package which allows large agencies or individual agents to shop the market for group health insurance for clients with one to fifty employees is now available for microcomputers using the CP/M operating system and the WANG 2200 series computers. The user selects the insurance carriers he desires and is then able to shop against deductibles, stop loss, maternity, dental RX card, supplemental accident, takeover provisions and industry classification. A complete set of supporting utilities and software maintenance contracts are also available.

CERES SOFTWARE INC.

CERES I is an interactive screen design and process utility for CP/M based systems that interfaces with many popular COBOLs. It consists of two major components, the interactive design screen utility and the run-time system.

With CERES I's interactive screen design utility, screens are designed directly on the terminal. There is no need to calculate the line number and column of your fields. Fields can be inserted, deleted, moved or copied on the screen for fast and easy screen design and maintenance. Attributes are defined for each field or default attributes may be used.

CERES I's run-time system accepts, formats, displays and extensively validates data. Some of its features include backup by field or character for data entry correction and help messages for data entry clarification.

Suggested retail price is $300. A manual alone is $25.

ELEKTROKONSULT AS

Disk Utilities for the Osborne-1

Elektrokonsult has developed a series of advanced Disk Utilities for the Osborne-1. Their main purpose is to aid the user in avoiding loss of data from disks.

DTEST tests disks for bad spots. Bad sectors are collected in a write-protected file, which reduces the probability of disk crashes. UNERA recovers accidentally ERAsed files. DDUP automatically recovers files with damaged sectors. DDMP enables you to examine and patch data on any sector, making it possible to repair crashed disks and to recover lost data.

The utilities will work on SD or DD disk and on many other CP/M systems. Each program is $29.95 or $99.95 for all four (plus $8 air shipping charges).

File Mover

For CP/M users with two or more CP/M computers with different disk formats, File Mover is a CP/M to CP/M file transfer utility which can transfer any type of CP/M files–including program files–from one computer to another over a serial link. It uses an error detection and correction protocol with checksums and automatic retry to ensure error free file transfer. File Mover runs on 8080, 8085 and the Z80 processor and can be used with baud rates up to 9600. It can easily be installed to run on most CP/M computers with a serial interface. Complete instructions are included. It is available on many popular disk formats, including 8" SS/SD, Osborne, Rainbow and Zenith (hard- and soft-sectored). Cost is $59.95 plus $8 for shipping and handling.

LIFEBOAT ASSOCIATES

C-Food Smorgasboard

A collection of useful functions and utility programs for low-level I/O, terminal independent I/O; decimal arithmetic, disk directory searching, direct interface with IBM-PC ROM BIOS for video and asynch port control, BDS "C" compatibility.

Executive Alert System

Decision support system for execu-

(continued on next page)

tives. Tracks key indicators of company performamce, does projections, year-to-date totals. Information is presented in raw numbers or bar charts on screen or IBM printer. Twenty-six key indicators are predefined, but user can replace them with his own or add more. Requires IBM monochrome monitor, 64K ram. Cost $250.

PACIFIC DATA SYSTEMS, Inc.

Moneytrack, a money management program for small computers that meets the standards of professional accountants is now available from Pacific Data Systems.

Moneytrack has the capacity to maintain complete transaction records for small businesses, farms or investments as well as the owner's personal accounts. It prepares a variety of reports to help met the requirements of financial institutions and the Internal Revenue Service, prints checks on different check forms and greatly simplifies the job of bank reconciliation. Users include owners of small businesses or farms, investors with many interests, small or large accounting firms, financial advisors, business managers and professionals such as doctors and lawyers.

Moneytrack is a complete package with its own operating system so it is ready to use as soon as the disk is inserted in the computer's disk drive. No knowledge of programming or computers is needed. It runs on the IBM Personal Computer, requires a 64K memory and dual 320K disk drives. Suggested retail price is $450.

SELVA SYSTEMS INC.

GL/M is a graphics language for microcomputers operating under CP/M which turns popular word processing printers into intelligent graphics output devices.

Designed for the individual who must produce information graphs rather than large presentation graphics, GL/M is a graphics language tool with an easy-to-learn instruction set and syntax for microcomputers. Applications include not only bar, line and pie charts, but also flow charts, histograms, figures and illustrations.

Graphs are easy to alter, as single commands in a GL/M graphics file can change size, aspect ratio and other characteristics. Ease of placement allows small, thumbnail graphs to be integrated with text. The software's output drivers are optimized to print quickly and unidirectional printer movement eliminates paper slippage at the lower edge of single sheets.

GL/M requires a microcomputer with 64K, CP/M, two disk drives and a printer. List price is $295.

TRIGRAM SYSTEMS

DES-Crypt™ is a software implementation of the National Bureau of Standards data encryption standard (DES) algorithm. It protects the privacy and integrity of the information contained in any file. Possible applications include protecting confidential financial information, patient records, student grades, software, databases, electronic mail, sensitive text, etc.

DES-Crypt is a low cost alternative to dedicated encryption hardware. It works with any file and includes functions for encrypting, decrypting, verifying encryption, data authentication, destroying plaintext, creating random hexadecimal keys and listing files.

It is an easy-to-use menu-driven system with extensive error checking and on-line help; no special knowledge is required to use it effective. The data authentication function can be used independently of encryption, so files can be protected against accidental changes or deliberate tampering without preventing legitimate access.

Safety features include: automatic maintenance test, checksums on hex keys to detect typing errors, redundant key entry, screen echo may be disabled when entering keys, first block of all ciphertext files is automatically verified during encryption. The user can verify the integrity of any of the programs or files using the data authentication function.

DES-Crypt is available for 8080/8085/Z80 systems running CP/M-80 and for the IBM-PC and other 8086/8088 systems running MS-DOS (PC-DOS) or CP/M-86.

New Versions

for CP/M-80

1. ASCOM-80 v 2.23
2. DATASTAR v 1.4
3. dUTIL v 1.1a
4. Precision BASIC v 1.6
5. Priorities v 1.11
6. Quickcode v 2.1a
7. UNICALC-80 v 2.3

for CP/M-86

1. CB 86™
2. PL/Mycro-86 v 1.17 w/o FLOATING POINT COMPILER
3. PL/Mycro-87 v 1.17 and v 2.07 w/FLOATING POINT COMPILER
4. PL/I-86

for PCDOS

1. ASCOM-86 v 2.23
2. AUTOSORT/86M v 1.08
3. FABS/PC v 1.06
4. LATTICE "C" Compiler v 1.03
5. muMATH/muSIMP-86
6. dBASE-II/PC v 2.3d
7. UNICALC/PC v 2.6

for MS-DOS

1. ASCOM-86 v 2.23
2. AUTOSORT/86M v 1.08
3. FABS/86M v 1.06
4. LATTICE "C" Compiler v 1.03
5. UNICALC-86 v 2.6

New Publications

Let the Government Pay for Your Computer, Vernon K. Jacobs, C.P.A. Research Press Inc.

An eleven-page, single-spaced report which describes a number of ways buyers of new desktop-size computers can justify tax recovery. Possibilities and limitations are discussed.

My Computer Understands Me: A Complete Guide to Computers for the Dental Office, by Barry Garson, D.D.S. Published by the author. $40.

(continued from page 9)

FIGURE 1

```
        LXI     H,I      ;load starting point
        SHLD    I%       ;save it
        JMP     @001     ;enter the loop
                         ;actual loop follows
@002:
        LHLD    I%       ;load loop pointer
        INX     H        ;step it
        SHLD    I%       ;save loop pointer
@001:LXI        D,64535  ;load max legal integer minus loop size
        DAD     D        ;and add it to loop pointer
        JNC     @002     ;loop if no overflow
```

FIGURE 2

```
        LXI     D,A.REAL     ;LOAD POINTER TO INDEX VARIABLE
        LXI     H,CODE(1)    ;LOAD POINTER TO LOOP START VALUE
        CALL    ?TRMM        ;ASSIGN BEGINNING VALUE TO A.REAL
        JMP     @000         ;ENTER LOOP
@001;
        LXI     H,1          ;LOAD STEP VALUE AS INTEGER
        CALL    ?CRSH        ;CONVERT TO REAL AND RETURN IN D & E
        LXI     H,A.REAL     ;LOAD POINTER TO INDEX VARIABLE
        CALL    ?ARSM        ;LIBRARY ROUTINE TO ADD REAL NUMBERS?
        LXI     H,A.REAL     ;RELOAD POINTER TO INDEX VARIABLE
        CALL    ?TRMS        ;LIBRARY TO SAVE REAL NUMBER?
@000    LXI     H,1000       ;LOAD LOOP LIMIT
        CALL    ?CRSH        ;CONVERT TO REAL NUMBER-RESULT IN D&E
        LXI     H,A.REAL     ;RECOVER POINTER TO INDEX VARIABLE
        CALL    ?CRMS        ;LIBRARY COMPARE OF REAL NUMBERS ??
        JP      @000         ;LOOP IF NECESSARY
```

FIGURE 3A

```
rem ************************************************************
    rem function to convert JOHN L. JONES to John L. Jones
    rem toggle to lower case except 1st char & after "  ".
    rem written by R. VanNatta 10/24/82
rem ************************************************************
def lowcase$(x2)                                REM FUNCTION NAME
    string x2$,lowcase$,scratch$,segment$       REM DECLARATIONS
     integer last.character%,character%,pointer%,flag%
    scratch$ = left$(x2$,1)                     REM DON'T CHANGE\
                                                FIRST CHAR.
    FLAG% = 0
    last.character% = len(x2$)                  REM FIND END OF LINE
    for pointer% = 2 to last.character%         REM SET LOOP
        segment$ = mid$(x2$,pointer%,1)         REM EXTRACT CHAR.
        character% = asc(segment$)              REM GET DECIMAL VAL
        if character% >= 65 \                   IF UPPER CASE CHAR
        and character% <= 90\
        and flag% = 0 then\
            segment$ = \                        FLIP BIT TO LOW CASE
            chr$(character% + 32)
        if character% = 32 then\               IF SPACE SET FLAG
            flag% = 1\                          FOR NEXT LOOP
                else\                           OTHERWISE
            flag% = 0                           REM CANCEL FLAG
        scratch$ = scratch$ + segment$          REM REBUILD STRING
    next pointer%                               REM LOOP
    lowcase$ = scratch$
fend
```

FIGURE 3B

```
rem ************************************************************
    rem function to convert JOHN L. JONES to John L. Jones
    rem demonstrates nested IF-THEN
    rem R. VanNatta version 11/12/82
rem ************************************************************
def lowcase$(x2$)                               REM FUNCTION NAME
string x2$,lowcase$,scratch$,segment$           REM DECLARATIONS
integer last.character%,character%,pointer%,flag%
scratch$ = left$(x2$,1)                         REM SKIP 1ST CHAR
last.character% = len(x2$)                      REM FIND END OF LINE
flag% = 0
for pointer% = 2 to last.character%             REM SET LOOP
    segment$ = mid$(x2$,pointer%,1)             REM EXTRACT CHAR.
    character% = asc(segment$)                  REM GET DECIMAL VAL
    if character% <= 65 THEN\                   IF LOW RANGE CHECK
        if character% = 32 then \               FOR SPACE (CHR(32))
                                                and
            flag% = 1\                          SET FLAG
                else\                           OTHERWISE RELEASE
                                                FLAG
            flag% = 0\                          AND SKIP TO END
                ELSE\
        if flag% = 1 then\                      IF FLAG SET
            flag% = 0\                          RELEASE IT AND SKIP
                else\                           OTHERWISE
            if character% <= 90 then\           TEST HIGH RANGE
                segment$ = chr$(character% + 32)
                                                REM FLIP BIT
    scratch$ = scratch$ + segment$              REM REBUILD STRING
next pointer%
lowcase$ = scratch$
fend
```

FIGURE 4

```
REM ************************************************************
    REM THIS FUNCTION WILL FLASH MESSAGE IN INVERSE VIDEO IN
    REM CENTER OF LINE 3 OF SCREEN
    REM SEE OCTOBER 1982 LIFELINES FOR ADDITIONAL EXPLANATION
REM ************************************************************
common clear.screen$,addr.cursor$,reverse.video.toggle$,\
    reset.video.toggle$,delete.to.end$
def at$(x1%) external                       rem function for abs
    string at$                              rem cursor position
fend
def inverse$(x2$) external                  rem FN for reverse
    string inverse$                         rem video of string
fend
def flasher(message$) public
integer index,delay                         rem declare variables
for index = 1 TO 6
    call at$(280-(len(message$)/2))         rem position cursor
    if mod(index,2) = 0 then\               rem set up flip flop
        print delete.to.end$.\              rem delete on odd numbers
    else\
        print inverse$(message$)            rem print on even numbers
    for delay = 1 TO 10000 :next delay      rem delay
    next index                              rem do it 6 times
fend                                        rem return
```

LIFELINES™/The Software Magazine™
1651 Third Avenue, New York, New York 10028

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

[redacted]