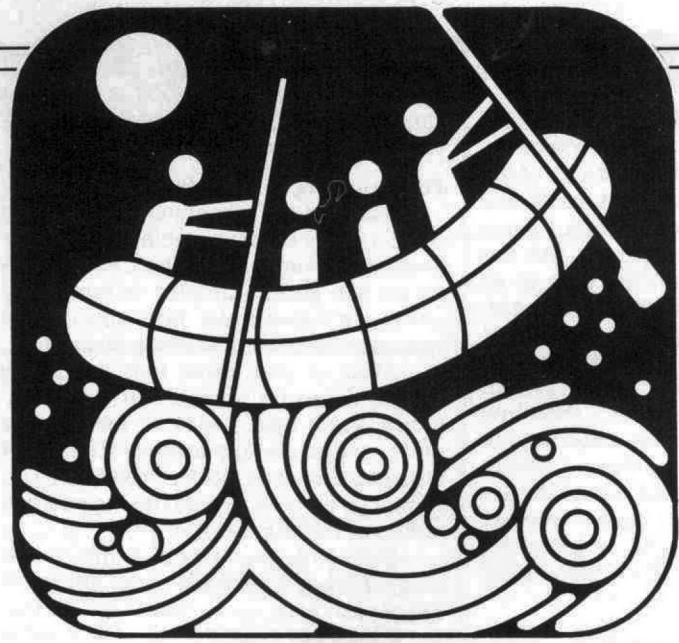


Z-100

SURVIVAL KIT

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

I guess you could say that this column was prompted by a slip of the tongue. It's not the first time I've put my foot in my mouth, but it is the first time I've had to pay my dies by writing a regular column for a magazine! Let me explain . . .

I own a company that publishes software for the Heath/Zenith Z-100 and PC compatible computers. As a part of that business, I send out a regular newsletter to my customers. From time-to-time (like every issue) I may make editorial remarks in the Newsletter, and I usually don't pull any punches when it comes to taking potshots at something that irks me. It's pretty well known around these parts that I am an avid Z-100 supporter, and most of the time, the Customer Newsletter reflects this minor prejudice.

Well, a couple of issues ago, I made some brash remarks to the effect that I didn't believe that REMark and SEXTANT were giving the Z-100 its fair share of press anymore. A copy of that issue of the Newsletter made the rounds at the HUG office, and made a generally unfavorable impression. My phone rings, and it's Pat Swayne. Pat said that he had read the Newsletter, and was wondering if I wanted to do something about the lack of Z-100 articles. I said "sure", and the rest is history. The 'something' he was talking about was writing this column.

What's This About?

One of the goals I have in writing this column is to prevent the Z-100 from dying a premature death. The verdict was pronounced when Zenith decided not to make any more Z-100s, but how long it takes for the sentence to be carried out will depend on support of the user community. Not too many would argue the fact that the Z-100 was a state-of-the-art machine when it was first introduced. That was back in 1982, around the same time as the first IBM-PC was released. But the Zenith machine was technologically superior to the PC in almost every respect. Many years rolled by before the PC compatibles began to match the Z-100's computing power and graphics. Even today, some people would argue about the relative strengths of the Z-100 and the PC clones. But anyone who is being totally objective about the situation will admit that technology has begun to overtake the Z-100. (Have you looked at a new Z-386 with VGA graphics lately?)

Just because the Z-100 can't quite keep up with the new kids on the block, there is no reason to throw it on the scrap heap. Unless, of course, you have money to burn. I decided quite a number of years ago that trying to keep up with the latest in computers is a never-ending quest. As soon as you think you have the fastest, meanest, and prettiest system on the market, along comes another that'll outdo your new pride and joy. You have to draw the line somewhere, and stick with a system that will get the job done. For many Heath/Zenith users, the Z-100 is where they draw the line.

Count the Reasons

There are actually quite a few reasons why you might want to hang on to your trusty old Z-100. Consider, for instance . . .

1. If you get a new computer, you'll probably have to buy new software. That means you'll have to learn how to use it all over again.
2. I have NEVER seen another computer with a keyboard that can compare with the one attached to my Z-100.
3. After several years of use, you have all the bugs and glitches worked out of your Z-100. Would you really want to go through that with a new computer again?
4. There should be a good replacement parts market for some years to come. If all else fails, you can buy a used fully-loaded Z-100 for less than a bare-bones PC clone.
5. If you're like me, your Z-100 has become like an old friend. How could you even think of trading it for one of those new-fangled AT clones?
6. The Z-100 is paid for . . . need I say more?

What I'm trying to say here is that the Z-100 you already own may be all the computer you'll ever need. It's a good machine. It may even be considered a classic some day. However, if someone offered to trade a Z-386 with FTM monitor for my Z-100, the ole' Z-100 would be gone in a heartbeat. There's a point where sentimentality has to end.

Where Do We Go From Here?

This first installment of "Z-100 Survival Kit" is just sort of an introduction. I won't get much done other than saying hello and outlining some ideas for future columns. Since this is a new column, its direction and emphasis have not yet been

determined. Your input will help me decide where to go from here.

As I mentioned at the start, I own a business which is a vendor to the Heath/Zenith market. In many ways, this will have a positive influence on this column, because I make my living supporting the Z-100. This means that I am familiar with the pros and cons of the Z-100, and am experienced with the machine from a software and hardware standpoint. I have to keep abreast of all the latest happenings that concern the Z-100. But there is a drawback to being in this position, because I will be somewhat limited in my ability to review commercial software that is available for the Z-100.

I expect that this column will tend to be more technical in nature than the average REMark article. And I also would like to put more emphasis on programming, with lots of code examples. If the feedback I get from Z-100 owners is correct, they make up the largest number of REMark readers who are still interested in recreational computing. Most business users have moved on to PC, AT or '386 compatible machines by now. And the 8 bitters are in a world all of their own.

This doesn't mean that the whole column is going to be targeted toward hard-core programmers. I'll try to get a good blend of stuff which is interesting to the average user. If you own a Z-100, you've probably had it for at least a couple of years, so you shouldn't need anybody to tell you what an AUTOEXEC batch file does. I think Z-100 users would like to have some technical stuff they can get their teeth into. If I'm wrong, I'm counting on you to let me know.

I encourage you to write and let me know what you think this column should be like. Be sure to include "Z-100 Survival Kit" at the top of the address, so I can keep this stuff sorted out of the normal business mail. One thing in particular that I'd like to encourage is questions you may have about your Z-100 (software or hardware). I'm not too sure there are many good sources for Z-100 specific information anymore, so I'd like this column to help fill that void. If I don't know the answer to your question or problem, chances are I can find someone who does. I'll try to answer any questions with a personal reply, and publish the most interesting ones (or ones of general interest) in this column. If I begin to get more letters than I can respond to . . . well, we'll cross that bridge if we come to it.

There are a couple of subjects I would like to avoid as much as possible in this column, even though they are specific to the Z-100. I don't want to get too deeply involved in ZPC patches and that kind of stuff. Not that I don't think it is important . . . but I prefer to leave that area to Pat Swayne. This is not to say that ZPC patches won't ever be mentioned here. I just don't want this column to turn into a "ZPC Update" clone.

The other thing I definitely will avoid like the plague (unless you tell me otherwise) is using CPM on the 8-bit side of the Z-100. Face it . . . CPM is extinct. If you like CPM better than DOS, then I guess you'll just have to be content with the thought that all good things come to an end sooner or later. The idea of putting an 8085 processor in the Z-100 in the first place was only intended to bridge the gap until 16-bit software became available. Who could have guessed back then that with the concurrent introduction of the IBM-PC, the new 16-bit software (which ran on the 8088 processor) would be developed so quickly? I doubt whether most Z-100 users have ever used anything other than MS-DOS (or Z-DOS) on their machines.

The PC Compatibility Question

There seems to be a lot of interest these days in trying to make the Z-100 as PC compatible as possible. Hardware emulators, Pat Swayne's ZPC software, the ZHS circuit board, PC style COM ports, and on and on. From a purely technical computing standpoint, all of these modifications are attempting to transform the Z-100 into an inferior machine. Sort of like paying a mechanic to 'untune' your car. Is PC compatibility really that important? Or is everyone just falling in line with the trend of trying to be compatible?

In the next installment of this column, I'll talk about the issue of PC compatibility as it relates to the Z-100. We'll look at some reasons why you should be concerned about compatibility, as well as situations where you would be better off sticking with your native Z-100. We'll discuss some of the different levels of compatibility which are available with the Z-100, and how each can be used to advantage. And I'll also show you how you can write programs which will run on either the Z-100 or PC compatible machines, and describe some different techniques for writing 'portable' code.

Probing the Monitor ROM

Another subject on the agenda for a future column will be a close look at the Z-100's MTR-100 monitor ROM program. This 'program-on-a-chip', which is included with every Z-100, is a gold mine of valuable routines for assembly language programmers. And the MTR-100 data segment holds system information that can be accessed from any language. The Z-100's monitor ROM chip should be a valuable tool for anyone writing Z-100 specific programs, particularly in light of the fact that the Z-100 is out of production — this means that there will not be any more revisions to the ROM firmware.

In this upcoming column I'll show you how to find the entry points to the monitor ROM, and how to use some of the routines. We'll also look at how you can access the MTR-100's data from different programming languages.

Programming the Hardware

Many of the peripheral interfaces in the Z-100 are programmable devices. The keyboard controller and the CRT controller are two good examples. In future columns, we'll investigate how you can do special tricks by programming these chips.

Most of you know that the Z-100 keyboard can be operated in 'up/down' mode. We'll take a look at how that is done, and some applications that might require this special mode. I'll also give you some tips on how to read the keyboard in polled or interrupt mode.

Did you know that the Z-100 can be programmed to have just about any number of scan lines, up to 500 or so? Or that the number of characters per line can be changed? Did you ever wonder how to write a program that uses the interlace mode of the Z-100. We'll look at these, and other things, that can be done by programming the CRT controller.

And Graphics

One of my main interests when it comes to computers, in general, and the Z-100, in particular, is graphics. So I'll devote a fair amount of time in upcoming columns to graphics applications. After all, the Z-100 was designed to be a graphics machine. It was one of (if not THE) first computer I know of that allowed graphical information to coexist on the screen with text. No special graphics mode required — just mix the lines, circles, and colored areas right in with the text characters — a pretty revolutionary idea way back when. Even today, the Z-100 is able to keep pace with the newer machines when it comes to graphics capability. A Z-100 with a Hughes 16 color V1 board, running in interlace mode (or with ProScan video) is equivalent in resolution and number of colors to the IBM VGA (virtual graphics array) standard. At this stage in the game, the main advantage held by the 80286 and 80386 machines is in speed. A pixel graphics or CAD program running on a Zenith Z-386 is going to be a lot snappier than the same program on a Z-100. But then, some of us have more time than money, right?

In Conclusion

I'll try to keep you abreast of what's new for the Z-100, and show you some of the tricks of the trade in programming ideas. I'd like this column to become a clearing house for information and support for the Z-100, so if you know something special about the Z-100, drop me a line, and I'll get the news out to the rest of the troops. I hope you have grasped by now that the purpose of the "Z-100 Survival

Continued on Page 27

Voice: (215) 387-4614 or 387-5572
BBS: (215) 387-4635 or 288-0262
Contact: Colin McGowan
Times: 2nd Wed of Month, 7pm-9:30pm
Location: 1st & 2nd Month each quarter
Glading Memorial Church
Loretta & Cheltenham Ave.
Philadelphia, PA 19124
(at Oxford Circle near Rt 1)
3rd Month each quarter
Philadelphia H/Z C&E Store
Roosevelt Blvd. & Bustelton
Philadelphia, PA 19124
(215) 288-0180

Pittsburgh HUG (PGH-HUG)
Heath/Zenith Electronics Center
3482 William Penn Highway
Pittsburgh, Pennsylvania 15235
Voice: (412) 824-3564
BBS: (412) 824-3732
Contact: Phillip Sidel, President
(412) 648-7384
Times: 3rd Tuesday of Month, 7pm
(Except June, July, December)
Location: Heath/Zenith Elec. Center

South Carolina

Anderson HUG
401 Tiffany Drive
Anderson, South Carolina 29625-1815
Voice: (803) 225-0084
BBS: N/A
Contact: John R. Miller
Times: Call for Time/Date
Location: Call for Location

Texas

Dallas/Forth Worth Heath/Zenith Users' Group
12022 C Garland Road
Dallas, Texas 75218
Voice: (214) 327-4835 (Store)
BBS: (214) 742-1380
Contact: Jon Gauthier (214) 997-0157
Charles Horn
Times: 1st Tuesday of Month, 7:00pm
Location: Heath/Zenith Elec. Center
12022 C Garland Road
Dallas, TX 75218

San Antonio Heath Users' Group
7111 Blanco Road
San Antonio, Texas 78216
Voice: N/A
BBS: (512) 341-0586
Contact: N/A
Times: 1st Wednesday of Month, 7:30pm
Location: San Antonio H/Z Elec. Center
7111 Blanco Road
San Antonio, TX 78216

Virginia

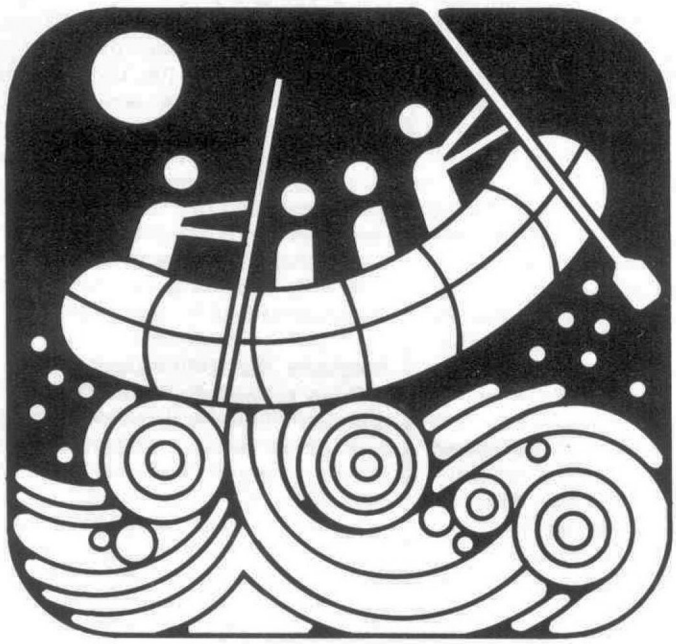
Tidewater Heath Users' Group (THUG)
c/o Heath/Zenith Electronics Center
1055 Independence Boulevard
(Haygood Shopping Center)
Virginia Beach, Virginia 23455
Voice: (804) 460-0997
BBS: N/A
Contact: Roy Hartley — (804) 428-9205
John Smith — (804) 468-6246
Times: 1st & 3rd Thursday of Month,
7:30pm
Location: Heath/Zenith Elec. Center

Wisconsin

Milwaukee Heath Users' Group (MHUG)
4439 N. Marlborough Drive
Shorewood, Wisconsin 53211
Voice: N/A
BBS: (414) 548-9866
Contact: Leif Pedersen, President
(414) 547-7966
Albert Kalina, Treasurer
(414) 962-2699
Times: 3rd Saturday of Month, 1-4pm
Location: Milwaukee School of Engineering
Walter Schroeder Lib. Bldg
Room L100
500 E. Kilbourn Avenue
Milwaukee, WI *

Continued from Page 16

Kit" is to help you get the most mileage out of your Z-100 computer. I'll not try to talk you out of getting a new machine, if that's what you have your heart set on, but for those of you who aren't ready to take it to the barn, stay tuned . . . relief is on the way! *



Z-100 SURVIVAL KIT

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

Hello again! I was hoping my introduction to "Z-100 Survival Kit" in the last issue would arouse your interest. Glad to see you're back! We didn't cover much serious ground in that first installment, but now that we have the formalities out of the way, we're ready to get down to business.

As promised, we are going to take an overall look at the question of PC compatibility for the Z-100. Over the past year or two, the goal of being compatible with, and emulating the PC's has received a lot of attention. Is it really that important? Why is it so difficult to achieve? Is it worth the effort?

Beyond compatibility, we'll look at software that runs on both the native Z-100 and PC compatibles. I'll also show you how you can write your own programs that run on either type of machine. There's a lot to talk about, so let's get started . . .

The PC Compatibility Question

To someone unfamiliar with the personal computer scene, it must seem strange that one of the main pursuits is trying to get your computer to do the same thing that everyone else's does. Particularly when the machine you have is superior to that owned by your neighbors. And from a purely hobbiest viewpoint (programming being the hobby), it is ridiculous. Give me a Z-100, and I'll write a program that will run rings around the nearest PC/XT clone.

To understand the quest for compatibility in the proper light you must realize that most computer users aren't interested so much in how wonderful or powerful their computers are, but in what the machine can do to help them with practical day-to-day problems. Things like balancing the books, keeping track of inventory, analyzing statistical data, or even drawing pictures. You quickly find that the power behind the machine is in the software. If nobody has written a program for your computer that does what you need done, then you're just out of luck.

A little history . . . In the beginning (circa 1975) most personal computers were created about equal. Hardly any of them would run the same software, because they were all different. The only unifying influence was the CPM operating system, but it was far from a standard (by today's standards). For the most part, if you wanted software to do a specific task, you had to write it yourself, or hire it done. As the years went by, the situation really didn't change much until IBM announced their own personal computer. And at that moment, the computer industry forever changed (for better or worse). Amid protests that IBM had stymied technological innovation, a new standard had been set. My use of the word "standard" here should not be taken as in "standard of excellence", but should be understood as meaning "something to conform with". Whatever your viewpoint about the IBM-PC revolution, you will have to admit that it caused the computer industry to start making computers which all looked and acted vaguely the same.

One of the main consequences of this turn of events is that software companies now had a stationary target to aim at. Where once there were just a few medium-sized software companies (mostly companies selling operating systems and development tools), now new companies sprang into existence almost overnight. Companies like Microsoft and Lotus Development began to show Wall Street that computer software was big business. As software development became more sophisticated to meet the needs of an ever-demanding audience, small companies began to be edged out of the software business because they couldn't compete.

Back to the present, we find that most of the flashy high-powered software we would like to use is sold by million dollar software companies (they're the only ones who have the resources to produce it). And, naturally, being in business for a profit, they only write software for computers that have a substantial user base. Which means mainly PC Compatibles and the Apple MacIntosh. All others need not apply, including the Z-100.

Why Do We Want Compatibility?

The obvious reason for wanting PC compatibility is to take advantage of programs that aren't available for the Z-100. There are literally tens of thousands of

software titles available for the original IBM-PC and its descendants, while the list of commercial software written specifically for the Z-100 is composed of only a few hundred programs.

But there are other reasons you might consider compatibility an issue. One of the most important of these is the ability to use the software interchangeably on a Z-100 and PC compatible computer. Many of you now own a PC compatible computer, as well as a Z-100. You'd like to be able to use the software on both machines, right? (We won't mention software license agreements here). There are two ways of doing this . . . buy programs that run on either machine, or fix the Z-100 so it can emulate a PC.

If you are considering buying a new PC compatible in the future, this gives you another reason to seek PC compatibility with your present Z-100. If your Z-100 could run PC software, you could go ahead and start using the programs you will use on the new machine. And you won't have to buy all new programs when you make the switch.

Emulating the PC Compatibles

The Z-100 is not very similar to the IBM-PC. As a matter of fact, you have to really look hard to find any similarities at all. The bus is different, the video layout isn't even close, the I/O chips aren't the same, the keyboard is different, the hard disk interfacing is incompatible. About the only things in common between the two machines are the 8088 CPU chips. Actually, it is a credit to the MS-DOS operating system that the Z-100 and IBM-PC run any of the same programs.

The secret to conquering this apparent list of incompatibilities lies in successful emulation of the PC's features. This is accomplished with more or less success by using special software, special hardware, or both.

Hardware Emulation

There are two alternatives for hardware PC compatibility with your Z-100. The GEMINI Board and the UCI Easy-PC. Both of these emulators plug into the Z-100 and provide fairly good results. I'm not going to spend any time describing how they work, because that has already been done. Reference the following articles for more information . . .

GEMINI Board	REMark January 1986, November 1986 SEXTANT #24
UCI Easy-PC	REMark June 1986, August 1986 SEXTANT #25

Even though these hardware solutions offer very good PC compatibility, there are some rough edges. For instance, neither will solve the problem of the incompatible Z-100 serial ports. Any program that communicates directly with the PC COM ports will not work using the GEMINI or UCI boards. UCI does have an optional (extra cost) Easy-I/O board available which solves this problem by providing the needed PC compatible serial ports.

Software Emulation

Emulation of the IBM-PC can also be accomplished to an amazing level without any additional hardware. The popular ZPC program available from HUG is the best and most widely used example of what can be done through software emulation. Some of you newcomers may not know that there were predecessors to ZPC (like PCEM.COM) which established the feasibility of software PC emulation on the Z-100.

ZPC doesn't provide the level of compatibility achieved with the hardware emulator boards, but it comes close enough to be a contender when you consider the difference in price. Software emulation will usually require that some changes or patches be made to the program before it will run. These are required where the PC program is trying to communicate directly with a port or other hardware that doesn't have a counterpart in the Z-100.

The ZPC software approach can be helped along by adding a little hardware. The ZHS circuit described by Pat Swayne (author of ZPC) helps by making more PC programs run with fewer (or no) patches. The board may be homemade, or is commercially available from Scottie Systems. Another help is the addition of an IBM style COM port for serial communication.

Limitations to Emulation

There is one important thing to consider about all the PC compatibility solutions presently available for the Z-100. They are all already absolete! One of the main reasons we are trying to emulate the PC compatibles in the first place is to be able to run all that flashy, state-of-the-art software. But have you considered that to effectively use a lot of PC software you need EGA or VGA graphics — neither of which is supported by ZPC or the GEMINI or UCI boards. It's only a matter of time before you won't be able to find PC software that uses the old CGA modes.

The Ultimate Emulation

We're about to wrap up our conversation about emulation here, but one important consideration has been left unsaid until now. The best way to run PC programs is to have a PC compatible computer on the desk next to your Z-100. By the time you purchase one of the hardware solutions to compatibility described above, you could have made a substantial down payment on a real PC clone. (I guess maybe I should say Zenith PC!)

Programs That Run on The Z-100 and PCs

This installment of "Z-100 Survival Kit" deals primarily with the need to run PC programs on a Z-100. We've discussed the standard solution, which is trying to turn the Z-100 into a near-PC compatible. But that's only half the story — especially if you do any programming. A lot of software is available that will run on both machines (without emulation). And if you know what to look out for, you can also write your own software that runs on the Z-100, as well as PC compatibles.

Programs That Don't Care

Any program that uses MS-DOS function calls whenever it communicates with a peripheral device (including the screen) should run equally well on a Z-100 or a PC compatible. Such a program is called a "Generic MS-DOS" program (sometimes referred to as "well-behaved", because it doesn't deal directly with the hardware). Until the recent advent of windowing environments, most assemblers and language compilers were generic MS-DOS programs. For instance, Microsoft's MASM assembler will run equally well on a Z-100 and a PC clone. So will the Microsoft 'C' compiler (but not Quick-C).

One big problem with this approach is in screen control. MS-DOS doesn't establish any standards for doing things like clearing the screen, positioning the cursor, etc. This means that a "well-behaved" program must be content with a scrolling type of text display, and no graphics. A partial solution to this problem is for a program to use the ANSI.SYS screen driver. The ANSI.SYS driver (called ANSICON .DVD on the Z-100) provides a standard set of screen and cursor control escape sequences.

Any PC program that expects to use the ANSI.SYS driver will specifically mention this fact in its documentation. If it does, there's a good chance it may run on the Z-100 (or any MS-DOS computer). To try it, do the following . . .

1. Copy the file ANSICON.DVD from your MS-DOS disk into the root directory of your boot-up disk.
2. Edit (or create) your CONFIG.SYS file by adding the following line;
   ```
   DEVICE=ANSICON.DVD
   ```
3. Be sure the CONFIG.SYS file is in the root directory of your boot-up disk. Then, reboot the computer to load the ANSI driver.
4. Now you're ready to try the program. Having the ANSI driver loaded won't make any difference unless the program uses the special ANSI escape sequences for screen and cursor control.

Want to write your own programs that use ANSI escape sequences for portability? Here are some commonly used ANSI escape sequences;

ASCII Command	Hex Values	Function
Esc[n;mH	1B 5B n 3B m 48	Move cursor to row 'n', column 'm'.
Esc[nA	1B 5B n 41	Move cursor up 'n' rows.
Esc[nB	1B 5B n 42	Move cursor down 'n' rows.
Esc[nC	1B 5B n 43	Move cursor right 'n' columns.
Esc[nD	1B 5B n 44	Move cursor left 'n' columns.
Esc[2J	1B 5B 32 4A	Clear screen, move cursor to top left
Esc[K	1B 5V 4B	Erase to end of line

The 'n' and 'm' letters indicate an ASCII decimal number. The Programmer's Utility Pack and most good MS-DOS reference texts have more information about the ANSI escape sequences.

The main problem with programs that use MS-DOS for everything is that they are noticeably sluggish when writing to the screen (with or without ANSI.SYS). The only way to avoid the slow screen output of DOS is to use BIOS routines or write directly to video memory. And if the program uses graphics at all, the programmer has no choice, because MS-DOS doesn't do windows (or graphics).

Programs That Differentiate

If a program uses true graphics (lines, circles, filled areas, etc.), or writes text directly to the video memory, then it must have access to at least two unique sets of graphics routines. One set for the Z-100, and another for the PC compatible. There are several general schemes that could be used to determine which routines should be used. Here are some examples:

1. The graphics routines for each machine could be kept in seperate object libraries or files. When the program is configured for a particular computer the appropriate set of graphics routines is linked with the main program to make the final executeable program. This method results in the fastest execution time and smallest code size, but is difficult to implement.
2. The graphics routines could be organized into a device driver or memory resident library. This method is not easy to implement with high-level languages, and suffers from slow calling times for routines in the library.
3. The program could contain all the graphics routines required for any configuration, and decide which routines to use at run time through conditional branching. A program which uses this method will have the advantage of running on either machine without reconfiguration, but will suffer somewhat in code size and speed.

Regardless of which of these methods you might choose for your own programs, they all have one requirement. At some point, they need to know the host computer type so they can choose the appropriate graphics routines. If the program will be configured ahead of time for a certain computer (as in 1 or 2 above), the obvious way to determine the host computer is to ask the user. But if the program will need to determine the host computer type each time it is run, a more convenient way would be appropriate.

I have found that a reliable way for a program to determine its host computer type is to check two bytes of system memory located at 0040:0000H and 0040:0003H. If this is a Z-100, the area beginning at segment 40H is the BIOS jump table, and bytes 0H and 3H will both be 0E9H (a jump instruction). If this is a PC compatible, the area at segment 40H is the BIOS data area, and 0E9H's will not be present. Here's how it looks in assembly language:

```
        MOV     AX, 40H                 ; get BIOS segment
        MOV     ES, AX                  ;
        MOV     SI, 0                   ;
        CMP     BYTE PTR ES:[SI], 0E9H  ; jump instruction?
        JNE     PC                      ; no, go to PC routine
        CMP     BYTE PTR ES:[SI+3], 0E9H ; check another place
        JNE     PC                      ; no, go to PC routine
Z100:   ...                             ; Z100 graphics routine
        JMP     COMMON                  ; join common code
PC:     ...                             ; PC compatible graphics rou
tine
COMMON: ...                             ; common code resumes here
```

Writing Your Own Programs
General Guidelines

If you want to write your own programs that run on the Z-100 and PC compatibles, here are some tips; Keep it as simple as possible. If your application doesn't require fancy graphics, don't worry about it. Generic DOS programs are definitely the best bet when practical. If necessary, use the ANSI device driver for more flexibility in text screen displays. Don't forget that an easy way to clear the screen using MS-DOS is to issue 25 line feeds.

Stay away from using keyboard input from special function, keypad, and arrow keys. There are no standard ASCII definitions for these keys. If you must use input from keys other than ASCII characters, symbols, and numbers, then you will have to write a keyboard input routine that recognizes the difference between the Z-100 and PC keyboards.

Use MS-DOS function calls for all disk input/output and communication to serial and parallel ports. Most high level languages use MS-DOS functions, but avoid using any language features that depend on BIOS calls or direct hardware contact.

If you program in BASIC, the easiest way to write programs for the Z-100 and PCs is with interpreted GW-BASIC. Most programs will operate correctly on either machine. But beware of the SCREEN command. You will need to take into consideration that the colors and screen resolution are different between the two types of computers.

Wrapping It Up

We've taken a good look at the question of PC compatibility in this installment of "Z-100 Survival Kit". It is an issue which is important to many Z-100 users. However, I don't feel like compatibility holds any secrets to the continued popularity of the Z-100. Running PC software on a Z-100 is just another way of saying "I wish I had a Z-248 instead".

If the Z-100 is to have a lasting user support base, it will be achieved through the strengths of the machine itself — running software designed to take full advantage of its features. In future columns, we'll look at ways to take advantage of some of those powerful features. ✱

Continued from Page 15

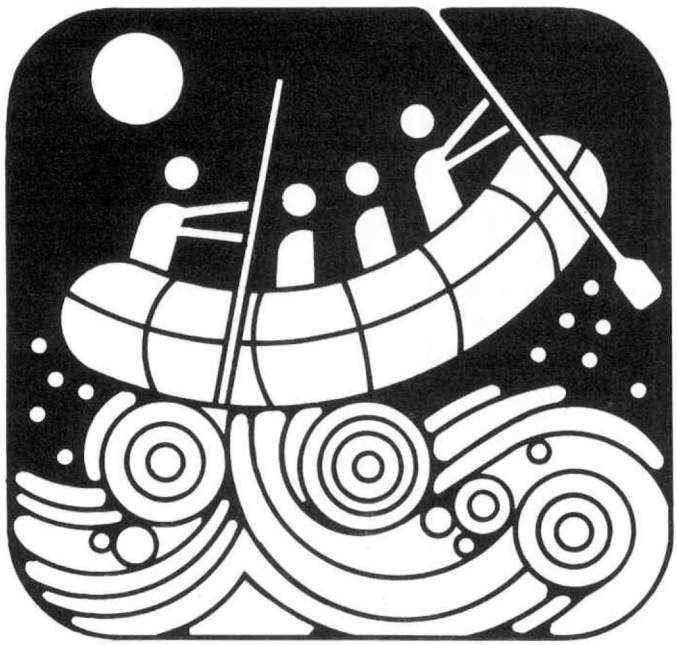
lutions like 640 by 480, we are starting to get some pretty good looking circles. The earlier machines always coughed up something which looked like a football at best.

I should note here that EGAPaint maintains its screen aspect ratio such that vertical and horizontal resolutions appear balanced on the screen and on the printer as well. Aspect ratios are selectible during setup but I am comfortable with the recommended aspect ratio of RIX Software.

EGAPaint
RIX Softworks, Inc.
18552 MacArthur Blvd.
Suite 375
Irvine, CA 92715
(714) 476-8266

Microsoft Mouse PCPaint
MSC Technologies
2600 San Tomas Expressway
Santa Clara, CA 95051
(408) 988-0211

Pallette
Software Wizardry
8 Cherokee Drive
St. Peters, MO 63376
(314) 477-7737 ✱



Z-100 Survival Kit #3

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

Writing Text to the Z-100 Screen

One requirement of almost every program is the ability to display text on the computer screen. If you write some of your own programs, there may be several different methods available to you for outputting text to the screen. We're going to look at some of those methods in this installment of Z-100 Survival Kit.

Let Me Count the Ways . . .

First off, let's take a look at all the possible ways that can be used to get text on the screen . . .

1. High-level routines
2. DOS function calls
3. BIOS calls
4. Monitor ROM calls
5. Writing directly to video memory

High-level routines can be thought of as the resident print commands available in most programming languages. For instance, the PRINT and PRINT USING commands in BASIC. Another example would be the printf() or puts() function in the 'C' language. If you have done batch file programming, the ECHO command may also be thought of as a high-level text output routine. These are commands (or function calls) which make life easy for those who are programming with a high-level language.

DOS function calls offer a quick and easy way of outputting text for the assembly language programmer. Some high-level languages (noteably BASIC and FORTRAN) make it very awkward to use DOS function calls in a program, but quite a few (like 'C' and PASCAL) include library functions specially used for DOS Functions.

BIOS calls might be used when speed is critical to an application program. Calling the BIOS for text output is noticeably faster than most built-in language print functions. But in exchange for the increase in speed, you must give up the portability (the ability to use the program on more than one type of computer) that normally comes with using high-level routines or DOS function calls.

Calls to the MTR-100 monitor ROM will give much the same results as BIOS calls, but one more layer of overhead will be stripped away.

Writing directly to the video memory is the most versatile way to write text to the screen. It is also the most complicated and least portable way, and should therefore be reserved for those applications where custom fonts or fast display are important. Unlike the PC compatibles, the Z-100 has no dedicated text mode where ASCII text can be written directly to video memory. Thus, in order to display text on the Z-100 screen, you must actually transfer the font design to the screen one pixel or byte at a time.

Now that we have the preliminary descriptions out of the way, let's take a more detailed look at each of these alternatives.

High-Level Routines

Most programmers (or those who dabble in programming) will never use, or need to use, anything other than the built-in capabilities of their language interpreter or compiler for text display. As long as the PRINT statement will do what you need to do, there is probably no need to add extra complication to your life. I'm not going to give any examples of using these high-level routines because that should be documented thoroughly in your programming language manual.

What I am going to talk about, with reference to high-level text routines, is under what circumstances you might need to use an alternative text outputting scheme. The obvious case, of course, is where more speed is required. You can greatly speed up text display by using BIOS calls, or writing directly to video memory. Using DOS function calls probably won't do much to help text display speed, since the high-level routines usually make use of DOS calls anyway.

Another case where the high-level routines won't do the job is when you need text to be displayed in positions other than the normal rows and columns on the screen. Most pixel graphics and CAD programs, for example, will allow you to place text anywhere on the screen, on a pixel-by-pixel basis. In order to do this, they must write directly to the video memory. If you want this same type of capability in your own program, you also will have to write directly to video memory.

Along the same lines, if you want to use custom text, which is not a standard size (i.e., 8 × 9 pixels), you must use direct video memory access. Text charac-

ters which are smaller or larger than the normal font, as well as italic text, are examples of the possibilities.

Another situation which may require use of a DOS or BIOS routine is when you don't want your program to be interrupted when a user types Control-C at the keyboard. Many high-level languages do not allow you to disable this feature during screen output, or keyboard input.

One last note about high-level text output routines. Be sure to get familiar with whatever resources are offered with your particular language interpreter or compiler. It is possible (but not likely) that the text output routine already uses the BIOS for output, thus making it unnecessary to look further for a speed upgrade. Most languages offer more than one alternative for displaying text on the screen — check them out. If you are working with a compiled language, your compiled program will usually be much smaller if you stay away from the more versatile print functions (like printf() in the 'C' language, or PRINT USING in BASIC). For instance, in the 'C' language, using the printf() function may cause the compiler to drag in all of the floating point functions, increasing the program size by several thousand bytes.

DOS Function Calls

The MS-DOS operating system, in addition to taking care of disk I/O responsibilities, has an entire library of useful functions which may be called from a user program. These functions aren't described in the Users' Manual that accompanies MS-DOS. To find out about them, you have to either buy the Programmer's Utility Pack (PUP), or pick up a book about MS-DOS at the bookstore. If you're a programmer type, you really should invest in the PUP while they are still available from Heath/Zenith. You'll wonder how you ever lived without it! Or if you're just curious, most bookstores these days have dozens of books describing the inside workings of DOS. Don't worry about finding one that describes the Z-100 (You'll be looking for a long time). The MS-DOS function calls for any machine that runs DOS are the same.

There are three DOS function calls which deal specifically with output to the computer screen. They are numbers 2, 6, and 9. Here is a brief description of each...

Function 2
Display Character

This function is used to display a single text character on the screen at the current cursor position. Load register DL with the character, and call interrupt 21H with register AH set to 2. Like this . . .

```
MOV DL, 'P'   ; prepare to display a 'P'
MOV AH, 2     ;   with function call 2
INT 21H       ; output to screen
```

This is the most bare-bones of all the DOS screen output functions. It will break to the Control-C routine if Control-C is typed at the console during the time the function has control.

Function 6
Direct Console I/O

This function is similar to number 2, except that it also allows you to input characters from the keyboard (which we aren't concerned with here). Load register DL with any printable character, load AH with 6, and call interrupt 21H.

```
MOV DL, 'P'  ; we'll display a 'P' again
MOV AH, 6    ;   this time using function call 6
INT 21H      ; output to screen
```

One important thing to note is that DOS function 6 does not check for Control-C at the keyboard. If it is important that your program not be interrupted, this is the screen output function to use.

Function 9
Display String

This function allows you to output an entire string of characters to the screen at one time. To use it, load register pair DS:DX with the segment and offset of the start of the text string. Load AH with 9, and call interrupt 21H. Note that the text string must end with a dollar sign ('$') character, which is considered the string terminator. Example . . .

```
        MOV DX, OFFSET TEXT         ; point DX to text
        MOV AH, 9                   ;   use function call 9
        INT 21H                     ; output to screen

        ...                         ; other instructions

TEXT    DB 'Hello there!$'          ; the text string
```

This example assumes a couple of things. First, that the register DS has previously been set to the segment containing the label TEXT. It also assumes (as indicated by the ...) that some instructions (like a RET, or the end of the program) occur between the function call and the TEXT label, since obviously, the DB instruction cannot be interpreted as code instructions.

You might ask, "what if I want a dollar sign to be printed?". Answer . . . you'll have to use function call 2 or 6.

How to Use DOS Function Calls in Your Programs

The next question which seems obvious is how to use these function calls in your programs. Obviously, the functions are easy to handle with assembly language, as the examples above show. Using the calls with other languages may not be as easy. Some languages may have provisions for calling DOS functions (like most 'C' and Pascal compilers), while others may make their use very difficult (like BASIC). Even among different implementations of the same language (like Microsoft 'C' and Turbo 'C') the methods for using DOS functions may differ.

Using Microsoft 'C' as an example, here is how we would use DOS function number 9 to print a string . . .

```
char text[14] = "Hello there!$";

main() {
    bdos(9, text, 0);
    }
```

The bdos() function will load register DX with a pointer to the start of the string text, load register AL with a zero (this is a dummy argument, since we don't need register AL), and then execute DOS function number 9.

Most 'C' and Pascal compilers come with a function similar to the Microsoft bdos() function, but the syntax will vary.

Most of you who program in 'C' will be asking "wouldn't it be a lot easier to use the puts() function?, like this . . .

```
main() {
    puts("Hello there!");
    }
```

Well, yes. That's the way I would normally output a string to the screen. It's hard to imagine writing an entire 'C' program using DOS function 9 for screen output. But if your program is short and simple (you don't get much shorter or simpler than "Hello there!"), you might be interested to note the following statistics; our example above, using the bdos() function, compiles into an .EXE program which is 2357 bytes long. The example using the puts() function compiles into a program of 5245 bytes. And if you wanted to take it another step, and use the printf() function, the result is 7233 bytes.

Okay, now I notice that all the assembly language buffs are laughing, because writing this simple example in assembly language . . .

```
CODE    SEGMENT
        ASSUME CS:CODE, DS:CODE
        ORG 100H
        ; start at 100H for .COM program
START:  MOV DX, OFFSET TEXT
        MOV AH, 9
        INT 21H
        INT 20H
        ; this function ends the program
TEXT    DB 'Hello there!$'
CODE    ENDS
        END START
```

. . . will result in an executable program that takes only 22 bytes!

Now that I have shown some examples of 'C' and assembly routines to make use of DOS function calls, I'll acknowledge that most of you probably use BASIC as your primary computer language. BASIC is one of those rather stiff languages (oh, I see the letters coming, now!) that doesn't have a great deal of flexibility for communicating with DOS or the hardware directly. But where there is a will,

there is a way. Witness the following . . .

Use DEBUG's mini-assembler to enter the following program fragment (don't include the comments). You can do this by running DEBUG, and then giving the 'A' command.

```
MOV BP, SP              ; get the stack frame
MOV SI, [BP+4]          ; get address of string descriptor
MOV DX, [SI+1]          ; get pointer to start of text string
MOV AH, 9               ; use function 9
INT 21H                 ; display the string
RETF 2                  ; return to calling program
```

Now if you unassemble what you have entered (using the 'U' DEBUG command), you can see the hexadecimal numbers that make up this program. They turn out to be . . .

```
89, E5, 8B, 76, 04, 8B, 54, 01, B4, 09,
   CD, 21, CA, 02, 00
```

This sequence of bytes actually represents the small program subroutine given in our example above. You can call this assembly language subroutine from a BASIC program like this . . .

```
50 TEXT$="Hello there!$"
90 FOR I=1 TO 8:READ PROG%(I):NEXT
95 DATA &HE589,&H768B,&H8B04,&H0154
96 DATA &H09B4,&H21CD,&H02CA,&H0000
100 DEF SEG
110 DOS9F=VARPTR(PROG%(0))
120 CALL DOS9F(TEXT$)
130 PRINT
```

Note that the hex integers used in the DATA statements to make the program have each byte pair reversed. This is because BASIC stores an integer with the least significant byte first in memory, then the most significant byte. Also, if your assembly language routine doesn't use an even number of bytes, don't worry — just stick a zero on the end.

If this kind of thing interests you, you might want to refer to the Appendices of your BASIC manual for a more thorough description of how to use the CALL statement. Microsoft suggests writing the original program with an assembler, and then loading it with the BLOAD command, but for short assembly routines, I like my way better.

For the most part, it is ridiculous to go through all of this trouble to display a string on the screen using a DOS function call from a high-level language. But there are times when the knowledge of how to do so may come in handy.

BIOS and Monitor ROM Calls

Up until this point, all of the techniques we have discussed for placing text on the screen have been useable for any MS-DOS based computer. So if you have a PC clone taking up space on the desk next to your Z-100, what I have said would apply to it, too. But from here on out, we're talking Z-100 only. Making BIOS calls, monitor ROM calls, and writing directly to video memory, are very machine specific actions. Any use of these methods on a PC compatible is definitely guaranteed to cause an instant crash.

The Z-100 BIOS (Basic Input/Output System) contains a user interface which may be used to communicate with the disk drives, console, printer, or auxiliary device. Using the BIOS interface is something that is fairly simple once you get the hang of it, but a proper tutorial is not really possible in the space we have left here. We'll do justice to using the BIOS interface in a future installment of Z-100 Survival Kit, but for now I'll just give you enough information to output characters to the screen (since that's the main subject we're trying to cover).

The BIOS function we're interested in is affectionately know as BIOS__CONFUNC. That's what they call it in the Programmer's Utility Pack, at any rate. You call the BIOS functions by making a long call to a jump table located at segment 40H. The address of the BIOS__CONFUNC vector is 40:0051 (hexadecimal). You can use the BIOS__CONFUNC routine to read or write a character to the console, check console status, etc. To write a character to the console (another word for the computer screen), you simply load register AL with the character, load AH with a zero, and call 40:0051. Like this . . .

```
BIOS    SEGMENT AT 40H
        ORG 51H
CONFUNC LABEL  FAR
BIOS    ENDS

CODE    SEGMENT
        MOV AL, 'P'             ; display a 'P'
        MOV AH, 0               ; code for CHR_WRITE function
        CALL CONFUNC            ; do it to it
        ...                     ; rest of program
CODE    ENDS
```

The purpose of the first portion of this code is to tell the assembler where the BIOS__CONFUNC vector is located. If you will be using DEBUG to assemble the program, you can use the 40:0051 address in the call statement . . .

```
        MOV AL, 50              ; this is a
 'P' to DEBUG
        MOV AH, 0               ;
        CALL 40:0051            ; call BIOS
_CONFUNC
```

The BIOS__CONFUNC character output routine ends up calling the MTR-100 monitor ROM to get its work done. It uses an entry point into the MTR-100 named MTR__SCRT which is at address FE01:0019. If you prefer, your program can call the monitor ROM directly by using this code . . .

```
MTR     SEGMENT AT FE01H
        ORG 19H
SCRT    LABEL FAR
MTR     ENDS

CODE    SEGMENT
        MOV AL, 'P'             ; display a 'P'
        CALL SCRT
        ;   using the MTR-100 ROM routine
        ...                     ; rest of program
CODE    ENDS
```

The advantage of going straight to the MTR-100 routine is to tweak the last bit of speed out of the system. To give you an idea of the speed difference, I tried displaying 50,000 characters using a looping assembly language routine. The results . . . using the BIOS__CONFUNC routine took 23 seconds. And using the MTR-100 SCRT routine, the time was 17 seconds. Just for laughs, I also tried using MS-DOS function call 2 — it took 58 seconds.

In defense of the BIOS interface, I must say that for most programs it is more convenient to use the BIOS because of the other console functions available. Another advantage to using the BIOS interface was to avoid problems if a new

version of the MTR-100 ROM was released with different entry points. But since the Z-100 is obsolete machinery, I doubt if Zenith will be providing any more MTR-100 versions, so that is no longer a factor — all of the MTR-100 versions I know of have the MTR_SCRT entry at FE01:0019.

Another consideration when deciding whether to use the BIOS interface or the MTR-100 ROM routine, is that the BIOS causes a user interrupt to be generated whenever a character is output to the screen. Memory resident utilities that depend on this interrupt to gain control will not work correctly if you bypass the BIOS by going straight to the MTR-100.

Using BIOS Output with BASIC

Here is a routine that you might actually find useful in your BASIC programs. Enter the following program using DEBUG's mini assembler . . .

```
100:    MOV BP, SP              ; get stack frame
        MOV SI, [BP+4]          ; find start of string descriptor
        MOV CL, [SI]            ; get length of text string
        CMP CL, 0               ; if null string,
        JZ 11F                  ; return immediately
        MOV CH, 0               ; CX is now character count
        MOV SI, [SI+1]          ; SI points to start of string
111:    MOV AL, [SI]            ; get a character
        MOV AH, 0               ; code for CHR_WRITE function
        PUSH SI                 ; save string pointer
        CALL 0040:0051          ; call BIOS_CONFUNC
        POP SI                  ;
        INC SI                  ; point to next character
        LOOP 111               ; loop until done
11D:    RETF 2                  ; return to BASIC
```

Now if you unassemble this program with DEBUG, you will get the following hex bytes . . .

```
89 E5 8B 76 04 8A 0C 80 F9 00 74 13 B5
   00 8B 74 01
8A 04 B4 00 56 9A 51 00 40 00 5E 46 E2
   F2 CA 02 00
```

Here's an example of how to use this assembly routine in a BASIC program . . .

```
40 DIM PROG%(17)
50 TEXT$="Hello there!"+CHR$(10)+CHR$(13)
90 FOR I=1 TO 17:READ PROG%(I):NEXT
95 DATA &HE589,&H768B,&H8A04,&H800C
96 DATA &H00F9,&H1374,&H00B5,&H748B
97 DATA &H8A01,&HB404,&H5600,&H519A
98 DATA &H4000,&H5E00,&HE246,&HCAF2,&H0002
100 DEF SEG
110 BIOS=VARPTR(PROG%(0))
115 FOR I=1 TO 1000
120 CALL BIOS(TEXT$)
125 NEXT
```

You'll notice that this assembly routine lets you print any BASIC string using the Z-100 BIOS interface. This will be about twice as fast as using the BASIC PRINT statement. The program sample above will print "Hello there!" on the screen 100 times.

Handling Special Characters

All of the methods of displaying text that we have discussed up to this point will handle non-printable ASCII characters in a special way. For instance, they will all recognize a carriage return or line feed code. They should also recognize special escape sequences that may be used to clear the screen, position the cursor, etc.

If you are using a high-level language, the character output routine you use may filter out some of these special characters. Therefore, you may need to experiment with the results (or consult the manual) to determine exactly how a particular high-level routine affects the character stream. For example, some routines may automatically insert a carriage return character whenever a line feed is sent to the screen. And some routines may filter out escape characters, making it impossible to send Z-100 escape sequences to the console.

To Be Continued . . .

In the next installment of Z-100 Survival Kit, I'll wrap up our discussion about displaying text on the screen, by describing how to write directly to video memory. I'll be including code samples for a character output routine, and we'll also talk about how to use large or fancy text in your programs. ✱

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

Z-100 SURVIVAL KIT #4

Writing Text Directly to the Z-100 Video Memory

The last issue of Z-100 Survival Kit was all about different ways to write text on the screen of the Z-100. We discussed how to display text with high-level routines, DOS function calls, BIOS calls, and MTR-100 monitor ROM calls. The last way of doing it is to write the text directly to video memory, and that is the subject of this installment of Survival Kit.

Understanding the Z-100 Video Memory Map

Before we start talking about writing text to video memory, it might be wise to take a quick look at the Z-100 video memory layout. The Z-100 is an all-graphics machine (i.e., there is no text mode) which has three seperate planes of video memory — one for each primary color (red, green, blue). Each plane is allocated a 64K chunk of the system RAM memory map, as follows . . .

```
C0000 to CFFFF  Blue video RAM bank
D0000 to DFFFF  Red video RAM bank
E0000 to EFFFF  Green video RAM bank
```

The numbers given above are five digit hexadecimal values representing the offset into the system memory map. The memory starting at C0000 (blue bank) is commonly described as segment C000, based on the way this memory must be accessed using the 8088 CPU segment registers.

Each text character position on the screen is composed of 9 bytes of viewable data, which are the nine bytes of the font design. For instance, here is a representation of the capital 'A' font character...

```
0 0 0 0 0 0 0 0  00H
0 0 0 1 1 1 0 0  1CH
0 0 1 0 0 0 1 0  22H
0 0 1 0 0 0 1 0  22H
0 0 1 1 1 1 1 0  3EH
0 0 1 0 0 0 1 0  22H
0 0 1 0 0 0 1 0  22H
0 0 1 0 0 0 1 0  22H
0 0 0 0 0 0 0 0  00H
```

The Z-100 screen is organized into 25 text lines each containing 9 scan rows. The beginning of each scan row is offset 128 (80H) bytes from the start of the previous scan row. Since there are only 80 displayable columns on the screen, this scheme leaves an invisible area of 48 bytes at the end of each scan row.

The start of each text line is offset 2048 (800H) bytes from the start of the previous one. This is enough memory for 16 scan lines (16 × 128 = 2048), but since only nine scan lines are used for each text line, this leaves 1152 (480H) bytes unused at the end of each text line.

It may seem strange that the video layout leaves all these unused 'holes' in the memory map. But this was done to facilitate address calculations. For instance, if the scan rows were contiguous, the calculation to find the start of a scan row would involve multiplying by 80. But since each scan row is actually offset by 128, the calculation can be done by bit shifting, which is much faster. The same type of argument also applies for the start of each text line — shifts are faster than multiplying by nine.

Calculating the Video Offset for A Text Character

Here is a formula which can be used to calculate the video RAM base offset for a particular text character . . .

```
((Line - 1) x 2048) + (Column - 1)
```

This formula assumes that the top line is line one, and the left-most column is column one. The first byte of the font character goes at this base address, and each successive byte (nine in all) is offset 128 from the last. For example, if you wanted to put the letter 'A' (see font example above) at the 21st column position of the 8th text line . . .

```
base address = ((8 - 1) x 2048) +
   (21 - 1) = (7 x 2048) + 20 = 14356
```

Therefore, the nine bytes of the font character would be placed at . . .

```
00H --->    offset 14356 (3814H)
1CH --->           14484 (3894H)
22H --->           14612 (3914H)
22H --->           14740 (3994H)
3EH --->           14868 (3A14H)
22H --->           14996 (3A94H)
22H --->           15124 (3B14H)
22H --->           15252 (3B94H)
00H --->           15380 (3C14H)
```

Colored Text

You might have noticed that we haven't been talking about which segment (video RAM bank) to use. This is because the address calculations are the same, regardless of which color bank you write to. Typically, all three banks would be written whenever you are putting text on the screen. Different color text is generated by writing the font pattern or clearing the character position of each bank. For instance, if you want white text, you would write the font pattern to every bank. Or if you wanted green text, you would write the font pattern to the green bank (E000) and clear the other two banks. The Z-100's video RAM port allows an easier way to write more than one bank of memory at a time, but our space here doesn't allow a complete discussion of that feature right now. We'll cover how to program the Video RAM port in another installment of the Z-100 Survival Kit.

Assembly Language Character Output Routine

Now that we have some of the preliminaries out of the way, let's get on with the show. Listing 1 is an assembly language routine that can be called to display a text character on the screen.

The character to display is passed to the routine on the stack, a method which is used by quite a few high language compilers. This routine was written specifically to interface with Microsoft 'C', but it should be useful with other compilers, and from other languages, as well. The character position on the screen, and the

text colors, are passed using public variables.

Here is an example of how to use the _prtc routine from an assembly language program . . .

in your data segment . . .

```
fore_color      db      ?
back_color      db      ?
text_line       dw      ?
text_column     dw      ?
```

code to call PRTC . . .

```
        mov     fore_color, 7           ; make foreground white
        mov     back_color, 0           ; make background black
        mov     text_line, 10           ; position will be line 10, column 1
        mov     text_column, 1          ;
        mov     al, 'A'                 ; will display letter 'A'
        push    ax                      ; push argument onto stack
        call    _prtc                   ; display the character
        add     sp, 2                   ; adjust stack
```

The Font Table

There is one thing you'll need to note before trying the _prtc routine in your own program. The start of the font table (represented by the variable FONT in our listing) is not defined. You will need to define your own font table of 95 printable ASCII characters (nine bytes per font character). The start of this table in your data segment should be indicated with the label "FONT".

An alternative to creating your own font table is to use the MTR-100 table that already exists in memory. The segment: offset address of this table is at offset 06FH in the MTR-100 data segment. Here's how you find it . . .

```
        mov ax, 0               ;
        mov ds, ax              ; get interrupt page segment
        mov si, 0FEH            ;
        mov ds [si]             ; get MTR-100 data segment (stored at 0:0FEH)
        mov si, 06FH            ;
        mov bx, [si+2]          ; get segment:offset of table
        mov ds, [si]            ; DS:BX is now table start address
```

If you elect to use the MTR-100 font table, you'll need to modify the _prtc routine so that SI is initialized to point to the table, and you'll need to set DS to the table segment.

Wrap-Up

The _prtc routine described here isn't quite as fast as simply calling the MTR-100 monitor ROM routine. The reason is because we haven't used any tricks to do multiple bank accessing, and we have kept the code as short and understandable as possible. More experienced programmers who are still interested may want to study the listing of the DFC (Display Font Character) routine in the MTR-100 ROM listings (available as a part of the Z-100 technical manual set). This routine demonstrates how to use the multiple access capability of the video RAM port to crank the last bit of speed from the Z-100 video.

The question might arise . . . "If the MTR-100 routines are faster (or just as fast) as my own routine, why bother?". The answer lies in the flexibility of having your own code to write characters to the screen. The routine given in Listing 1 is just a starting point for other variations you might want to develop for displaying odd size fonts, or writing text in non-standard graphics modes. If your program can live with the standard 25 line by 80 column display, your right — it probably would make a lot more sense to stick with DOS functions or BIOS calls for screen output.

Customized Fonts

For the ambitious programmer, the possibility exists to program the Z-100 to display any type, or shape, of font. I'm not talking about simply changing the design of the standard font here. I mean using fonts with different sizes or characteristics than the standard text font. For example, you might develop a font with a 12 × 16 matrix size (instead of the standard 8 × 9). Or you might want to display italic characters on the screen. Most graphics programs give you the ability to have these types of fancy fonts.

Generally speaking, any font that doesn't fall neatly into byte boundaries will have to be displayed on the screen one pixel at a time. In other words, your program will have to scan the font character matrix from left to write, top to bottom, and set the pixels one at a time. This will be slow, but can reasonably be done in assembly language.

Fan Mail

In the first installment of Survival Kit, I asked you to write and tell me about your interests. And I asked for your help in determining a direction for this column. Well, the mail has started to flow from that first column in January 1989. The response has been very enthusiastic (and very voluminous). Right at this moment, I'm not too sure how I'm going to keep up with it all. But rest assured that I am formulating a plan, even at this very instant. Part of the plan is that I hope this mail subsides to reasonable levels soon!

About one fourth of the mail I have been getting is simply expressing the writer's gratification at having a Z-100 specific column to read. I appreciate this kind of mail, but you really should be sending it to Jim Buszkiewicz. If you think this column is great, don't tell me (I already know!), tell the editor.

Q & A

The balance of the mail (about three pieces per day at present) is composed of letters asking questions about the Z-100. Most of them are very technical questions. Some of them I don't know the answers to. But I got myself into this thing . . . I guess I can make it through.

As stated in installment #1 of Survival Kit, I'll try to get you a personal reply if you write. I think it's too much to ask people to wait several months to see the answer to a question in the magazine. If things just get too crazy, you'll at least get a note acknowledging that I received your letter. As for including a SASE with your letter — I'm not real picky, but that would be nice. I'd much rather have a self-addressed, stamped fifty dollar bill (just kidding — don't send money!).

At any rate, it would appear that the first directional push my readers are giving me is for a question and answer section in this column. I'll try to be accommodating. Here goes . . .

Q. Running under Z-DOS, I was able to patch the BIOS to speed up the step rate of my floppy drives. Do you know the patches to do this with MS-DOS versions 2 or 3?

A. It's much easier with DOS 2.x or up. You configure the step rates of all your floppy drives by using the CONFIGUR .COM program (option D on the main menu) supplied with MS-DOS. Keep in mind that the maximum step rates are typically 6 milliseconds for 5-1/4 inch drives, and 3 milliseconds for 8 inch drives. Individual drives may not be quite this fast.

Q. I'm looking for a good memory diagnostic program. I know I have bad memory, because one of my larger programs consistently causes a Parity/Buss error. But my present RAM diagnostic doesn't find the problem.

A. The best memory diagnostic I've seen is the one included with the Z-100 disk-based diagnostics, which I assume are available through Heath/Zenith. I got my copy of the diagnostics because my company used to be a Zenith service center. The disk-based diagnostics contain a RAM test (for system and video RAM) which will not only do a thorough check, but will also tell you which chip is causing the problem.

Q. I would like to install a 60 megabyte hard disk in my Z-100, but it is my understanding that only later versions of MS-DOS (not available on the Z-100) allow one to break the 32 megabyte barrier. How can this be done?

A. MS-DOS doesn't really care about the megabyte capacity of a hard disk. The real limitation is the number of sectors (or

Listing 1

```
;       _prtc - Display a character on the screen
;
;       void prtc(char c);
;
;       Entry:
;               c = character to display
;               fore_color, back_color should be set to desired colors.
;               text_line, text_column should be set to desired position.
;                no range checking is done on text_line and text_column.
;       Returns:
;               nothing
;
;        Action:
;               Displays character on screen.
;               text_column is incremented by one.  Line feeds and wrap are
;                 not handled.

                extrn fore_color:byte, back_color:byte
                extrn text_line:word, text_column:word

                public _prtc
_prtc           proc    near

        push    bp                      ;
        mov     bp, sp                  ;
        mov     al, [bp+4]              ; get character to display
        cmp     al, 32                  ; test for legal ASCII value
        jge     PC2                     ;
PC1:    pop     bp                      ; if not legal, simply return
        ret                             ;
PC2:    cmp     al, 127                 ;
        jnc     PC1                     ;
        push    di                      ;
        push    si                      ;
        mov     bx, _text_line          ; get text line for character
        dec     bx                      ;
        xchg    bl, bh                  ; this is the same as multiplying
        xor     bl, bl                  ; BL by 256
        shl     bh, 1                   ;
        shl     bh, 1                   ;
        shl     bh, 1                   ; AX has now been multiplied by 2048
        mov     di, bx                  ;
        add     di, _text_column        ; DI now holds video RAM offset
        dec     di                      ;
        mov     ah, 0                   ; calculate font offset
        sub     ax, 32                  ; by multiplying font index by 9
        mov     si, ax                  ;
        shl     ax, 1                   ;
        shl     ax, 1                   ;
        shl     ax, 1                   ;
        add     si, ax                  ;
        add     si, offset FONT         ; SI is font offset
        mov     dx, 0E000H              ; get video RAM segment
        mov     ah, 4                   ;
PC4:    push    ax                      ;
        push    si                      ;
        push    di                      ;
        mov     es, dx                  ;
        xor     bx, bx                  ;
        test    byte ptr _fore_color, ah; make color masks
        jz      PC7                     ;
        mov     bl, 0FFH                ; BL is foreground color mask
PC7:    test    byte ptr _back_color, ah;
        jz      PC8                     ;
        mov     bh, 0FFH                ; BH is background color mask
PC8:    mov     cx, 9                   ; will write 9 bytes
PC9:    lodsb                           ; get byte from font table
        mov     ah, al                  ;
        and     ah, bl                  ; AND with foreground mask
        not     al                      ; reverse font image
        and     al, bh                  ; AND with background mask
        or      al, ah                  ; Or them together
        mov     es:[di], al             ; write byte to video RAM
        add     di, 80H                 ;
        loop    PC9                     ; next font byte
        pop     di                      ;
        pop     si                      ;
        pop     ax                      ;
        shr     ah, 1                   ;
        sub     dx, 1000H               ;
        cmp     dx, 0C000H              ;
        jge     PC4                     ;
        inc     byte ptr _text_column   ;
        pop     si                      ;
        pop     di                      ;
        pop     bp                      ;
        ret                             ;
_prtc           endp
```

clusters) that can be accommodated. Heath/Zenith gave Z-100 owners a way around this limitation (not previously available to PC clone owners) by allowing the hard disk to be PREP'ed using 1024 byte sectors, instead of the standard 512 byte size. To use a hard disk larger than 32 megabytes, simply run the PREP program using the /K switch. This will tell PREP to use 1024 byte sectors. One disadvantage to using 1024 byte sectors is that the minimum amount of space consumed by any file will be increased to 2048 bytes (assuming a smallest cluster size of two sectors). If you have a 40 meg hard disk that formats out to say 35 megabytes, you might be better off just using 512 byte sectors, and living with a 32 meg capacity disk. Especially if most of your files are small. *

#5

Z-100

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

SURVIVAL KIT

The Technical Information Gap

Quite a bit of the mail coming in as a result of this column is from Z-100 users who crave in-depth technical information. I can appreciate that, because I, too, am interested in knowing everything I can about the '100. And in recent times, there doesn't seem to be many places you can turn to for such information . . . unless you're lucky enough to have an active users' group in your neighborhood.

As far as orphan computers go, the Z-100 is pretty well documented. Zenith published a decent technical manual for the machine, and the service literature and spec sheets describe its operation with a fair amount of detail. Of course, if you're not a Zenith Service Center, you don't have access to all the service literature, but most of the info you need to program the machine is in the Z-100 Technical Manual (a two volume set).

Although most of the information about the Z-100 hardware is available, portions of it can't be easily understood by the novice. One of my goals in writing this column is to try to bridge the gap between raw technical information and practical applications. After all . . . having the Motorola spec sheet for the MC2661 Enhanced Programmable Communications Interface in front of you is one thing . . . but knowing what to do with it is another.

I'd enjoy writing technical stuff every column, but I'm afraid the average person would quickly become bored with that. So you techies are going to have to be patient — we'll bite off a chunk at a time. You're in luck this month though, because I'm going to talk about programming the Z-100 keyboard chip. If you're just a casual programmer, don't be scared away. I'll try to keep it light, and throw in a few code examples in BASIC.

How the Keyboard Hardware Works

The Z-100 keyboard circuitry is a model of simplicity, or complexity, depending on how you look at it. In addition to the array of key switches, one might expect to find a bundle of individual gates or encoder chips to convert the key switch positions to key codes. But instead, the keyboard circuit is composed mainly of only four chips. The secret to this low chip count is that the main keyboard encoder chip is a full-fledged 8-bit microprocessor, complete with two I/O ports, 1000 bytes of ROM, 64 bytes of RAM, internal clock and timer. The Intel 8741A Universal Interface Microcomputer used as a keyboard encoder is really a little computer that runs independently inside your Z-100. It has its own ROM program which handles key code assignments, autorepeat, and key buffering. There are several other chips associated with the keyboard circuit, but they are used only to generate the key click, bell, and reset signals.

Fanatical hackers might also want to note that the program in the 8741A chip is stored on a UV eraseable EPROM, so you could actually modify the keyboard encoder programming if you want — the programming instruction set is given in the Intel spec sheet for the 8741A. I can't even imagine why you would want to reprogram the keyboard encoder, but given the possibility, someone will think of a reason.

After seeing that the keyboard encoder is really a computer in itself, it's easy to explain how the keyboard works. The 8741A scans the keyboard switch array by reading its two I/O ports (one port scans the rows, the other scans the columns). When a switch closure occurs (or a switch opening in up/down mode) the key switch is identified, and an appropriate key code is placed on the data bus. The RAM in the encoder is used to provide a 17 key first-in, first-out buffer in case the processor can't service the keyboard right away.

Programmable Features of The Keyboard Encoder

The keyboard encoder can be programmed to operate in one of two different modes. The default (power up) mode is called the ASCII scan mode. In ASCII mode, the keyboard matrix is scanned until a key is pressed. After the key switch is read, scanning resumes to see if another key will be pressed. Two keys, at most, are read at a time and decoded (thus allowing key modifiers such as SHIFT or CTRL to be used). When one of the keys are released, scanning resumes again. The key codes read at the data port during ASCII scan mode are the ASCII codes you are familiar with. Special codes are generated for keys that don't have an ASCII equivalent (like the function keys).

The other mode of keyboard operation is called UP/DOWN (or event driven) scan mode. In this mode, the encoder scans continuously, regardless of what, or how many keys are down. In addition to generating a keycode when the key is pressed, a unique keycode is also generated when the key is released. The key codes read from the data port in UP/DOWN mode are arbitrary codes, and not

the standard ASCII key codes. Each key has a unique up and down code.

The ability to switch between normal ASCII mode, and UP/DOWN mode, is one of the nice things about having a microprocessor for a keyboard encoder. Instead of requiring a totally different keyboard circuit, all that is necessary is for the internal encoder program to use a different scanning algorithm.

In addition to the two scan mode choices described above, you may also select how keystrokes will be detected by your program. The way it is normally done (by the BIOS) is by using a hardware interrupt generated by the keyboard encoder whenever a key code is available at the data port. This interrupt is serviced by an interrupt routine which reads the keycode, and places it in the type-ahead-buffer. Alternatively, you may tell the keyboard encoder not to generate interrupts, in which case you will need to continuously poll the keyboard status to see when a key is struck. More on this later.

Another feature that can be enabled or disabled by software is the key autorepeat. When enabled (the default), this feature will cause the key to automatically begin repeating after it is held down for a short time. The repeat rate (11 keys per second) cannot be changed by software, although the FAST REPEAT key can be used to jump it to 28 keys per second. The FAST REPEAT key will cause the key to repeat **even if autorepeat is disabled.**

We all know that the Z-100 keyboard makes a little click sound when the keys are pressed (and released, in UP/DOWN mode). This feature can also be turned off with software. The key click is generated by the same circuit and oscillator as the bell sound. Both are 1 kHz tones, but the short 10 ms duration of the key click makes it sound like a 'click'.

Lastly, your program can disable the keyboard encoder completely, so that all keystrokes will be ignored. After disabling the keyboard, the only way to get it back again is to send an 'enable keyboard' command, or do a master reset (CTRL RESET).

For further information about the operation of the keyboard encoder, you might want to refer to the Z-100 Technical Manual. A fairly detailed description is given on theory of operation and programming of the encoder.

Let the Programming Begin!

'Nuff talk. I think we're ready to do something with the keyboard encoder. There are three ports that are used to communicate with the encoder chip;

```
Data Port          0F4H   ...   24
4 decimal
Command Port       0F5H   ...   24
5 decimal
Status Port        0F5H   ...   24
5 decimal
```

BASIC Program to Experiment with Keyboard Encoder Commands

```
100 REPEAT=0 : GOTO 140
110 PRINT"Hit 'R' to toggle AutoRepeat feature..."
130 IF INPUT$(1)<>"R" THEN 110
140 IF(INP(&HF5) AND 2)=2 GOTO 140
150 IF REPEAT THEN REPEAT=0 : OUT &HF5,2 ELSE REPEAT=1 : OUT &HF5,1
160 GOTO 110
```

Listing 1

The data port (at 0F4H) is a read-only port. (What would you expect a keyboard to do with data you wrote to it, anyway?) The command port (at 0F5H) is used to tell the keyboard encoder chip what you want it to do. And the status port (also at 0F5H) is used to see if the keyboard processor is ready, or if there is a keycode available on the data bus. Obviously, the command port is a write-only port, and the status port is a read-only port. (Otherwise, how could they be at the same address?)

The most obvious thing we might want a program to do with the keyboard encoder is read keystrokes, but let's leave that for a minute. It's easier to demonstrate how to write commands to the keyboard encoder by writing a program that does some other things. How about a sample program that enables or disables the autorepeat feature. (We could also do a simple program that turns the key click on or off, but I think that subject has been run thoroughly into the ground in every Heath/Zenith publication I have ever read.)

Consider the simple BASIC program shown in Listing 1. Line 100 assures that the keyboard encoder starts out in autorepeat mode. Line 130 waits for you to hit a key. If the key is a capital 'R', the autorepeat mode is toggled on or off, otherwise, the prompt message is re-displayed. Line 140 is a loop that checks to make sure the keyboard processor is ready to accept a command. Actually, at the speed which GW-BASIC executes commands, there isn't any way the program could keep up with the keyboard processor, so this line could be eliminated from the listing without any ill effects. But it is included here to demonstrate good programming practice. If you program in assembly language or a compiled language, you're looking for trouble if you don't poll the keyboard processor status before issuing commands. Line 150 of our program is where the keyboard encoder receives its orders.

Every time you want to send a command to the keyboard encoder, you should do two things;

1. Check bit 1 of the status register to make sure the keyboard processor is ready to accept a command. If bit 1 is zero, it's okay to proceed.
2. Write the command to the command port.

Want a simple subroutine that can be used to issue any command to the keyboard controller? Listing 2 gives versions

Routines to Issue a Command to the Keyboard Encoder
BASIC Routine (Command Byte is in Variable C)

```
1000 IF(INP(&HF5) AND 2)=2 GOTO 1000
1010 OUT &HF5,C : RETURN
```

Listing 2a

'C' Language Routine
(Command Byte is Passed as an Argument)

```
keycom(c)
int c;
    {
    while(inp(0xF5) & 2));
    outp(0xF5, c);
    }
```

Listing 2b

Assembly Language Routine
(Command Byte is Passed in Register DL)

```
KEYCOM  PROC    NEAR
        IN      AL, 0F5H
        TEST    AL, 2
        JNZ     KEYCOM
        MOV     AL, DL
        OUT     0F5H, AL
        RET
KEYCOM  ENDP
```

Listing 2c

of such a routine in BASIC, 'C', and assembly language.

By now you must be getting curious about what types of commands are available, other than the ones to turn autorepeat on and off. Table 1 gives a complete list of the commands that can be sent to the keyboard encoder.

You can try experimenting with some of these commands using your own variation of the BASIC program presented above. Try modifying the program to turn the key click on/off, or enable/disable the keyboard. May I strongly suggest that you save your first trial of the keyboard disable/enable program on disk before trying it, because if you did something wrong you may have to reboot to regain control.

Experimenting with the Scan Modes

Now that we're all experts in sending commands to the keyboard encoder, it's time to move on to more urgent matters. I know that turning the key click on and off is a pretty impressive maneuver, but we generally expect more from a well rounded keyboard interface program. In particular, it would be nice if it would tell us what keys are being pressed.

As mentioned above, the Z-100 offers two different methods of scanning the keyboard, and we're going to try both of them. The sample program shown in Listing 3 serves two purposes; it demonstrates how to use either mode with a BASIC program, and it also tells you what key codes are being generated (for those of you who don't have access to the key code tables in the tech manual).

Some explanations are in order . . .

After you select which scan mode you would like in line 100, the program writes a command to the keyboard which disables the hardware interrupts generated by the encoder. If we don't do this, then nothing will happen in our BASIC program, because BASIC's keyboard interrupt routine will steal all the keystrokes before we get 'em. Even after disabling the hardware interrupt, you'll still notice that a character comes up missing now and then (in other words, you hit a key and no key code number is displayed). This indicates that BASIC is still lurking in the background somewhere, watching over things. Further evidence of this 'big brother' feature of BASIC is that if you continue to hit Control-C, you can regain control (when using ASCII scan mode). As an interesting aside, when you are in UP/DOWN mode, you can sometimes cause the program to break by hitting the 'G' key, since the down code for 'G' is an ASCII three, which is the same as Control-C. Of course, once you break out of the program, you are stuck in UP/DOWN mode, which is sure to cause unexpected results, followed quickly by a crash. Moral of the story . . . be sure to save the sample program before trying UP/DOWN mode.

Keyboard Encoder Command Codes

COMMAND	CODE (in Hex)
Reset	00
AutoRepeat ON	01
AutoRepeat OFF	02
Key Click ON	03
Key Click OFF	04
Clear FIFO buffer	05
Generate Key Click	06
Generate Bell	07
Enable Keyboard	08
Disable Keyboard	09
Event Driven (UP/DOWN) Mode	0A
ASCII Scan Mode	0B
Enable Interrupts	0C
Disable Interrupts	0D

Table 1

Back to the program . . . line 120 is where we select the proper command; 0AH for UP/DOWN mode, or 0BH for ASCII scan mode. Line 150 is a loop similar to the one we used to make sure the keyboard processor was ready to accept a command. Except this loop is checking to see if a key code is ready at the data port. If bit 0 of the status register is set (1), then a key code is ready, otherwise, keep trying. This method of reading the keyboard chip is referred to as 'polling'.

Line 160 is responsible for reading the key code from the data port, and displaying it on the screen. And the code at line 500 is the little subroutine from Listing 2 that sends a command to the keyboard encoder.

This sample program is really rough, but I wanted to keep it as simple as possible, and still demonstrate how to access either scan mode. Obviously, if you are going to write a program that uses UP/DOWN mode, you would want to make sure that the keyboard is left in ASCII scan mode before you exit. Otherwise, DOS wouldn't have a chance.

You may have noticed that I haven't mentioned anything about 'alternate keypad mode' or 'key expansion'. These special keyboard functions are implemented by the Z-100 firmware — not by the keyboard encoder. Whenever you read the keyboard encoder directly (through polling, or an interrupt routine) all you get is the raw key scan codes.

What Use is All This?

A good question! Almost every program you write will probably be quite content to use the normal language keyboard interface functions. In BASIC, the INPUT, INPUT$, and INKEY$ commands will handle just about any situation I can think of. GW-BASIC even has the ability to implement event-trapping subroutines via its ON KEY command.

Pascal and 'C' also have a fairly nice set of keyboard input functions, as long as you can be content with using the normal ASCII scan mode. If you are an assembly language programmer, MS-DOS function calls offer a good keyboard interface, but again, you are limited to the ASCII mode.

As far as I can see, there are only a few legitimate reasons for going to the trouble of accessing the keyboard encoder directly;

1. A memory resident routine may want to avoid using any DOS function calls.
2. You may want to prevent any other programs from intercepting your program's keyboard input.
3. You may have a real-time application that requires use of the UP/DOWN mode.

Reason number one is probably not a valid excuse for going directly to the keyboard hardware. Even though DOS is not re-entrant, you can solve this problem by calling a BIOS keyboard input routine. Not only is this much simpler than writing an entire keyboard input routine yourself, but it also allows you to take advantage of the BIOS type-ahead-buffer.

Reason number two is valid, providing your program must maintain control. Accessing the keyboard hardware directly will prevent most (but not all) memory-resident utilities from working. If this is your intention, then have at it. Just keep in mind that this technique will generally cause your program to be labeled as 'ill-behaved'. (As opposed to 'well-behaved' programs, which follow all the rules, and only use DOS function calls).

Reason number three is the best excuse for writing your own keyboard input routine, and is probably the least understood of all. What do we mean by a real-time application? Well, the most common type of real-time programs are games, but any type of program that depends on knowing exact key movements

BASIC Program to Experiment with Keyboard Scan Modes

```
100 CLS:INPUT" ) ASCII Scan Mode   2) UP/DOWN Scan Mode   :",M$
110 IF M$<>"1" AND M$<>"2" THEN BEEP : GOTO 100
115 C=&HD : GOSUB 500
120 IF M$="1"THEN C=&HB ELSE C=&HA
130 GOSUB 500
150 IF(INP(&HF5) AND 1)=0 THEN 150
160 PRINT INP(&HF4) : GOTO 150
500 IF(INP(&HF5) AND 2)=2 THEN 130
510 OUT &HF5,C : RETURN
```

Listing 3

might be a candidate for UP/DOWN scan mode.

An Interrupt Driven UP/DOWN Mode Keyboard Input Routine

If you are writing a program that needs to use the UP/DOWN scan mode, the most versatile way of doing it would be to use the interrupt driven mode of operation, and write a simple interrupt routine that services the keyboard interrupts. Why not simply poll the keyboard? Well, the main reason is that this ties your program down to polling duties, when it could be doing something more productive. By using interrupts, the keyboard interrupt routine will automatically service any incoming keystrokes without any attention from your application program. Then when your program is ready for a key code, it can simply pull one out of the key buffer.

The assembly language program shown in Listing 4 gives a working, practical example of how an interrupt driven keyboard routine might be implemented. This program uses the UP/DOWN mode, but it can be easily converted to use the ASCII scan mode by simply changing the key code equates, and omitting the code that places the keyboard encoder into the UP/DOWN mode.

To assemble this listing into an .EXE program, you'll need the Microsoft MASM assembler, and the LINK object module linker. Create a source file named UDTEST.ASM using a text editor, and then issue the following commands;

```
MASM UDTEST;
LINK UDTEST;
```

Here's a brief explanation of how the program works. The main line program begins at the START label. This is the entry point when the program is invoked from DOS. The program first saves the original keyboard interrupt vector which points to the BIOS interrupt routine. We need this vector so that interrupts not generated by the keyboard can be passed to the original routine, and we also need to save the vector so it can be restored when we end our program. Next, we install our own keyboard interrupt routine (KBD__INT) in the interrupt table. The keyboard encoder is placed into the UP/DOWN mode, and finally, we call the main program routine (TESTIT).

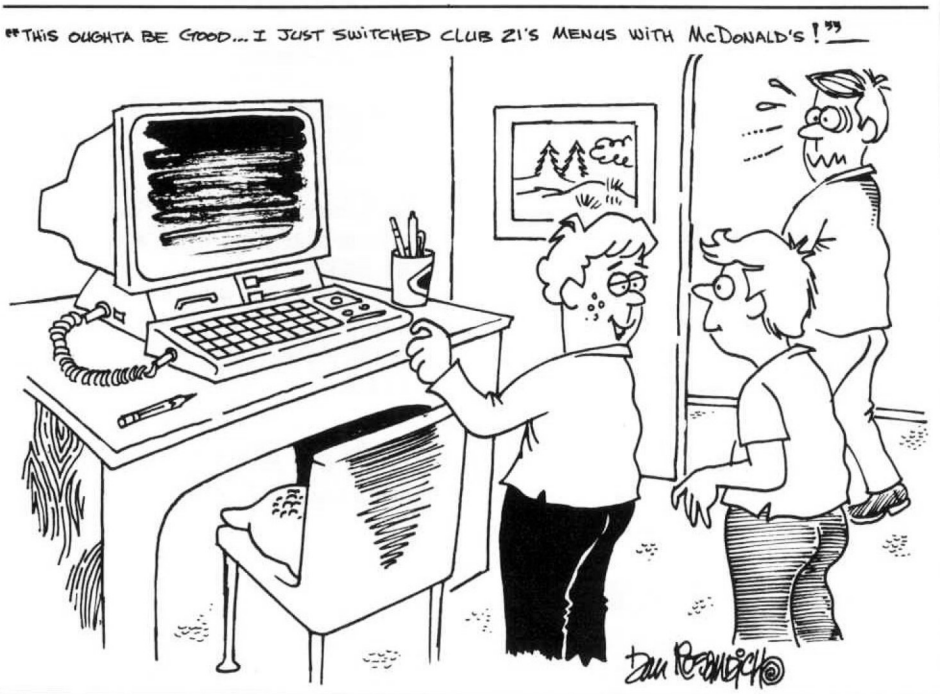
When the TESTIT routine returns (ESC is entered at the keyboard), we need to clean up before we quit. The keyboard encoder is put back in its default ASCII scan mode, and the original BIOS keyboard interrupt vector is restored to the interrupt table. Then we return control to DOS.

The TESTIT routine allows us to move a 'dot' around on the screen by pressing the arrow keys. This is a very simple routine that essentially does nothing, except provide us with a demonstration. In fact,

Listing 4

```
; UDTEST: This program demonstrates how to use UP/DOWN scan mode in an
;   assembly language program.  It sets up its own keyboard interrupt routine,
;   switches the keyboard to UP/DOWN mode, and then allows you to move an
;   asterisk (dot) around on the screen using the arrow keys.

INT_KD              equ     46H             ; keyboard/vertical sync interru
pt
ESC_K               equ     4FH             ; ESC key down code
UP_K                equ     3BH             ; UP arrow key down code
DOWN_K              equ     3AH             ; DOWN arrow key down code
RIGHT_K             equ     33H             ; RIGHT arrow key down code
LEFT_K              equ     3FH             ; LEFT arrow key down code

STKSEG  segment stack
        db      10 dup(?)                   ; don t need much stack
STKSEG  ends

DATSEG  segment
KEYCODE         db      0                   ; current key code
CLS             db      27,'x5',27,'E'      ; initialize screen
                db      'Use arrow keys to move dot, or ESC to end...'
                db      27,'Y',43,71,'*',27,'D$'
CURSON          db      27,'y5$'            ; turn cursor back on
UP              db      ' ',8,27,'A*',27,'D$'   ; move dot up
DOWN            db      ' ',8,27,'B*',27,'D$'   ; move dot down
RIGHT            db      ' *',27,'D$'       ; move dot right
LEFT            db    ' ',8,8,'*',27,'D$'   ; move dot left
DATSEG  ends

PGMSEG  segment
        assume  cs:PGMSEG, ds:DATSEG, ss:STKSEG, es:nothing

START:  mov     ax, DATSEG              ; get our data segment
        mov     ds, ax                  ;
        mov     al, INT_KD              ; save system keyboard interrupt
        mov     ah, 35H                 ;
        int     21H                     ;
        mov     cs:BIOS_I, bx           ;
        mov     cs:BIOS_I+2, es         ;
        push    ds                      ; now, install our interrupt routine
        mov     dx, offset KBD_INT      ;
        mov     ax, PGMSEG              ;
        mov     ds, ax                  ;
        mov     al, INT_KD              ;
        mov     ah, 25H                 ;
        int     21H                     ;
        pop     ds                      ;
ST1:    in      al, 0F5H                ; put keyboard in UP/DOWN mode
        test    al, 2                   ;
        jnz     ST1                     ;
        mov     al, 0AH                 ;
        out     0F5H, al                ;
        call    TESTIT                  ; call the main program
ST2:    in      al, 0F5H                ; put keyboard in ASCII scan mode
        test    al, 2                   ;
        jnz     ST2                     ;
        mov     al, 0BH                 ;
        out     0F5H, al                ;
        mov     dx, cs:BIOS_I           ; restore system interrupt vector
        mov     ds, cs:BIOS_I+2         ;
        mov     al, INT_KD              ;
        mov     ah, 25H                 ;
        int     21H                     ;
        mov     ah, 4CH                 ; return to DOS showing no errors
        mov     al, 0                   ;
        int     21H                     ;

TESTIT  PROC    near                    ;
        mov     dx, offset CLS          ; clear screen, turn cursor off
        mov     ah, 9                   ; and put '*' in center
        int     21H                     ;
TEST1:  mov     al, KEYCODE             ; get the current key code command
        mov     dx, offset UP           ;
        cmp     al, UP_K                ; UP arrow key?
        je      TEST2                   ;
        mov     dx, offset DOWN         ;
```

```
        cmp     al, DOWN_K              ; DOWN arrow key?
        je      TEST2                   ;
        mov     dx, offset RIGHT        ;
        cmp     al, RIGHT_K             ; RIGHT arrow key?
        je     TEST2                   ;
        mov     dx, offset LEFT         ;
        cmp     al, LEFT_K               ; LEFT arrow key?
        jne     TEST4                    ; if none, skip it
TEST2:  mov     ah, 9                   ;
        int     21H                      ; move the dot
        mov     cx, 4000H               ; slow things down a bit
TEST3:  loop    TEST3                    ;
TEST4:  cmp     cs:KEYBUFF, 0           ; wait for another key code from
        je      TEST1                   ; the keyboard interrupt routine
        mov     al, cs:KEYBUFF          ; get the keycode
        mov     cs:KEYBUFF, 0           ; reset to show no keycode available
        cmp     al, ESC_K               ; is this the ESC key?
        je      CLEANUP                 ; yes, prepare to quit
        mov     KEYCODE, al             ; otherwise, store as current code
        jmp     TEST1                   ; process the key code
CLEANUP:mov     dx, offset CURSON       ; turn cursor back on
        mov     ah, 9                   ;
        int     21H                     ;
        ret                             ;
TESTIT  endp

; KBD_INT is our keyboard interrupt routine

        assume  cs:PGMSEG
KBD_INT:push    ax                      ;
        in      al, 0F5H                ; check keyboard status
        test    al, 1                   ;   if none available, must be
        je      JMPBIOS                 ;   vertical retrace or light pen
        in      al, 0F4H                ; get key code
        mov     cs:KEYBUFF, al          ; save it in buffer
        mov     al, 20H                 ; tell interrupt controller that we
        out     0F2H, al                ;   have serviced the interrupt
        pop     ax                      ;
        sti                             ;
        iret                            ; and return from interrupt

JMPBIOS:pop     ax                      ; do a far jump to the BIOS interrupt
        jmp dword ptr cs:BIOS_I         ;   routine so it process interrupt

BIOS_I  dw      0,0                     ; BIOS interrupt routine address
KEYBUFF db      0                       ; keycode buffer (small buffer, eh?)

PGMSEG  ends
        end     START
```



since this program doesn't do anything of philosophical significance, it would have been a lot easier to simply poll the keyboard instead of using an interrupt routine. (In effect, our program is simply polling the one byte key buffer, instead of the keyboard encoder's data port.) But the idea here is to demonstrate how to write an interrupt driven keyboard routine.

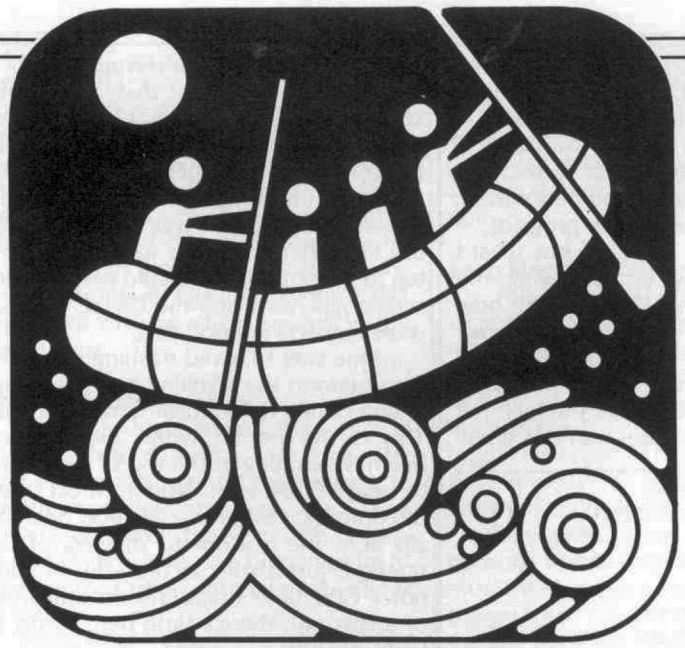
The real heart of our program is the short interrupt routine which begins at the label KBD_INT. This isn't really a part of our main program, but is a separate little piece of code that gets called everytime a key is struck. It is invoked automatically everytime a keyboard interrupt occurs. The first thing our interrupt routine needs to do is check the keyboard status to make sure a key code is available. You might ask why this is necessary, if the interrupt was caused by a key being struck. The answer is that the particular interrupt used by the keyboard is also used by the video display to signal a vertical retrace, and by the light pen.

If we check the keyboard status and find that a key is not ready, we can safely assume that the interrupt was not generated by the keyboard, and simply jump to the original BIOS interrupt routine. If a key code is ready, our routine reads it, and stores it in the key code buffer. A normal program might want to have a buffer that is slightly larger than one byte, in order to prevent keystrokes from being lost while the program is doing other things. This key code buffer is normally referred to as a type-ahead-buffer. Of course, if you want to implement a type-ahead-buffer, you'll need to write special routines that take care of storing and retrieving key codes from the buffer. That's one nice thing about using DOS, or the BIOS, for keyboard input — all this overhead stuff is taken care of for you.

Once the key code is saved, the last thing we need to do is tell the interrupt controller we are done processing the interrupt. This is done by sending a 20H to port 0F2H. I don't want to get into a discussion of the 8259A interrupt controller chip here, so just take my word for it. If you don't do this, nothing will work. You might be asking, "how come we didn't do that before jumping back to the BIOS, when a key wasn't ready?" The answer ... the BIOS interrupt routine does it when it is done. You'll also notice that when we process the interrupt ourselves we must use an IRET instruction to return, but if no key code was ready, the BIOS interrupt routine will issue the IRET.

Wrapping It Up

In the last installment of "Z-100 Survival Kit", I included a Question & Answer section. I plan to include Q&A most of the time, but there just isn't going to be room this month. I promise to do better next time. Until then . . . be sure to keep in touch! *



#6

Z-100

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

SURVIVAL KIT

Those Odd Escape Sequences

All of you are pretty familiar with the various escape sequences that can be used to control the Z-100's screen and cursor. (See Appendix B, "Symbols and Codes", of the Z-100 User's Manual.) For instance, 'ESC E' can be sent to the console in order to clear the screen. But what about those funny escape sequences that aren't so self-explanatory? The ones that no one ever explained how to use? Well, it's about time we learned what they do, and how to use them.

The escape sequences I'm talking about all transmit information from the console, back to the computer. One of them is used to obtain the cursor position;

```
ESC n          Cursor position report
```

Two of them are used to identify the terminal type;

```
ESC Z      Identify as VT52
ESC i 0    Zenith identify terminal type
```

And four of them are used to transmit characters from the console;

```
ESC _      Transmit character at cursor
ESC ^      Transmit current line
ESC ]      Transmit 25th line
ESC #      Transmit page
```

To do justice to this thing, I'll have to split it into two installments of "Z-100 Survival Kit". This issue, we'll talk about the character transmit escape sequences, and all that goes along with that. In "Survival Kit #7," we'll find out about the cursor position, and identifying the terminal type. And I'll also go into some detail about using re-directed input with a program, since these escape sequences can cause problems along those lines.

Getting Information from the Console

Usually, when you think about a computer screen, you think of an output device. You write characters or pixels to the screen, but you don't expect it to talk back. Since the display is memory-mapped, if you want to know something about what is on the screen, you simply read the contents of the video memory.

But before memory mapped displays became the rage, most computers had to make do with something called a 'terminal'. This was a separate piece of equipment which contained the video screen, and a keyboard. Input from the keyboard was received via an RS-232 line, and text output was transmitted to the terminal over the same line. Most popular terminals also had the capability to move the cursor, and perform other screen control functions, in addition to displaying text. These commands were generally prefaced by an ASCII escape character, to differentiate them from normal text.

But there was no way for the computer to tell what was displayed on the screen at any given time. Since many programs (like screen dumps, or interactive programs) needed to know what characters were in certain positions, some terminals (like the Heath H-19) had a method for overcoming this problem. An escape command was available, which when received by the terminal, caused the terminal to transmit a character (or characters) back to the computer. To the host computer, it was just like the characters were being typed on the keyboard of the terminal.

Upward Compatibility

When the Z-100 was designed, one of the major considerations was that it should be able to run 8 bit CPM software. (That was before MS-DOS had such a firm foothold). And most of that 8 bit software owned by Heath computer users was designed to be run on older Heath computers, like the H-8 with an H-19 terminal, or the H-89. The only way to get around making major changes to the existing software base was by allowing the Z-100 to emulate an H-19 terminal. And that's where we get all of these escape sequences from. They were all developed for the H-19 terminal (and therefore, for the H-89 computer).

I'm diverging from the main subject a little, but I just wanted all you new kids to know why the Z-100 has some of these weird escape commands. They're just holdovers from the previous Heath computers.

How Does This Work?

All of these terminal emulation features are implemented in the MTR-100 monitor ROM program. It is responsible for translating any escape sequences sent to the console, and performing the required function.

Most of the escape functions are performed the instant they are received — in fact, most just require that one or another flag byte be changed to indicate a new condition. But the transmit character functions work a little differently.

Whenever you tell the console to transmit a character, line, or page, it sim-

ply makes a note of the request, and then returns control back to your program. The note that it makes is kept in the MTR-100 data segment, in an area called the 'Transmit Structure'. Many of you may have seen this Transmit Structure in the monitor ROM listing (included with the Z-100 Technical Manual set) and wondered what in the heck it was for. Well . . . prepare to be enlightened.

The MTR-100 Transmit Structure is located at offset 2E7H of the ROM's data segment. (This offset will be different in older versions of the ROM.) Here's what it looks like . . .

```
XMT_STRUC
    BURST        DB    ?        ; Characters to transmit per VSYNC
    BCOUNT       DW    ?        ; Remaining characters in current burst
    COUNT        DW    ?        ; Characters left to transmit
    COL          DB    ?        ; Horizontal column to transmit
    ROW          DB    ?        ; Vertical row to transmit
    XMT_COLOR     DB    ?        ; Current color state transmitted
    XMT_GRAPHIC  DB    ?        ; Current graphic state transmitted
    XMT_REVERSE  DB    ?        ; Current reverse video state transmitted
XMT_STRUC ENDS
```

Let's assume for a moment, that you send the escape sequence ESC – (transmit current line) to the console. Regardless of whether the command was sent using BASIC, assembly language, the DOS command line, or whatever, control is eventially routed to the monitor ROM program. The ROM program makes a note of which line the cursor is on, and saves this information in the ROW variable of the transmit structure. It puts a zero in the COL variable to indicate that the transmission should start with the first character in the line. And finally, it sets the COUNT variable equal to 80, indicating that the entire 80 column line will be transmitted. Then, control returns to your program.

If you're trying to figure out what's happening here (by disassembling the ROM code) as I have done, you're now left with a puzzle — because no characters got transmitted. A few variables were changed, but how does that accomplish anything? The answer lies in the fact that the characters will be transmitted later. All the escape sequence did was cause the ROM program to set up the Transmit Structure with instructions about what should be transmitted.

Next question. When are the characters transmitted, and by what mechanism? This takes a little more digging in the ROM code, but the solution is both clever and elegant.

The Vertical Retrace Interrupt

Some operations that affect the screen need to be done at a time when the screen is not being refreshed. Moving the cursor is one of these operations, but the most important one is scrolling the screen. If the screen was scrolled during the time that the pixels were being written to the screen, you would get annoying screen glitches when the CRT-C start address registers are updated.

One way to avoid trashing the video is to perform the scrolling and cursor updating during the vertical retrace interval. This is the period when the electron beam is returning to the top of the screen to begin a new scan. Now to most of us, the time it takes for the electron beam to zip from the bottom to the top of the screen is just a split second. But in computer CPU time (measured in millionths of a second), there's time here to do all kinds of stuff.

In order to take advantage of this 'nap' the video takes 60 times a second, the Z-100 generates a hardware interrupt whenever a vertical retrace begins. And the MTR-100 monitor ROM has an interrupt service routine that jumps right in and starts updating the screen whenever it gets control. The first thing it does is scroll the screen, if necessary. Then it checks to see if the cursor needs to be moved. And finally . . . yes, you guessed it

. . . it transmits any characters that need to be transmitted. But since the vertical retrace period is somewhat limited, only a few of the characters are transmitted at a time — typically 16 characters at a shot. This transmission of a small group of characters is called a 'burst'. The exact number to be transmitted during each retrace interval is held in the variable BURST in the Transmit Structure. The variable BCOUNT keeps track of how many are left in the current burst.

Accessing Video Memory

All of this might seem fairly logical, if not just a little complicated, so far. The monitor program records the request to transmit characters, initializes the Transmit Structure, then transmits the characters during vertical retrace time in order to prevent any screen interference. Right? Not quite.

That's what I thought at first too, but there's a catch. All that's required in order to transmit a character, is to read the video memory and put the character in the keyboard buffer. Well, putting the character in the keyboard buffer obviously doesn't interfere with the screen, and neither does reading the video memory.

True, in some PC compatible computers (like the original true blue IBM-PC), indiscriminate reading or writing to the video memory would cause video interference. But us Z-100 owners are lucky, because we don't have that problem. The reason we don't is because the CRT-C controller has priority over the CPU when it comes to video RAM access. Basically, this means that if your program tries to access video RAM at the same time as the CRT-C controller (for screen refresh), the CRT-C controller tells your program to take a hike until the video refresh is done. In technical terms, the video circuitry issues wait states to the CPU.

So if reading characters from the screen doesn't cause any video glitch problems, why does the ROM wait until vertical retrace time? Why doesn't it simply transmit them all at once when the escape command is received? Good question.

Searching for a Motive

As long as we know how the transmit character routines work, I don't guess it matters a whole lot why they do it the way they do. But whenever I get involved in figuring out how something works, it bothers me if I can't see the underlying logic. The situation I have been describing here is puzzling, but the answer, once known, is ridiculously obvious.

Whenever your program asks the console to transmit some characters, it must read those characters as they are transmitted. You do this just as if you are reading from the keyboard. Here's an example of how you might read a line from the console with a BASIC program . . .

```
100 INPUT"Enter line number to read"; L
110 LOCATE L, 1                          ' get line number
120 PRINT CHR$(27);"^";                  ' make this the current line
130 WHILE I$<>CHR$(13):I$=INPUT$(1):L$=L$+I$:WEND' read the transmitted chars
```

But if the monitor ROM program simply transmits all the characters at once when the escape command is received, how is your program going to gain control to receive them? In other words, by the time line 130 receives control, all the characters would have been transmitted. And the LINE INPUT statement would be waiting for nothing.

In order to make this character transmission scheme work, the monitor program needs to send a character, and then turn control back to your program, so it can read the character. Then another character can be sent, and another, until the transmission is complete. Actually, more than one character can be sent at a time, since the BIOS maintains a type-ahead-buffer which will hold quite a few characters.

Is it beginning to be obvious why the ROM program uses the vertical retrace period to send just a few characters at a time? It doesn't have anything to do with video timing. The retrace interrupt just happens to be a convenient way of transmitting the characters, while allowing your program to keep control so they can be read.

Receiving the Transmission

Okay, I can see most of you have had enough of this theoretical stuff. Time to put this information to use. The BASIC program example above gives you an idea of how simple it is to read a line from the screen, using the 'transmit current line' escape sequence.

Here is a BASIC subroutine that could be called from your program to read a line from the screen. It is similar to the program above, but variable L should be set to the proper line number before calling the subroutine. The characters from the line are returned in the string L$.

```
1000 LOCATE L, 1
1010 PRINT CHR$(27);"^";
1020 WHILE I$<>CHR$(13):I$=INPUT$(1)
                              :L$=L$+I$:WEND
1030 RETURN
```

Here is a similar routine in the 'C' language . . .

```
char linebuff[255];

getline(line)
int line;
    {
    char *buff;
    printf("\33Y%c \33^", line+32);
```

```
    buff = linebuff;
    while(*buff++ = getch()) != 13);
    }
```

And here it is again, in assembly language . . .

```
DATA    segment
HEADER     db    255,0                    ; needed for DOS function 10
LINEBUFF   db    255 dup(?)
ESCTEXT    db    27,'Y ',27,'^','$'
DATA    ends

GETLINE proc
        mov     di, offset ESCTEXT
        add     al, 32                    ; line number is passed in
        mov     [di+2], al                ; register AL
        mov     dx, di
        mov     ah, 9
        int     21H                       ; send commands to console
        mov     dx, offset HEADER
        mov     ah, 10
        int     21H
        ret
GETLINE endp
```

There's More Here Than Just Characters

If you write a program that uses one of these routines, and then run the program, you may be surprised at the contents of the transmitted line. There could be more in the line buffer than just ASCII text. So it's time we discussed the format used for the character transmission.

The first thing to realize is that characters on the screen are more than just ASCII codes. They can also be displayed in different colors. And they can be normal text or graphics characters, as well as normal or reverse video. When the ROM program transmits characters from the screen, it wants to make sure you know about the attributes of the characters, as well as the ASCII codes.

It does this by transmitting escape sequences along with the characters, in exactly the same way you use escape sequences to display characters with special attributes. The character transmission routine keeps track of the text color, graphics mode, and reverse video mode. If any changes occur in the color or modes, the appropriate escape sequence is transmitted to indicate the change. At the start of the transmission, the color is assumed to be white text with a black background, graphics mode is assumed to be off, and reverse video is assumed to be off.

Let me give you an example. The screen line we want to transmit consists of the word "ONE" in green text, followed by the word "TWO" in white text, followed by the word "THREE" in white reverse video. All three words are separated by a single space. Here are the codes that will be transmitted . . .

```
<ESC> m 4 0 0 N E   <ESC> m 7 0 T W O
                <ESC> p;T;H;R;E;E;<ESC>;q
```

. . . followed by 67 spaces, and a carriage return.

The idea here is that if you transmitted this exact string of characters back to the console, the characters would appear just as they did originally. There are only five escape sequences that will ever be transmitted from the console;

```
ESC m <fore> <back> change the foreground
                    and background colors
ESC p     enter reverse video mode
ESC q     exit reverse video mode
ESC F     enter H-19 (block) graphics mode
ESC G     exit graphics mode
```

Every transmission (even transmit character) is terminated with a carriage return. This is necessary since there is no way to tell in advance how many characters may be transmitted. Consider for instance, that the 'transmit character at cursor' command could result in a ten byte string of characters, if the character is a non-white, reverse video graphics character.

Special Problems Using BASIC

If you are using the BASIC programming language, be forewarned that BASIC plays all kinds of games with the escape sequences which are transmitted. The exact rules to the games will depend on which version of BASIC you are using.

For instance, ZBASIC version 1.x filters out the <ESC> character from the transmission, but leaves the scape sequence operands. In other words, if there is a color change, the <ESC> character won't make it, but the 'm' and color numbers do.

If you are using GWBASIC v2.x, the <ESC> characters come through, but instead of being CHR$(27), they show up as CHR$(1). Strange!

One way to avoid this situation, if all you want is the straight text (without attributes) is to use a LINE INPUT statement, instead of reading each character individually. Like this . . .

```
1000 LOCATE L, 1
1010 PRINT CHR$(27);"^";
1020 LINE INPUT""; L$
1030 RETURN
```

This is actually a much simpler way of doing it . . . and the LINE INPUT statement will strip out everything except the text characters themselves.

Buffer Considerations

One of the problems you will have to consider when writing a program that uses the transmit character escape sequences (especially transmit page) is how large to make the buffer which will hold the characters. There are 1920 text positions on

```
#define YES     (-1)
#define NO      0
#define ESC     27

struct {                          /* global character info structure      */
    char value;                   /* ascii value                          */
    char fore, back;              /* foreground, background colors        */
    char gmode, rmode;            /* graphic flag, reverse video flag     */
    } chr;

scrnchr(row, column)
int row, column;
    {
    char ch, buff[11], *bp;
    chr.fore = chr.back = '7';                  /* assume default values    */
    chr.gmode = chr.rmode = NO;
    printf("\33Y%c%c", row+32, column+32);      /* position cursor          */
    printf("\33_");                             /* transmit char command    */
    bp = buff;
    while((*bp++ = getch()) != 13);
    bp = buff;
    while(*bp++ == ESC) {                       /* translate escape seq s   */
        switch(*bp++) {
            case 'm': chr.fore = *bp++; chr.back = *bp++; break;
            case 'p': chr.rmode = YES; break;
            case 'F': chr.gmode = YES; break;
            }
        }
    chr.value = *(--bp);                        /* and get ascii code       */
}
```

Listing 1
'C' Function to Read Character at Cursor

```
CODE    segment
        assume cs:CODE

XMT_CMD db      27,'#$'          ; escape sequence to transmit page
BUFFPTR dw      offset BUFFER    ; pointer to current character position
BUFFER  db      2000 dup(?)      ; buffer for transmitted characters
DONE    db      0                ; flag indicating transmission is complete

GETPAGE proc near
        mov     ax, 0                       ;
        mov     ds, ax                      ;
        mov     bx, 3FEH                    ;
        mov     ds, [bx]                    ; get MTR-100 data segment
        push    ds                          ; save for later
        mov     bx, 83H                     ;
        push    [bx]                        ; save BIOS routine address
        push    [bx+2]                      ;
        mov     [bx], offset STUFF          ; patch in our routine
        mov     [bx+2], cs                  ;
        push    cs                          ; set DS = CS
        pop     ds                          ;
        mov     dx, offset XMT_CMD          ; output transmit page command
        mov     ah, 9                       ;
        int     21H                         ;
PS2:    cmp     DONE, 0                     ; wait until characters received
        je      PS2                         ;
        pop     ds                          ;
        pop     [bx+2]                      ; restore system S_XMTC address
        pop     [bx]                        ;
        ret                                 ;
GETPAGE endp

; THIS IS OUR ROUTINE TO HANDLE THE TRANSMITTED CHARACTERS

STUFF   proc far
        push    di                          ;
        mov     di, cs:BUFFPTR              ; get current buffer position
        cmp     di, offset DONE             ; out of buffer space?
        je      STUFF1                      ; yep, bail out
        mov     cs:[di], al                 ;
        inc     di                          ; ready for next character
        mov     cs:BUFFPTR, di              ;
        cmp     al, 13                      ; end of transmission?
        jne     STUFF2                      ;
STUFF1: mov     cs:DONE, 0FFH               ;
STUFF2: pop     di                          ;
        ret                                 ;
STUFF   endp

CODE    ends
```

Listing 2
Assembly Language Routine to Transmit Page

the screen, but in a worst case example, the transmit page routine could send back as many as 19,200 characters (ten for each text character). Of course, that's not very likely, but where do you draw the line? Is 2000 bytes enough? 3000 bytes?

My suggestion would be to do one of two things. If the character attributes are important to your application, then make your best guess about how many bytes will be transmitted for a typical screen. Be liberal, unless you're really running tight on memory. Then, design your character input routine to check for buffer overflow. It can then either discard the extra characters, or generate an error message.

The other way to handle the problem is to have your character input routine filter out the escape sequences (like the BASIC LINE INPUT statement does). This way you will know exactly how many characters to expect. Transmit character will be one byte, transmit line will be 80 bytes, and transmit page will be 1920 bytes. And, of course, each of these will have the obligatory carriage return on the end, which can also be filtered off.

Let's Take it a Character at a Time

Probably the most useful of the character transmit escape sequences is the one that transmits the character at the current cursor position. It's not often that a program would need to know every character on the screen, but it's easy to think of reasons you would want to read a single character at a specified location.

BASIC already has the SCREEN function (not to be confused with the SCREEN command) which reads the character at the cursor position. So there's no sense in showing you how to use the 'transmit character at cursor' escape sequence in BASIC. But the 'C' language has no such function (none that will work on the Z-100, that is). Listing 1 is a 'C' function that will read the character at the cursor position, along with its attributes. The character value, colors, and modes are returned in an external data structure.

A Page Full of BEEPs

If I asked for a show of hands from the people who have (on their own) been able to successfully use the 'transmit character at cursor' and 'transmit line' escape sequences, I expect there would be quite a few. But I'm willing to wager that almost no one has been able to get 'transmit page' to work correctly. Am I right? Be honest now.

When I first started playing around with it (having already mastered the character and line routines), I got some unexpected results. It seems like all you would have to do is expand the 'transmit line' routine to read about 2000 characters, and you'd have it. But such is not the case. Everytime I tried to transmit the page, I would just get a long BEEP, and the characters read from the console were all goofed up. The first couple of lines were okay, but then all kinds of characters would come up missing. The rest of the transmission was useless.

To make a long story short, I hacked the code for quite a while, trying to find out what was happening . . . in particular, what was causing that annoying BEEP. When I finally found the answer, it was something I should have suspected all along. It's amazing how clear things become after you figure them out!

The long BEEP you hear whenever you send the ESC # (transmit page) command to the console, is just the BIOS telling you that the type-ahead-buffer can't hold any more characters. The bad news is that the ROM program is transmitting the characters faster than they can be read from the type-ahead-buffer. There doesn't seem to be any way to read the characters fast enough (not even with assembly language), so the slowpoke that's causing the problem must be the DOS keyboard input routine.

Ah . . . someone is asking "How come this problem doesn't occur when a line is transmitted?". Well, you can thank the type-ahead-buffer for that. The buffer (in a Z-100) is longer than a line. Once all the characters are in the buffer, then they can be read at any speed that is convenient. But, when the ROM program tries to transmit a full page, the buffer quickly fills up, because the characters aren't being read fast enough.

The good news is that there is a solution to this problem. Actually, I can think of two ways of handling the situation.

```
CODE    segment
        assume cs:CODE
        org 100H

START:  mov     ax, 0                   ;
        mov     ds, ax                  ;
        mov     bx, 3FEH                ;
        mov     ds, [bx]                ; get MTR-100 data segment
        mov     bx, 2E7H                ; offset of BURST variable
        mov     byte ptr[bx], 8         ; (must be MTR-100 v2.x)
        push    cs                      ; set DS = CS
        pop     ds                      ;
        mov     dx, offset XMT_CMD      ; output transmit page command
        mov     ah, 9                   ;
        int     21H                     ;

        mov     di, offset BUFFER       ; READY TO READ TRANSMITTED CHARACTERS
        mov     cl, 0                   ;
PS2:    mov     ah, 7                   ;
        int     21H                     ; get a transmitted character
        cmp     al, 13                  ; end of transmission?
        je      PS6                     ; yes, skip ahead
        cmp     al, 1BH                 ; escape character?
        jne     PS3                     ;
        mov     cl, -1                  ; yes, flag to look for operand
        jmp     PS2                     ;
PS3:    cmp       cl, -1                  ; look for escape operand?
        jne     PS4                     ;
        mov     cl, 0                   ;
        cmp     al, 'm'                 ; yes, if an  m , skip next two  chars
        jne     PS2                     ;
        mov     cl, 2                   ;
        jmp     PS2                     ;
PS4:    cmp     cl, 0                   ; skip this character?
        je      PS5                     ;
        dec     cl                      ; yes, decrement skip count
        jmp     PS2                     ;
PS5:    mov     [di], al                ; save character
        inc     di                      ; bump pointer
        jmp     PS2                     ;

PS6:    mov     si, offset BUFFER       ; READY TO PRINT
        mov     cx, 24                  ; will print 24 lines
PS7:    push    cx                      ;
        mov     cx, 80                  ; 80 columns per line
PS8:    lodsb                           ; get character
        mov     dl, al                  ;
        mov     ah, 5                   ;
        int     21H                     ; print it
        loop    PS8                     ; next character
        mov     dl, 13                  ;
        int     21H                     ; print CRLF at end of line
        mov     dl, 10                   ;
        int     21H                     ;
        pop     cx                      ;
        loop    PS7                     ; next line

        int     20H                     ; return to MS-DOS

XMT_CMD db      27,'#$'                 ; escape sequence to transmit page
BUFFER:                                 ; buffer area is in memory above program
CODE    ends
        end     START
```

Listing 3
PS.COM — A Print Screen Program which Uses Transmit Page

The first way is to simply bypass the BIOS routine, and have the ROM program transmit the characters right to your program's buffer. Each character that is transmitted from the screen is passed in register AL to a BIOS routine whose address is stored at offset 83H in the MTR-100 data segment. This address is stored in four bytes (two words); the first word is the offset, and the second word is the segment. Listing 2 shows how this might be done in assembly language . . .

Of course, this routine is designed to be included in a program. It is simply a subroutine that allows you to read the screen. One important thing to note is that the STUFF routine (our substitute for the BIOS routine) must be declared as a FAR procedure. This enables the assembler to generate the correct far return instruction needed to get back to the calling routine in the ROM program.

Now that I've made you suffer through that, I'll tell you about the other method of making 'transmit page' work correctly. This way is actually much simpler, and is probably the way the designers expected the situation to be handled. We'll just slow down the transmission rate of the characters so our normal keyboard input routine can keep up. That sounds pretty logical, doesn't it?

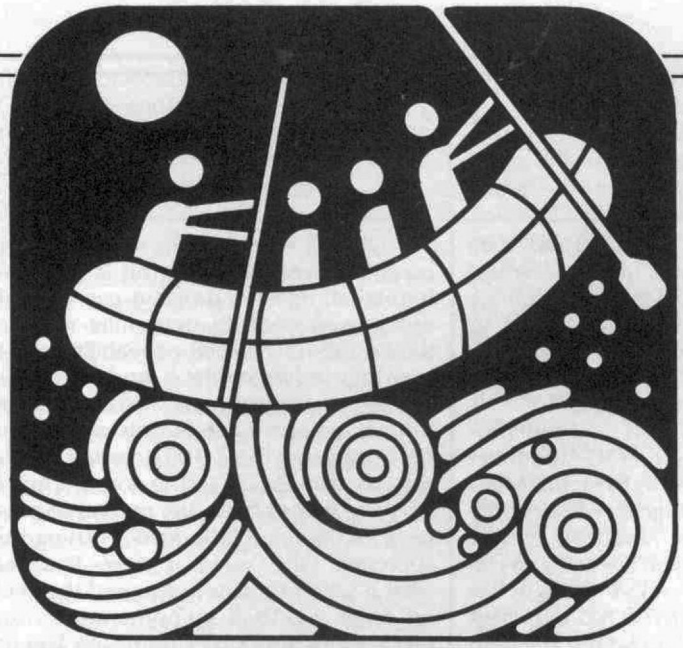
Remember way back toward the beginning of the column, when I was boring you to death with all the details of how the character transmission scheme worked? All that stuff about Transmit Structures, and Bursts, and such? Well here's a case where it helps to know the method behind the madness. How about if we told the ROM program to transmit less than 16 characters per burst? (Remember, a burst of 16 characters are transmitted during each vertical retrace period.) If we set the BURST variable for, say a 2 character burst, then only two characters would be transmitted during each vertical retrace. This works out to 120 characters transmitted per second, instead of the normal rate of 960 characters per second. And guess what? It works beautifully!

Listing 3 is a complete assembly language program which will dump the entire screen to your printer (PRN device). It uses the slow-down technique discussed above. You'll note that we are using a BURST value of eight. You may have to experiment with different values in your own programs to see how fast you can allow the characters to be transmitted. Using interpreted BASIC, for instance, you'll be lucky to use a BURST of 3, and get away with it. Notice also, in Listing 3, that the program filters out all the escape sequences from the transmission, since most printers would choke on them.

This program may not be much good for anything, since we normally think of a print screen program as being memory resident. (Although at 110 bytes, I believe it's probably the shortest print screen program around.) But the listing shows how you can use the transmit page feature in your own program.

Until Next Time

Well I've done used up all my space again! Remember, next installment we'll be discussing some more of those mystifying escape sequences. And I'll also be touching on the subject of using redirected input to automatically execute a program from a script file. And hopefully, there will still be a little room for a small Q & A section. 'Till then . . . keep in touch!



#7

Z-100

SURVIVAL KIT

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

Cursor Position, Terminal Identification, and Re-Directed Input

In the last issue of "Survival Kit", I began an in-depth discussion of those odd Z-100 escape sequences which cause characters to be transmitted from the console. This column, we're going to look at several more of these escape sequences. One that tells you the current cursor position, and two that are used to determine the Z-100 system configuration. I'll also talk a little about re-directed input, and how these odd escape sequences can affect that capability.

Maneuvering the Cursor

There are lots of ways to manipulate the cursor on the screen. The most obvious of these is by sending text characters, tabs, line feeds, or carriage returns to the console. All of these commands cause the cursor to move to a new location, and are understood by even the dumbest of terminals.

Smarter terminals will allow you to tell the cursor to go directly to a specified line and column position. On the Z-100 this is done with the escape sequence:

```
ESC Y <row> <column>
```

. . . where 'ESC' is the ASCII escape command (decimal 27) and <row> <column> denote single ASCII characters which indicate the destination row and column.

Since ASCII codes below 32 are considered to be non-printing characters, the row and column characters in the escape sequence are offset by 32. In other words, if you want to move the cursor to the first column of the first row, you would issue the following command:

```
ESC Y <space> <space>
```

<space> indicates an ASCII space character (decimal 32). The ASCII byte equivalent of the escape sequence above would be like this:

```
27 89 32 32
```

The proper ASCII characters for the row and column can be found by adding 32 to the desired row or column number. This assumes that the first column of the first row is row zero, column zero. As another example, suppose you want to move the cursor to row 8, column 56. Adding 8 to 32 gives 40, which is the ASCII code for a left parenthesis. Adding 56 to 32 gives 88, which is the ASCII code for an uppercase X. Therefore, you would use the escape sequence:

```
ESC Y ( X    or . . .    27 89 42 88
```

Finding the Cursor

There are several ways for a program to determine where the cursor is located at any particular time. Perhaps the most straightforward of these is for the program to simply keep track of the cursor position. After all, the program is in control of the cursor position, so it should be able to keep track of where it is at. The program would need to have two variables, one for the row position, and the other for the column position. And it would need to update these variables whenever the cursor was moved, or text was output to the screen.

This sounds simple enough, until you begin to think about what is involved in keeping track of the cursor position. For instance, the program would have to check each string of ASCII text sent to the console to see if there are any special characters, like backspaces, tabs, carriage returns, or line feeds. The program would also have to check output to the console to determine if there were any escape sequences which move the cursor. This would be a major challenge, since there are many escape commands which affect the cursor position.

As long as a program is constrained in its use of special characters, it is feasible to determine the cursor's position by updating variables. But if your program will require a lot of flexibility in cursor positioning (such as would be the case with a word processing or spreadsheet program, or a game) there is just too much overhead involved with keeping track of the cursor. A better way is needed, and the Z-100 provides one. The following escape sequence can be sent to the console to inquire about the cursor position:

```
ESC n
```

This escape sequence is referred to as a "cursor position report". This is one of those odd escape sequences which transmits characters from the console. You may have wondered why I got sidetracked talking about the ESC Y escape sequence above, since it does not transmit any characters from the console. The reason is because ESC Y and ESC n perform opposite functions, and work quite similarly.

Whenever you send the escape sequence ESC n to the console, the console responds by transmitting four ASCII characters. The format of these characters is exactly the same as the ESC Y command. For example, if the cursor was sitting at row 8, column 56 on the screen when you requested a "cursor position report" with

ESC n, the console would respond by transmitting these four characters:

```
ESC Y ( X
```

Look familiar? These are exactly the same characters you would have transmitted if you had wanted to move the cursor to that position.

Here is a BASIC subroutine that will find the cursor coordinates:

```
100 PRINT CHR$(27);"n";
110 I$=INPUT$(4)
120 ROW=ASC(MID$(I$,3,1))-31
130 COLUMN=ASC(MID$(I$,4,1))-31
140 RETURN
```

This demonstrates how to use the "cursor position report" escape sequence to determine the cursor location. Note that we are simply discarding the first two characters returned by the console (ESC and Y). The only ones we are interested in are the third and fourth characters, which tell us the row and column. You'll also notice that we are subtracting 31 from the ASCII row and column codes, instead of 32. This is because BASIC numbers the rows and columns starting with row one, column one, instead of row zero, column zero.

Here is a 'C' language function that does the same thing:

```
getpos(r, c)
int *r, *c;
        {
        printf("\33n");
        getch();
        getch();
        *r = getch() - 32;
        *c = getch() - 32;
        }
```

In order to use this function, you should declare integer variables for the row and column, and then call the function like this:

```
int row, column;
getpos(&row, &column);
```

Since the 'C' language can only directly return one value from a function, we are passing the address of the row and column variables to the function so it may directly update the values. The alternative would be to have separate functions which return the row or the column position of the cursor.

Going Directly to the Source

It's said there is more than one way to skin a cat. And there are at least three ways to find the cursor position. We have discussed two of them; having your program keep track, and using the "cursor position report" escape sequence. So you might ask, "how does the console know where its cursor is at?". Obviously, it has to be keeping track of the cursor position, if it is able to tell you the coordinates.

The MTR-100 monitor ROM program is responsible for processing the cursor escape sequences, and the current cursor coordinates are stored in its data segment at the following offsets;

	Offsets into MTR-100 data segment	
	v1.x	v2.x or greater
HORZ_CHAR (column)	028FH	0291H
VERT_LINE (row)	0290H	0292H

The address of the MTR-100 data segment can be found in the interrupt page, at address 0000:03FE. For assembly language programs, it may be easier to find the cursor position by directly reading the MTR-100 data, instead of the other methods we have discussed.

One last consideration . . . if your program directly modifies the CRT-Controller chip cursor registers (R14, R15), the MTR-100 monitor ROM program will become confused, and will not report the correct cursor coordinates. If you are going to use the "cursor position report" escape sequence, or read the MTR-100 data segment directly, you must use the standard console commands and escape sequences to move the cursor.

Identifying the Terminal Type

Two of the odd escape sequences we have mentioned can be used to determine the terminal type with software. They are:

```
ESC Z       Identify as VT52
ESC i 0     Zenith identify terminal type
```

Now I hate to sound ignorant, but I don't know what the heck a VT52 terminal is, let alone what it would be used with. As far as I know, such a thing does not exist today, and if there is a standard terminal protocol built around the VT52, it must not be terribly popular.

The most important thing we need to know about the VT52 is that whenever it receives the escape sequence ESC Z, it transmits back the sequence:

```
ESC / K          or ...  27  47  75
```

That's it. Nothing more to it. Presumeably, this escape sequence would be used by a communications program to determine if it is talking with a VT52 type of terminal. By returning the characters ESC / K, the Z-100 is merely saying "yes, I understand".

Be a Little More Specific

The ESC i 0 "Zenith identify terminal" escape sequence is a lot more interesting than the VT52 one. This is because the console not only responds to the inquiry, but it does so with meaningful information about the video configuration. Whenever you send the sequence ESC i 0, the console responds by transmitting the following characters:

```
ESC i E <pov><vrs>
```

where . . .

<pov> is a character that denotes the number of planes of video RAM. It will either be '1' or '3'.

<vrs> is a character that tells the video RAM chip size. It will be 'A' for 32K chips, or 'B' for 64K chips.

Okay, it isn't anything worth jumping up and down about, but this is useful information, right? A program can use this escape sequence to determine whether the host Z-100 has color capability, or if it may be used in interlace mode.

Some programmers also like to use this escape sequence to determine if the host computer is a Z-100. In other words, they send an ESC i E to the console, and if anything comes back, the computer must be a Z-100. I'm a little leary of using this approach. Oh, I guess it works fine, but what if your program performed this trick on some MS-DOS computer that used ESC i E for some other purpose? The results could get strange.

Here is a little BASIC program that can be used to determine the Z-100's video configuration using the "Zenith identify terminal type" escape sequence:

```
100 PRINT CHR$(27);"i0";
110 I$=INPUT$(5)
120 PVR=ASC(MID$(I$,4,1))-48
130 VRS=(ASC(MID$(I$,5,1))-64)*32
140 PRINT PVR;"planes of video RAM,
                  using";VRS;"K chips."
```

Redirected Input

One of the more interesting features of MS-DOS version 2 and above is redirected input/output. Redirected output gives you the ability to route the normal screen output of a program to another device, like a disk file or the printer. And redirected input allows a program to take its keyboard input from another source, like a disk file. Since this is a Z-100 specific column, and I/O redirection is a generic DOS feature, I'm not going to spend much time describing how to use these features here. However, there are some peculiarities of using redirected input with the Z-100 that I'd like to mention. But first, a short introduction.

Most of you are probably familiar with using redirected output. You can use it to output a disk directory to your printer:

```
DIR > PRN
```

or you can use it to map error messages and other screen output of a program to a disk file, for later reference. For instance, if your program was named TEST.EXE, you could use this command:

```
TEST > ERROR.TXT
```

But when it comes to redirected input, many of you are wondering "what good is it?" Why would you want to redirect the input of a program? Well, I could list quite a few situations where redirected input is useful, but far and away the most valuable is the ability to use a script file to automate the input to a

program. For example, create a file named SCRIPT.TXT consisting of the following lines (<RET> indicates the RETURN key):

```
B <RET>
E <RET>
<RET>
H <RET>
E <RET>
```

Now execute the command:

```
CONFIGUR < SCRIPT.TXT
```

This will cause serial port B to be automatically configured for the Diablo 630 printer, using the DOS CONFIGUR program. The way it works is by running the CONFIGUR program, and taking the required keyboard input from the file SCRIPT.TXT. In other words, every time the CONFIGUR program expects a key to be typed at the console, it takes one from the file, instead.

The advantage to doing this, is that the configuration process can now be done automatically (say from a batch file) without any attention from the operator. If you have a program that requires you to use a different printer, you could reconfigure "on-the-fly" with a batch file. Like this:

```
ECHO OFF
ECHO Switch to Daisywheel Printer
                              please . . .
CONFIGUR < DAISY.CNF
EDITOR
ECHO Switch to Dot Matrix Printer
                              please . . .
CONFIGUR < DOTMAT.CNF
```

In this batch file, the file DAISY.CNF would contain the configuration commands for the daisywheel printer, and DOTMAT.CNF would contain the command keystrokes for the dot matrix printer. EDITOR is the name of the program you are running that needs to use the daisywheel printer. Get the picture?

You can use this technique of using redirected input with many programs. In order to find out which characters need to be put in the SCRIPT file, simply run the program, and keep track of every keystroke you make. You can do this by simply making a note of each keystroke on a piece of paper as you are going through a trial run of the program. Remember to record EVERY keystroke, even the carriage returns and control codes. Remember also, that you must record the keystrokes required to exit the program, and return to the DOS prompt. Otherwise, the program will not return control to DOS when you run it using redirected input. After you have a record of all the keystrokes, use an editor program to create a disk file composed of the keystokes you have recorded. If all the keystrokes are plain ASCII printable characters, you can just use a text editor, like EDLIN. However, if the program required any control characters (like Control C), you may need to use an editor like DEBUG, which allows non-printable characters to be included in the file.

And Now for the Bad News

A while back, someone wrote to me saying that he would like to see a version of SETZPC (the program used to configure ZPC) that would allow all ZPC parameters to be specified on the command line. As it is now, SETZPC is an interactive program which requires the user to answer questions about his desired configuration. This person wanted to be able to invoke SETZPC from a batch file, and automatically change the ZPC parameters, without further user intervention. Quite a reasonable request, I would say.

My first response to this inquiry was that he should ask Pat Swayne, author of the program. But then the idea struck me that it should be possible to use redirected input with SETZPC, along with an appropriate script file, to automate the configuration process. So I began experimenting with that idea.

I stepped through the SETZPC process of selecting each ZPC parameter, noting each key that was used. Then I made a file named SCRIPT.TXT containing all of these keystrokes (ten in all). Next, I tried running SETZPC with redirected input:

```
SETZPC < SCRIPT.TXT
```

The program started executing automatically, drawing its keyboard input from the script file, until it reached the last question which required a keyboard response. Then it hung. Nothing could be done except to reboot. The only thing I could figure was wrong, was that my script file didn't contain enough characters. So I stepped through the program once again, and noted my keystrokes. Everything seemed to be okay. What's going on here? Well, it took me a long time to figure this out the first time it happened. To make a long story short, the problem has to do with those peculiar escape sequences we have been talking about.

One of the things SETZPC does when it first starts up is check to see if it is running on a Z-100 computer. (Actually, it is checking to see if the computer is in Z-100 mode or PC mode of ZPC). The way it does this (yep, you guessed it) is by using one of those "identify terminal type" escape sequences. The one it uses is ESC Z (identify as VT52). After SETZPC sends ESC Z to the console, it then expects to receive a character back from the console if Z-100 mode is in effect. Technically speaking, the Z-100 would send back ESC / K, but SETZPC doesn't care what it receives. Anything at all coming back from the console is considered to be fair notice that the host computer is in Z-100 mode.

So what effect does this have on our redirected input experiment? Our script file contains 10 characters, which are presumeably the responses to the questions asked by the SETZPC program. But when SETZPC attempts to read a character from the console after sending the ESC Z command, it reads one from our script file instead, since the input is being redirected. The result is that our script file comes up one character short at the end.

The solution? Our script file should contain the character(s) that SETZPC expects to see when it issues the ESC Z escape sequence. In other words, our script file should include a dummy character at the very beginning, to fool SETZPC into thinking the console is transmitting the character. Then everything will work okay. Try it and see.

Ignore the Garbage

We have now solved our problem of using SETZPC with redirected input. But there is one other small thing you will notice. After the SETZPC program is run using this technique, the letter 'K' will be displayed at the DOS prompt. Where did that come from?

Well, when the SETZPC program issued the ESC Z command the console wanted to reply with ESC / K. But since the program was taking its input from the script file, the characters transmitted by the console weren't being received. As soon as the SETZPC program was done, input was directed back to the console, and the transmitted characters popped out at the DOS prompt. The ESC / didn't print because they were considered to be non-printable characters. But the 'K' was displayed.

Which Programs Do, and Which Don't?

Good question. That is, "How do you know if a program uses one of these odd escape sequences that will goof up your redirected input attempts?". The answer is to use redirected output. (Those of you who are already lost in this discussion, will be raving maniacs by the time I'm through!).

As an example, try running SETZPC using redirected output . . . like this:

```
SETZPC > OUTPUT.TXT
```

As soon as the first screen comes up, you can just terminate the program using Control-C. Now look at the OUTPUT.TXT file with an editor like DEBUG. One of the things you'll find is the infamous ESC Z sequence, which tells you that the program expected some characters to be transmitted back from the console. Therefore, you know that you need to include those characters in your script file in order to use redirected input with that program.

For another example, you might want to try running a compiled ZBASIC program using redirected output. When you check the output file, you'll find that the first thing sent to the console by the compiled program is ESC i 0. The compiler apparently uses this to determine the color video status of the Z-100. This should tell you that your script file for compiled ZBASIC programs needs to be prefaced with the characters:

Continued on Page 48

```
*  This script will log you on to the Heath Users' Group Bulletin Board
*  and capture the current online Bargain Centre listing, as well as the
*  current message base.  These are all stored in C:\COMM\HUGBBS.LOG.
*  Insert debugging code here - the ECHO statement
   ECHO
   WAITFOR "Enter Your FIRST Name"
   SEND "FIRSTNAME"
   WAITFOR "Enter Your LAST Name"
   SEND "LASTNAME"
   WAITFOR "Enter Your HUG ID Number"
   SEND "998877"
*  Get the Online Bargain Centre List.
   WAITFOR "Function or <H>elp"
   SEND "OL";
   DOWNLOAD ASCII "C:\COMM\HUGBBS.BC"
*  Receive the message base using ASCII protocol in a continuous stream.
*  Scanning from message 1 will always start at the lowest-numbered
*  message in the message base.
   WAITFOR "Function or <H>elp"
   SEND "RC";
   WAITFOR "Scan From Which Message"
   SEND "1";
   DOWNLOAD ASCII "C:\COMM\HUGBBS.LOG"
   SEND "^M";
   WAITFOR "Function or <H>elp"
   SEND "G";
   WAITFOR "Leave Private Message"
   SEND "N";
   HANGUP
```

Figure 5

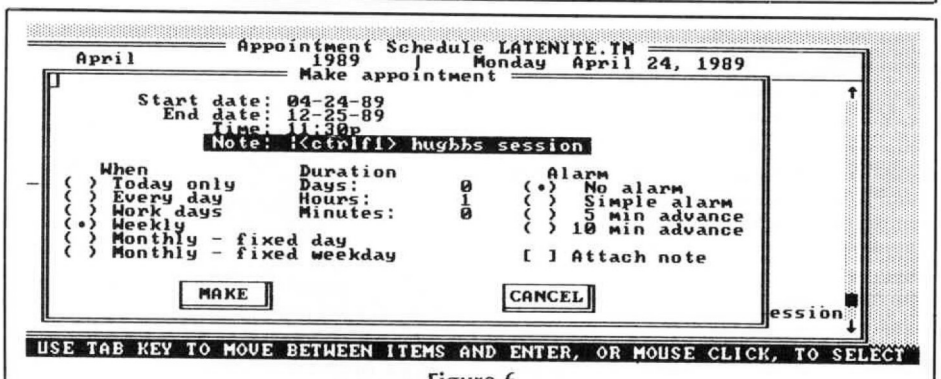


Figure 6

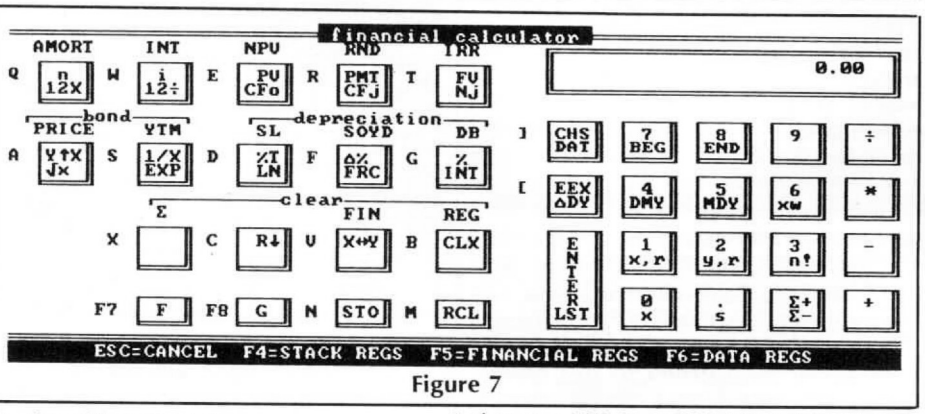


Figure 7

Continued from Page 41

`ESC i E <pov> <vrs>`

(See the description earlier in this column for the meanings of these characters.)

And don't forget . . . this same kind of problem using redirected input can be caused by any of the "odd" escape sequences we have talked about. Generally, the problems associated with programs that use ESC Z or ESC i 0 to identify the terminal type are pretty easy to overcome. But if a program uses ESC n (cursor position report), or any of the transmit character escape sequences, it usually won't be practical to use redirected input with it, since it would be difficult to tell in advance exactly what input the program was expecting.

Q&A, Where Art Thou?

savings account (your mileage may vary considerably!). Using the manual's example on "Solving IRA and savings account problems" as a guide, I found that if I start saving $218.64 per month, I'll reach $40,000 after 120 months. The only trouble is, I have three other daughters; I suspect I'll have to write a lot more articles in the future!

Using the Utilities

There's one last application provided with PC Tools Desktop that provides four useful utilities. Using this last menu selection, you can change the hot-keys assigned to pop up the Desktop itself or activate the Clipboard cut and paste facilities when you're running other programs. You can also display a complete ASCII table, change the menu and window colors used by the Desktop system (as I mentioned earlier), or unload PC Tools Desktop from memory. Addition of these functions rounds out an impressive collection of programs and gives you the control you need to keep PC Tools Desktop from interfering with other programs that share the use of your microcomputer.

Wrapping Up

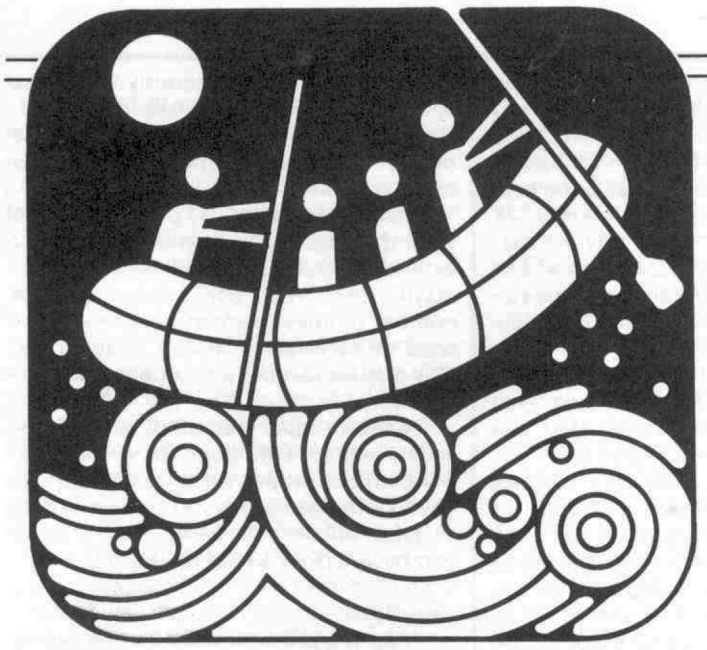
It's been a long journey through the programs that make up the PC Tools Deluxe Version 5 Desktop Manager. Hopefully, you will have found the information you need to make a decision on this package, especially in light of the PC Shell and PC Format coverage in the first article of this series.

But there's a few more programs to cover before we finish with PC Tools Deluxe Version 5, so you may want to stay tuned. Next time, I'll cover the PC Cache disk caching program, PC Secure (the file encryption/compression/decryption program), PC Backup (the quick floppy and hard disk backup system), and the hard disk utilities Mirror and Compress. If you have any questions about anything I've presented, please drop me a note and I'll try to respond promptly. The Technical Support folks at Central Point Software are very helpful, and are also quite willing to help you work out any problems or bugs you seem to be encountering. See you next time! ✱

I know I've been promising another Question and Answer session for several columns now. And the questions are beginning to pile up. Don't worry — everybody gets a personal reply ASAP. But the purpose of the Q&A section is to spread the knowledge around a bit, so the same questions don't get asked over and over.

Well guess what? We're out of space again in this issue. I guess I just don't know when to shut up. But I'll tell you what. The next installment of "Z-100 Survival Kit" will be devoted entirely to answering some of your more interesting questions. I promise!

'till then, keep in touch! ✱



#8

Z-100

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

SURVIVAL KIT

Q&A

For the last few issues, I haven't had room for a question and answer section in the "Survival Kit" column, and I keep promising that the next issue will. Hopefully, this installment will help get us caught up. The whole column this month is devoted to answering some of the more interesting questions that I have received.

Question:

I miss the 'plus' key on my numeric keypad. Is there any way to make the Z-100's ENTER key into a plus (+) key?

Answer:

Yes, the Z-100 ENTER key can be mapped so that it is a plus (+) key. This is one of the nicest things about the Z-100; its versatility. There is a program named FONT.EXE that comes with version 2.x or 3.x of MS-DOS for the Z-100. You can use this program to remap the keyboard codes, as well as for changing the font designs.

To make the ENTER key perform as if it were a plus (+) key, follow this procedure . . .

A. Invoke the FONT program, by typing 'FONT' at the DOS prompt.
B. Select the Key Map Editor from the main FONT menu.
C. Switch to page two of the key map by hitting the F1 function key.
D. Select to change the code generated by the ENTER key by hitting F2, then typing 8D, followed by a RETURN.
E. Map the key to the plus (+) key by entering 2B as the new code, followed by a RETURN (2B is the ASCII code for '+').
F. Exit the Key Map Editor by hitting the F3 key.
G. Select to write the new font file to disk. You might use a name like PLUS.FNT.
H. Exit the FONT.EXE program.

Now you have a font called PLUS .FNT that considers the ENTER key to be the same as the plus (+) key. In order to load this font, simply type;

```
FONT PLUS.FNT
```

You might want to include this command in your AUTOEXEC.BAT file if you want the modified key mapping to be in effect all the time.

One important consideration . . . some programs can't be used without the ENTER key. For instance, WatchWord uses the ENTER key to switch back and forth from insert to command mode. When you create your special key mapped font, you may want to assign the ENTER key function to another less-used key combination. A good choice might be to use the Shifted ENTER key (code CD) as the ENTER key (in other words, map CD to 8D in the Key Map).

Question:

I would like to speed up my Z-100 to 8 MHz, but I've heard that some Z-100s won't run at the faster speed. Which one of the speed upgrades should I use?

Answer:

Speeding up the Z-100 to 8 MHz can sometimes cause problems. The safest (and most expensive way) to do it is to use the Heath/Zenith HA-108 kit (see detailed description in July 1985 REMark Magazine). The HA-108 kit includes all the parts necessary to properly speed up the Z-100, and convert the motherboard to 256K memory chips. Alternatively, (and less costly) several vendors offer speed-up kits which simply include the parts to speed up the system clock. The difference between these bargain priced kits, and Heath's HA-108 speed-up is that Heath's kit comes with faster versions of most speed critical ICs.

I have installed a speed-up kit from C.D.R. Systems Inc. in my Z-100, and have had no problems at all. But I may have been lucky. Many users report that they have to switch the main 8088 processor, I/O chips, and various other components in order to get their Z-100's operating at the faster speed. The manual that accompanied the C.D.R. speed-up kit was very thorough, and includes a complete troubleshooting section which lists probable slow chips, in the order they are most likely to fail.

Question:

I would like to upgrade my Z-100's operating system to version 3.1 of MS-DOS, but how will I convert all of my present floppy disks (formatted with Z-DOS) to the new system?

Answer:

Any new version of MS-DOS should be able to read file formats of previous versions. For example, MS-DOS 2.x or 3.x can read the old Z-DOS 1.1 format without any problems. After upgrading to the new version, you should format new floppies, and copy old data and programs to the new format disks.

Question:

Is there any way to use Zenith's FTM (flat tension mask) monitor with my Z-100.

Answer:

Sorry, but the news is bad. I don't know of any way to use the FTM monitor with a Z-100. The problem is that the

ZCM-1490 is a fixed frequency monitor, which uses a 31kHz video frequency. The output from the Z-100 is 15.75kHz. This means that the Zenith FTM monitor will not be able to sync on the Z-100's video frequency.

It would be nice if Zenith would make the Flat Tension Mask technology available in an autosynchronizing type of monitor (similar to NEC's MultiSync monitor). This way it could be used on virtually any computer, with any graphics card. But until that time comes, I don't know of any way to use it on a Z-100 computer.

Question:

I've heard that PC clones can be speeded up (especially screen I/O) by using a faster version of the ROM, which is copied into RAM memory. Can this be done on the Z-100?

Answer:

Writing a modified version of the Monitor ROM into an area of RAM may be possible on the Z-100, but there isn't much reason for doing it. I have examined the MTR-100 monitor ROM source code in some detail, and quite frankly, I don't think it can be speeded up by any significant degree. The original programmer was pretty sharp.

The majority of code in the IBM-PC ROM is actually the BIOS routines. It is the BIOS interrupt routines which are helped most by modifying the code for faster operation. The BIOS of the Z-100 is not held in ROM, but is provided on disk, as a software program. The Programmer's Utility Pack contains the complete source code for the Z-100 BIOS, and may be modified, and re-assembled to your heart's content. It is quite possible that the Z-100 BIOS could stand some fine tuning to increase speed, but beware; this type of thing should only be attempted by experienced systems programmers.

Question:

I have tried several programs that use interlace mode on the Z-100, but the screen seems to wrap around from the bottom to the top. In other words, some of the lines that are supposed to be on the bottom of the screen, appear at the top, and some of the text lines are split. I have 64K RAM chips installed on the video board. What's the problem here?

Answer:

It sounds like you have not configured the video board to use 64K RAM chips. Sure, you may have 64K chips installed, but you must also set jumper J307 on the video board to indicate that 64K chips are being used. There are three possible settings for jumper J307; Low 32K, High 32K, or 64K. These choices deserve some explanation.

The video logic board in the Z-100 was designed to use 64K chips. However, to decrease costs, Heath/Zenith allowed 'bad' 64K chips to be used as 32K chips. Generally, the faulty 64K chips would only have a few bad cells in them. So if you could find a set of 24 chips that had all good cells on the bottom, or all good cells on the top, it was okay to use them as 32K memory. Most of the "All-in-One" Z-100's came with one bank of good 64K RAM chips installed for monochrome operation. But most of the earlier low-profile models came with 'faulty' 64K chips jumpered to be used as 32K chips. Depending on whether the chips were good on the top or bottom, jumper J307 was set for either low 32K or high 32K.

Whenever you switch to good 64K chips in the video, you must also reflect this change by changing jumper J307. You may want to refer to the Z-100 Technical Manual for more information about the jumper settings. You may also want to note that you can put jumper J307 in the 64K position, even if you only have the 'faulty' 64K chips provided by the factory. This will allow you to use programs that operate in interlace mode, or use two pages of video memory. However, doing this with 'faulty' 64K chips will result in a few unwanted dots of color on the screen.

Question:

I have a Z-100 low profile model (ZW-110) which has three planes of 32K video memory chips. Everyone says I should replace the 32K video memory chips with the larger 64K chips. It won't cost much for the parts, but I don't like to take the computer apart, except for a good reason. What will I gain by putting in the 64K chips?

Answer:

Using 64K video memory chips provides more than twice as much memory as is required for the standard Z-100 display. But you must use a program which knows how to take advantage of the extra memory in order to get any benefit from the 64K chips. Most programs don't require this overabundance of memory, so you won't be able to tell any difference in their operation.

I can think of three reasons why a program would require that 64K chips be installed. First of all, they allow the Z-100 to display a higher resolution image. The standard Z-100 video provides a resolution of 640 × 225 pixels. Higher resolution modes, such as those available using interlace or ProScan, may have resolutions up to 640 × 512 pixels. In order to display these higher resolutions, more than 32K of memory is required to store the bit-mapped image.

Another advantage for having 64K memory chips is so the program can have two 'pages' of video memory. This allows the program to compose a page of text or graphics out of sight of the viewer. Then when the page is done, it can be displayed almost instantaneously simply by changing the CRT controller's start address register. This page switching scheme makes slide-show type presentations more attractive.

Another reason a program might want the large video memory chips is for extended scrolling capabilities when displaying text. With 64K RAM chips, the number of lines that can be scrolled forward or backward increases dramatically. This may be useful for text processing applications.

Keep in mind that all of these interesting uses for 64K video RAM chips depend on the proper software for support. Most of the programs you use (including all DOS utilities) probably don't care if you have 32K or 64K chips installed.

Question:

I have a printer, mouse, and modem, that all connect to the serial port of my Z-100. Most of my software that supports these devices will allow me to use them on either serial port A (J1) or serial port B (J2). Other than the obvious difference in gender, what is the difference between these two ports?

Answer:

Actually, there is quite a lot of difference between the two serial I/O ports on the Z-100, considering the fact that they both use identical 2661 interface chips. Most of the differences can be traced to simply pinout differences, but some of the RS-232 lines also have a different logic design.

One of the ports is supposed to be a DTE port (that stands for Data Terminal Equipment), and the other port is to be used as a DCE port (Data Communication Equipment). The DCE port (J1) is intended as a printer port, and the DTE port (J2) is for a modem. That's the way it's advertised to work, but it's not the way it always comes out in real life. For instance, it seems like most serial printers would rather compete for J2 with your modem.

Actually, for most RS-232 devices, the differences between the two ports can be corrected by something called a modem adaptor with full handshake. Or you can build your own adaptor cable with this wiring:

```
1        <-------->      1
2        <-------->      3
3        <-------->      2
4        <-------->      5
5        <-------->      4
6        <-------->      8, 20
7        <-------->      7
8, 20    <-------->      6
```

Notice that it doesn't matter which end is the right end, since the cable is symetrical. (It looks the same from either end.) In order to use it as an adaptor, though, you'll want a female DB-25 on one end and a male on the other.

Question:

I would like to use my Z-100 and a graphics editor program to compose title frames for my home video movies (to be played back on a VCR). I haven't been able to get very good results trying to take a picture of the screen directly with my video camera. Do you have any good ideas?

Answer:

The obvious answer is simply to connect your VCR to the composite video output of the Z-100. The composite video output of the Z-100 is the same RS-170 video signal that your camera generates. (Conversely, the color video monitor you use with the Z-100, makes an excellent video monitor for your camera/VCR setup.)

To do this, simply design the screen that you want with your favorite graphics editor. Then connect the composite video output of the computer to the input of your VCR. Record for a few seconds (however long you want the screen to be displayed) and then stop the recorder. That's it.

The disadvantage to this method is that the composite video output of the Z-100 is only a monochrome signal. So your title frames won't be able to take advantage of all the Z-100's colors. If you absolutely have to have color, there are devices available which will convert a digital RGB signal to color composite video, but they are fairly expensive (in the $200 to $300 range).

Question:

I just purchased a used Z-100, and being a late arrival, have lots of catching up to do. I want to learn all I can about the Z-100. Can you give me a fairly complete list of sources of information about the Z-100 computer?

Answer:

Okay, I'll try. I appreciate a person who has the initiative to try and become an expert through home study. Here is a list of reference materials which contain Z-100 information:

1. *Z-100 User's Manual.* You should have received this loose-leaf manual when you purchased your Z-100. It contains general user information about the operating system, BASIC programming, and most of the information necessary for a novice to program the computer. No price is available, since I don't believe this manual is sold seperately. Published by Zenith Data Systems.
2. *Z-100 Technical Manual.* This is a three volume set which includes detailed descriptions of the Z-100 hardware (with schematics), manufacturer's spec sheets on all programmable IC devices, and a listing of the MTR-100 monitor ROM program. Every serious Z-100 programmer must have this manual. $50.00 (but may not be available any longer). Published by Zenith Data Systems. Available (maybe) through the Heath Catalog Order Center Phone (800) 253-0570
3. *Programmer's Utility Pack.* A loose leaf manual and disk set. Contains descriptions of all MS-DOS function calls, and the Z-100 BIOS functions. Describes EXE and COM program development, and how to write DOS device drivers. Many useful programmer's utility programs are included on disk, as well as complete source code for the Z-100's BIOS. A must for every Z-100 programmer's shelf. $150.00. Published by Zenith Data Systems. Available (maybe) through the Heath Catalog Order Center Phone (800) 253-0570
4. *How To Use Zenith/Heath Computers* by Hal Glatzer. Softbound book (144 pages) giving entry level information for users of Heath/Zenith H-8, H-89, and Z-100 computers. $19.95.
   Copyright 1982 by:
   S-A Design Publishing Company
   515 W. Lambert, Building E
   Brea, CA 92621-3991
5. *Z-100 Software Directory.* A humongous, but outdated listing of software that works with the Z-100. I'm sure this loose leaf book is out of print by now. You'll need to find a used copy. Original cost $30.00.
6. *Z-100 Service Literature.* This stuff is not generally available to the public — only to Zenith Data Systems Service Centers. Most of the programming information included with this stuff is available in the Z-100 Technical Manual. But the service literature contains in-depth "theory of operation" descriptions, as well as parts lists, troubleshooting information, and field service bulletins. Since the Z-100 is obsolete merchandise now, you might try checking with a local ZDS dealer to see if they will give you their Z-100 service literature, instead of throwing it in the trash. There are several thousand pages of it, so making photocopies wouldn't be practical.

In addition to the reference works listed above, there are a few periodical type publications which might be of interest to Z-100 owners:

1. *REMark Magazine.* Includes this column (Hip, Hip, Hurray!) and other occasional articles specific to the Z-100. If you're serious, try to get all the back issues since July 1982. (That's the first issue that mentions the Z-100). $22.95 initial/$19.95 renewal per year, includes:
   - membership in the Heath Users' Group.
   - Published 12 times per year by:
     Heath Users' Group
     P.O. Box 217
     Benton Harbor, MI 49022
     Phone 616-982-3838
2. *SEXTANT Magazine.* An independent magazine (not related to Heath/Zenith) which generally has one or two Z-100 specific articles in each issue. Z-100 articles begin in issue #2 (Summer 1982). (No longer in existence.) Sextant Publishing Company.
3. *Z-100 LifeLine Journal.* 16-20 pages devoted exclusively to the Z-100. Can't plug this too much, since my company publishes it.
   $24.00 per year.
   Published 6 times per year by:
   Paul F. Herman Inc.
   3620 Amazon Drive
   New Port Richey, FL 34655
   (800) 346-2152
4. *H-Scoop Newsletter.* Zenith corporate news and product announcements, along with product reviews and Heath/Zenith scuttlebutt. A little light on Z-100 specific information these days. $24.00 per year.
   Published 12 times per year by:
   H-Scoop/Quikdata
   2618 Penn Circle
   Sheboygan, WI 53081-4250
   (414) 452-4172
5. *BUSS Newsletter.* Zenith corporate news and product announcements. Consists mostly of contributions by readers. Occasional, but rare Z-100 specific topics. (No longer in existence.) Sextant Publishing Company.
6. *BUSS Directory.* Contains a list of Heath/Zenith vendors, local Heath Users' Groups, and the most complete Heath/Zenith periodical index around (which includes listings for REMark, SEXTANT, and BUSS). Last edition was 1987-88. (No longer in existence.) Sextant Publishing Company.

Question:

I would like to buy either a 'C' or Pascal compiler. Can you tell me which ones will run okay on my Z-100 computer?

Answer:

As far as I know, any compiler you care to purchase will run on the Z-100. Some of the more recent offerings include a graphics windowing type of user interface which will not run directly on the Z-100 (like Quick-C or Turbo Pascal). But even these compiler packages should include a command line version of the compiler which can be used on a 'generic MS-DOS' machine.

One thing these compilers will not include, however, is a graphics library that can be used with the Z-100. And some of their standard screen control functions (like clear screen) may also cause problems when used on the Z-100. You can get around this problem by investing in one of the Z-100 graphics libraries which are available, or you may want to write your own Z-100 function library.

Question:

I have version 2.11 of MS-DOS for the

Z-100, and am considering upgrading to version 3. But Heath/Zenith wants $150 for the new version, which seems a little steep. Are there any compelling reasons to upgrade from version 2 to version 3 of MS-DOS?

Answer:

I used to answer this question with an immediate yes! After all, the operating system is the most important piece of software you own, and it should be kept up to date. But in recent times, I have had to reconsider this attitude in a new light, regarding the Z-100.

First of all, there isn't really that much difference between MS-DOS version 2.x and 3.x. Specifically, version 3 of MS-DOS introduced the following new features:

1. AT style 1.2 Mb floppy drive support.
2. Direct control of print spooler by application software.
3. Expanded international character and keyboard support.
4. Extended DOS function error reporting.
5. Support for networked applications, including file and record locking.
6. Support for larger hard disks (greater than 32 Mb).

But of these new version 3 features, the Z-100 implementation does not include the 1.2 Mb floppy drive support. And the Z-100 has always been able to use larger hard disks, up to 64 Mb. Direct control of the print spooler is nice, but it requires appropriate application software, which is not available for the Z-100, to my knowledge. And besides, this feature seems to be present in version 2 of DOS, too — it was just documented when version 3 was released. Expanded international support would only be useful for persons living outside the United States. And extended DOS error reporting will only be interesting to programmers, who will probably choose not to take advantage of it, since doing so would make their programs version 3 dependent.

The only new features of version 3 that are left with any merit are the networking features. Are you using your Z-100 as a part of a network, or multi-user system? Probably not. Okay, I think we can agree that version 3 of MS-DOS doesn't offer much significant improvement over version 2.

Another reason put forth for upgrading the operating system is to insure that further updates will be available. In other words, what if version 4 of DOS is released, and you haven't upgraded to version 3 yet? This argument doesn't hold water — for two reasons.

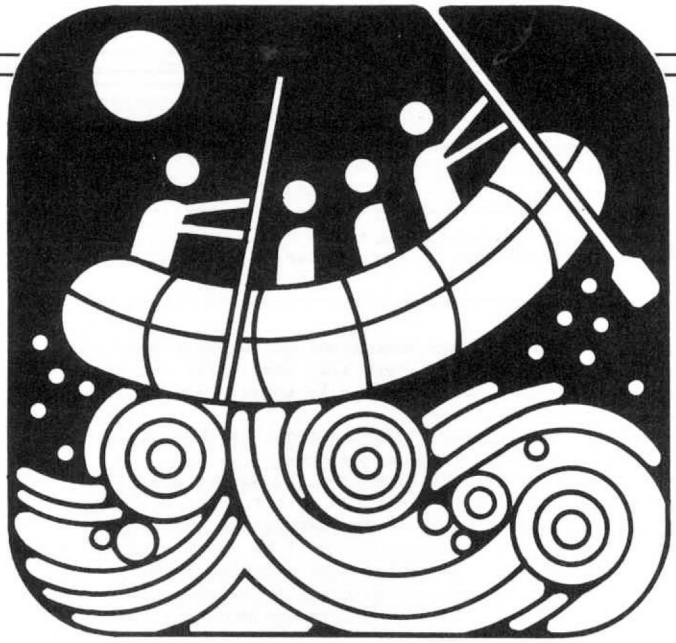
First off, Z-100 owners needn't worry about staying current any longer. There will never be a version 3.3 for the Z-100, let alone a version 4. So if you are destined to be forever out of date, you might as well stick with version 2.

Secondly, Microsoft (or Zenith) doesn't seem to have any upgrade policy when it comes to DOS. When a new version is released, you get to buy it all over again for the list price. Zenith offered a discount coupon to owners of version 2 of MS-DOS some time ago, but that offer has reportedly expired. If you didn't upgrade to version 3 before the coupon expired (or if you didn't receive the coupon), you're out of luck. You'll have to dish out $150 for version 3.

Conclusion? If you have $150 burning a whole in your pocket, buy MS-DOS version 3. Otherwise, forget it — it's not worth it!

IMPORTANT NOTE: The points made above only apply to the differences between MS-DOS version 2 and version 3, which are minor. If you are still using Z-DOS (MS-DOS version 1.x), you definitely need to byte the bullet and upgrade to version 2 or greater. The differences between version 1 of MS-DOS, and version 2 are staggering — it's almost like a different operating system. Most programs written these days are designed for version 2 or above, and may not run correctly using version 1.

Keep in touch! *



#9 Z-100

SURVIVAL KIT

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

ZPC Revisited — By Popular Demand

In the very first installment of this column I stated that I didn't want to get too deeply involved in describing ZPC patch information. But that was before the letters began coming in. My mail has been running about 10 to 1 in favor of continued support for ZPC patch information for new PC programs.

There are several factors at work here. First of all, Pat Swayne's ZPC Update column has virtually disappeared from the pages of REMark. For several years, Pat kept us abreast of the patches required to run the latest PC software under ZPC. Occassionally, another ZPC update article filters down to us, but patch information for new programs, and new releases of old programs, are beginning to get away from us. Many of you are continuing to discover how to run your PC programs under ZPC, but there isn't a public forum for exchange of that information.

Even though Pat has been too busy to continue the ZPC Update tradition, he has silently been at work trying to keep track of information people send to him. He has periodically added new patch information to the PATCHER.DAT file on the ZPC Upgrade Disk (HUG #885-3042-37), and has tried to keep track of bug reports and fixes to the ZPC program itself, that users have sent to him. I too, have received quite a bit of patch information for running PC software under ZPC.

By popular demand, I have talked with Pat Swayne and offered to carry the torch for ZPC patch information into the indefinite future. But now the question is this; how should all this information be collected, organized, and distributed to the Z-100 community?

Must We Choose Sides?

At this point, I can't help but put in my two cents worth about the subject at hand. It seems like whenever I mention running ZPC on the Z-100, I get a lot of angry letters pro and con, so I might as well stir things up real good. My associations with Z-100 users lead me to believe that there are three distinct attitudes that prevail regarding the use of ZPC.

First, there are those who spend every spare moment in the trivial pursuit of PC compatibility. Every public domain, shareware, or commercial PC program they can get their hands on presents a new challenge, whether the program has any useful features or not. These people are only interested in the "conquest". Once they figure out how to run the program under ZPC, they will probably lose all interest. There is some hacker spirit here. This isn't a quest for practical use of the Z-100; it is a hobby. We have these people to thank for much of the ZPC patch information that has already been compiled.

At the other end of the scale are the Z-100 purists. These are the people who insist that PC emulation represents a degradation of their Z-100. Even though they admit that ZPC is a dandy program, it is for others to use, not them. Most of the purists believe that a Z-100 can do anything better than an IBM-PC can. Not coincidentally, this group consists mostly of programmers who are able to make the Z-100 do what they need it to do.

The mainstream Z-100 user is somewhere in between these two extremist groups. He uses ZPC when he needs to, but prefers to use native Z-100 software when available. This is obviously the most well-rounded approach.

I would consider myself to be somewhere near the top of "mainstream", and occasionally slipping into the "purist" camp. Actually, my "purist" affiliation is not so much because I dislike PC emulation. It has more to do with the fact that I have an AT clone sitting next to my Z-100. So if I want to run PC software I just run it on the PC. It seems silly to go to the trouble of emulating it on the Z-100, under the circumstances.

But I realize that many of you don't have a second computer (perhaps because your spouse would raise an eyebrow). And I certainly wouldn't want to suggest that you should out-and-out trade your Z-100 for a lowly PC compatible clone. That would probably be a mistake, unless you can afford to go for an 80286 or 80386 machine. So that brings us back to the subject at hand — ZPC emulation of your favorite PC software.

ZPC Patch Information — Some Problems

As most of you know, the key to success in using ZPC, many times depends on developing patches for the PC software. Sure, some programs will run straight up under ZPC, but many will not. The problem of patching software to run under ZPC is something that is unavoidable. Some things just can't be emulated with software, and therefore, the program must be changed.

The HUG User's Manual that comes with the ZPC Emulator is an excellent source of information about how ZPC works, and about the type of problems encountered in PC software which need to be patched. I expect that many of the

questions Pat and I receive regarding how to make patches to programs could be avoided if everyone would simply read the manual!

Being an ingenious lot, many Z-100 owners have read the instructions, and have mastered the techniques of applying patches to programs so they can run under ZPC. Information about many of the patches you have developed have been sent to HUG, and to me . . . elaborately described in lengthy letters, or scrawled on the back of an envelope.

But there are several problems with the present scheme (or lack thereof) for submitting patch information. The biggest problem is trying to make sure that the file that is being patched is, in fact, the proper version of the program. Even though the present PATCHER program reports "Your Program was successfully patched", it really doesn't have any way of knowing whether the patches were applied correctly.

Sometimes, hints about using the program under ZPC are not available along with the patch information. For example, the program may need to be run in PC mode 7, as well as be patched. It would also be nice to have some information about the logic of the patches which were developed, to help other users find patches to future versions.

And lastly, it would be nice to have a list of the software that didn't require patches, along with any special instructions for its use. A data file containing patch information is nice to have, but software titles not included in the patch list leave a question in the user's mind. Does it work okay without patches, or is it just that patches have not yet been developed?

Let's Get Organized

What I have in mind is a single all-encompassing data file which would contain all the information you might need to know about patching programs for ZPC. Something similar to the present PATCHER.DAT, but bigger and better (the new file will be named PATCHIT.DAT). This file will be accompanied by a new utility, named PATCHIT.EXE, which will be used to patch programs, and provide information about their use under ZPC. I am beginning development on PATCHIT.EXE now (I'm writing this in August), and will be compiling all the patch information I have at present. By the time this column is printed, everything should be ready to go.

I am depending on your support to make this project a success for all of us. Continue to send in information about patches you have developed, or programs you have found to work without patching. The new PATCHIT program has the capability to do patch verification, and give user information about the patched program. But in order to provide this type of support to ZPC users, patch developers (that's you!) will need to provide more complete information when patches are submitted.

```
------- New Entry Field -------------------------------------------------
*
        Any line beginning with an asterisk '*' signifies the start of a
        new PATCHIT.DAT File entry.

------- Program Name Field ----------------------------------------------
N textstring
        where "textstring" is the name of the program.

------- Requirements Field ----------------------------------------------
R [N][P][Z][C]
        where N,P,Z,C are single letters indicating any requirements for
        proper operation of the program.  More than one letter requirement
        may be listed, except when requirement N is shown.  The letters
        indicate the following requirements;
                N = none, program works without patches or ZHS support.
                P = program requires software patches.
                Z = program requires ZHS circuitry for proper operation.
                C = program requires PC style COM port for proper operation.
        Note that when it is questionable as to whether the ZHS support
        circuitry is required, the Z requirement will be shown until proven
        unnecessary.  Users who run the programs on a ZHS equipped Z-100 may
        not be able to tell if the program would run on an unmodified system,
        therefore they should show requirement Z if any doubt exists.


If the Requirements Field 'R' contains requirement P, the following fields
must be present;

------- Filename Field --------------------------------------------------
F filename
        where "filename" is the name of the file to be patched.  If more than
        one file needs to be patched for this application, each file to be
        patched must be declared with an F field line.

------- Patch Field -----------------------------------------------------
& hexaddress s1,s2,s3,...sn > d1,d2,d3,...dn
        where;
                hexaddress      = a hexadecimal address of up to five digits.
                                  This is the file offset where the bytes to
                                  be patched are located.
                s1,s2,s3,...sn  = two digit hexadecimal values of the original
                                  bytes to be patched.  PATCHIT uses these
                                  values as an integrity check, and to
                                  determine if the file has already been
                                  patched.  These values are optional.  If
                                  original byte values are not known, the
                                  entire s1...sn series may be replaced by a
                                  single question mark '?'.
                d1,d2,d3,...dn  = two digit hexadecimal values of the patches
                                  to be applied at the specified offset.

        Each '&' field line specifies a series of contiguous bytes to be
        patched.  If more than one area of the file must be patched (which
        is likely), multiple '&' field lines must be used.  In other words,
        each 'F' field line may be followed by as many '&' field lines as
        are necessary to provide the required patch information.
```

Figure 1
Required PATCHIT.DAT Fields

Following is a detailed description of the format which will be used in the PATCHIT.DAT file. Your program patch submissions may be provided in this format, or you may simply submit the required information, and I will make up the file entry.

PATCHIT.DAT File Format

Each program entry in the PATCHIT .DAT file will have several fields describing the program or file, and giving information about any patches that may be required. Each field will be described on a separate line of the file (in other words, the fields are separated by a CRLF). Some of the fields are required fields, and must be present in order for PATCHIT.EXE to patch a program. Other fields are optional, and may be present to give information about the program or file, and to help insure that patches are applied correctly.

Each field is designated by a single character in the first column position of a line, followed by a space. Lines are limited to 80 characters in length.

Required fields for each PATCHIT .DAT file entry are shown in Figure 1. Notice that a valid PATCHIT.DAT file entry could be composed of only the informa-

```
------- Publisher Field -----------------------------------------------------
P textstring
        where "textstring" is the name of the program's author or publisher

------- Version Field -------------------------------------------------------
V textstring
        where "textstring" provides version or release information about
        the program.

------- Summary Field -------------------------------------------------------
S textstring
        where "textstring" is a brief description of the program.

------- Instruction Field ---------------------------------------------------
I textstring
        where "textstring" provides instructions or other information which
        should be displayed on the screen after patches have been successfully
        applied to the file(s).

------- Contributor Field ---------------------------------------------------
C textstring
        where "textstring" is the name of the person, company, etc.
        responsible for providing information about this entry in the
        PATCHIT.DAT File.
```

Figure 2
Optional Fields for Each Program Entry

```
------- File Date Field -----------------------------------------------------
D filedate
        where "filedate" is the date of the file to be patched.  This is the
        DOS file creation date shown by a disk directory listing.  The date
        should be in the form mm/dd/yy or mm-dd-yy.

------- Check Field ---------------------------------------------------------
? hexaddress s1,s2,s3,...sn
        where "hexaddress" is a hexadecimal address of up to five digits
        which gives an offset into the file.  The series s1...sn are
        contiguous byte values which should match the bytes at the specified
        offset in the source file.  PATCHIT uses the values given in the
        Check Field to help determine if this is the correct file to patch.
        Note that the patch field 'P' also allows for byte verification.
        The check field is provided for more thorough checking, where
        necessary.

------- Comment Field -------------------------------------------------------
' textstring
        where "textstring" may be any comment.  PATCHIT ignores comment
        fields.  They are provided so that patch logic may be documented.
        PATCHIT will also ignore any line which is totally blank, or begins
        with a white space character (space or tab).  Therefore, if you
        prefer, comments in the PATCHIT.DAT File may begin with a space or
        tab character.
```

Figure 3
Optional Fields for Each File to be Patched

tion available in a present PATCHER.DAT file (that is, file name and bytes to patch). And, in fact, most of the entries in our first PATCHIT.DAT file will have only basic information, since the extended information will not have been provided. So we'll need more information about many existing entries, as well as information about new programs. Hopefully, many of you who have provided patch information in the past will help fill in the missing information for users who will need it.

The PATCHIT program has the capability of using optional information, if it is provided. The fields described in Figure 2 are optional, and if included, should immediately follow the program name 'N' field. The fields shown in Figure 3 are optional, and if included, should immediately follow the Filename 'F' field for each file to be patched.

Figure 4 gives a brief summary of all the different PATCHIT.DAT fields, and shows which ones are required. The reason that only a few fields are required is so that the existing data base of ZPC patches (from HUG's PATCHER.DAT) could be utilized. When providing new submissions, the information should be as complete as possible.

Figure 5 shows a couple of examples of how the PATCHIT.DAT fields can be used to provide information for a program entry.

Suggestions for Contributors

When you contribute information about a program patch (or a program that does not require patches), provide as much information as possible. After all, most of the information requested by the PATCHIT.DAT fields will be readily available to you while you are researching the patches. Of particular importance is the version of the program, the file date of any files that need patching, and the original values of bytes which are patched. PATCHIT.EXE will use this information to verify that a correct patch has been applied.

If no original values for patched bytes are provided, PATCHIT.EXE will simply display the message "Program has been patched". But if the PATCHIT.DAT file contains original values (in the Patch field '&' or Check field '?'), it may report "Program has been patched successfully".

You may be as verbose as you like with your comments. I will edit them, if necessary, to keep the PATCHIT.DAT file size reasonable.

Some programs may warrant more than one entry in the PATCHIT.DAT file. For example, if the program will run without patches with the ZHS circuit, but requires patches to run on a plain Z-100, then a separate entry should be made for each case. One entry would not contain any patch information, but would inform the user (use the 'R' field) that the ZHS circuit was required. The other entry would show that the ZHS circuit was not required, and include the appropriate patch information.

Spreading the News

Now that we have a specification for submitting ZPC patch information, and a

```
* New entry                                                        Required
N Program Name                                                    Required
P Publisher's Name
S Summary of program
V Version
I Instructions
R Requirements  (N,P,Z,C)                                         Required
C Contributor's name

For each file to be patched (if any)...

F Filename to patch                                               Required
D Filedate
? Check         (address s1,s2,s3,...sn)
& Patch         (address s1,s2,s3,...sn > d1,d2,d3,...dn)         Required
' Comment
```

Figure 4
PATCHIT.DAT Field Summary

```
**********************************************************************
N Norton Utilities, Standard Edition
P Peter Norton Software
V 4.5
S MS-DOS disk utilities
R N
' The line above indicates that no patches or ZHS circuit is required

**********************************************************************
N LogiCADD
P Generic Software
V 2.0
S This is the Logitech version of Generic CADD
R P
I In order to use a mouse with LogiCADD, a PC style COM port will be required.
C Paul F. Herman Inc.

F CADD.EXE
D 10/02/86
   Change all instances of ... MOV BX, F000 ...to... MOV BX, B000
& DCCB F0 > B0
& DD35 ? > B0
& DD72 ? > B0
& DDDB ? > B0
   Change all instances of ... MOV SI, FA6E ...to... MOV SI, 0
& DE2C 6E,FA > 0,0
& DE94 ? > 0,0
& DF0F ? > 0,0
```

Figure 5
PATCHIT.DAT Program Entry Samples

volunteer (who, me?) to collect, organize, and distribute it, the only remaining question is how to make it available to other Z-100 users. Since I must support a family and run a business, as well as write this column, I have to be able to cover my expenses of handling and distributing the PATCHIT disk. If you want a copy of the latest version of the PATCHIT disk (which includes the PATCHIT.EXE program and the PATCHIT.DAT file) send me your request along with a check for $10.00. No phone orders or VISA/MC orders will be accepted. The contents of the disk may be considered to be in the public domain for anyone to use, so feel free to give it to anyone you please, or upload it to your favorite bulletin board.

As most of you know, Z-100 specific articles are getting harder and harder to find, so I'm not really willing to squander valuable column space describing lengthy patches to programs so they will run under ZPC. If I published patch information in this column, there just wouldn't be much space for more interesting stuff. So the preferred method of distributing patch information for now will be on disk, as described above. From time to time, I may break down and do a column about new programs that we've been able to get running under ZPC, or other interesting news on the ZPC scene.

The Future for ZPC

Although some will skoff at the idea, I dare say that Pat Swayne's ZPC is one of the most important pieces of software that has ever been written for the Z-100. It is a prestigious piece of work which takes software PC emulation way beyond the point most people thought was possible. I wish I had written it.

What does the future hold for ZPC? I tread lightly in this area, because when all the cards are played, the fact is that the future of ZPC is for Pat Swayne to determine. It's been several years since the last ZPC Upgrade disk, and in the mean time, lots of changes to the ZPC program itself have been put forth by users. Pat acknowledges these modifications, and agrees that some of them offer substantial improvements in PC emulation, or program operation.

Some of the modifications have been published in REMark and SEXTANT magazines. And many are available on the Heath Bulletin board for all to explore. Will another ZPC upgrade be forthcoming? We'll just have to wait and see. ✱

Z-100 Survival Kit #10

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

Learning How to Patch Programs for ZPC

I guess I'm just going to have to face up to the fact that most of the Z-100 world craves the ability to run PC compatible software. A big percentage of the mail I receive from "Survival Kit" readers is from people who want to know more about how to patch programs to run under ZPC. I've tried to avoid doing this, but . . . the pressure is unrelenting. By popular demand, Survival Kit #10 and #11 will be devoted to teaching you how to become an expert in the art of ZPC patching.

If you will recall, I stated in Survival Kit #1 that I didn't want this column to turn into a "ZPC Update" series. That is still my feeling, but it appears that the only way I can maintain this position is to provide ZPC patch information on the side. The first step in this plan was taken in Survival Kit #9, where I announced a new PATCHIT utility, and gave a detailed format specification for program patch information which is submitted by users. The obvious next step is to teach you all how to develop the patches so that all I will have to do is organize the data and distribute it.

When in Doubt, Read the Instructions

Probably the most frustrating aspect of trying to provide technical support to people who want to develop patches is the fact that most of the information they need is right in the ZPC manual. Pat Swayne has given a very adequate (and understandable) explanation about the type of things which require patching for operation with ZPC.

Over the course of this, and the next, installment of Z-100 Survival Kit, I will be guiding you by the hand through procedures needed to patch programs. But in the end, most of what we cover will be summarized by pages 11 through 14 of the ZPC manual.

What Seems to Be the Problem?

I am left with the conclusion that there are five factors leading to ignorance about how to make ZPC patches (see the list in Figure 1).

1. Users don't have an understanding of assembly language programming.
2. Users don't know how to use an editor like DEBUG to search for areas to patch.
3. Users don't understand how PC specific programs work, or why patches are required.
4. Users haven't read pages 11 through 14 of their ZPC manual.
5. Users don't have a ZPC manual because they have a 'borrowed' copy of ZPC.

Figure 1

I tried to express number 5 as politely as possible; if this is your excuse, then you really have no excuse. Making unauthorized copies of commercial software (yes, ZPC is a commercial program!) is not only illegal, but immoral. And stealing it from a users' group is double-dirty. Shame, shame! If you fall into group 1, there may not be any hope for you. At least not for the next several months, while you take a crash course in 8088 assembly language. Yes, most of these patches could be made by a user who doesn't know what he is doing, but as soon as something varies from the game plan which is described, you'll be lost. Much of the code that needs patching can be written in a number of ways, and developing appropriate patches will require at least an elementary understanding of what is taking place in the program.

If number 3 is your problem, it may be because you also fall into category 4. I'll be covering specific patch circumstances in the next installment of Survival Kit. We'll take a detailed look at what types of things PC specific programs do that cause problems, and what patches are required to correct them.

The rest of this column will be devoted to group number 2. I know (based on letters and phone conversations) that many of you are not proficient in using DEBUG. It is absolutely necessary that you understand how to use an editor which allows byte-by-byte editing of a file. This could be DEBUG, or SYMDEB, Norton Utilities, HADES, or any number of other public domain file editor programs which are available. Our discussion will concentrate on the use of DEBUG, mainly because everyone has a copy (it

comes with DOS).

DEBUG — A Short Tutorial

DEBUG.COM is a down-and-dirty file editor that has been included with DOS since day one. There are lots of alternative editors available, but I happen to think that DEBUG is a nice program. It's short. It's quick. And it does everything you need it to do, to develop program patches.

There have been quite a few articles in various publications about how to use DEBUG. And Zenith's version of the MS-DOS manual contains a more than adequate description of how to use the program. But most users aren't willing to take the time to study the documentation thoroughly. DEBUG is one of those kinds of programs that you don't use very much. So it seems a waste to learn every detail about how to use it. The result is that you have to refer to the manual each time you want to use DEBUG, and some of the finer details of its use are neglected. I know, because I still have to refer to the DEBUG manual occasionally when I use it.

For our purpose at hand, which is developing patches for programs to run under ZPC, we don't need to know how to use all of DEBUG's features. In fact, the only DEBUG commands we need to worry about are those shown in Figure 2.

A	Assemble 8088 code instructions
D	Dump (or Display) contents of memory
E	Enter or change memory locations
Q	Quit, and return to DOS
R	Register contents (display or change)
S	Search memory
U	Unassemble 8088 code instructions
W	Write file to disk

Figure 2

I find that most users have an elementary understanding of these commands. But in order to use DEBUG to find patches (or make patches), there are a few additional tricks of the trade that aren't immediately obvious from reading the manual. We're going to cover each of these commands in detail, paying special attention to details which will be necessary for our goal of patching programs. I'll describe the commands in the order we're likely to use them, not in alphabetical order as they are listed in Figure 2. But first, we need to take a look at how DEBUG is started, and how it loads programs into memory.

Invoking DEBUG

DEBUG may be invoked with the following command:

```
DEBUG filename.ext
```

This command runs the DEBUG program, and tells it to load the named file into memory. This is pretty simple to understand, but there is a complication; if the program we want to patch is an EXE program, its name must be changed before loading it with DEBUG. If you attempt to write an EXE file back to disk with DEBUG, you will get an error message that says "EXE files cannot be written".

There is a good reason why DEBUG cannot write an EXE file to disk, and why we can't directly patch an EXE program. The DEBUG manual isn't very clear on this situation, so let's take a few paragraphs to find out what's going on.

When you load a program file (.COM or .EXE) with DEBUG, the program is not simply loaded into memory. There's more to it than that. You see, DEBUG is more than just a hex file editor — it also allows you to execute or single-step through programs. Therefore, when DEBUG loads a program, it loads it just as though it was going to run the program.

For COM programs, this means that a program segment prefix (PSP) is built at the first available load address. The PSP is 100h bytes long, and contains information that may be useful to the program, such as the location of the environment strings, address of the control-C handler routine, any command line arguments, etc. Immediately after the 100h byte PSP, the COM program is loaded. This is exactly the way the program would be loaded into memory by DOS, if you were executing it from the DOS prompt. You'll notice, when using DEBUG to investigate a COM program, or other file, that the (D)ump or (U)nassemble commands always begin with address 100h in the current segment. That's where the actual program begins. And whenever a program or file is written back to disk, the default start location for the write operation is 100h.

The way EXE programs are handled is completely different. Every EXE program begins with a header that includes initialization information, as well as relocation (or fixup) addresses. When DEBUG loads an EXE program, it does it just as though it were preparing to run the program. This means that a PSP is built, and information in the file header is used to perform "fixups" to the program code as it is being loaded into memory. Without the address fixups, you wouldn't be able to execute or single-step the program. After the program is loaded, the header information is simply discarded.

Now, there are several reasons why the resulting EXE memory image which has been loaded can't be written back to disk. First of all, the EXE file header (which is at least 512 bytes long, maybe larger) is not stored in memory. It is used to load the EXE file, and then overwritten. If you were allowed to write the program back to disk, it would not contain the required header information. Secondly, since fixups were made to the program code after it was loaded into memory, the code you see with DEBUG is not the same as the code in the program on disk (the jump addresses will be different). In other words, the memory image of the EXE program is different from the disk file image.

Since our main objective is to be able to make patches to a program, and write it back to disk, it is obvious we can't work directly with EXE files. The solution to this problem is, however, very simple. Just rename the EXE program so that it has a different extension. Then DEBUG will think it is working with a non-EXE file. My favorite substitute name is simply the program name without any extension. In other words, rename TEST.EXE to just TEST.

DEBUG Preliminaries

Okay, let's assume that you have invoked DEBUG, and loaded the program you want to patch. If it was an EXE program, you renamed it first. Now what? Try hitting 'r' and then RETURN. You should see something that looks like Figure 3.

This display shows the contents of all of the 8088 CPU registers, along with the current instruction pointed to by the instruction pointer. I'm not going to try to explain everything there is to know about this display of register contents — I'll assume that you have the basic assembly language knowledge it takes to figure most of it out (remember group 1?).

There are, however, a few specific points I would like to make while we are here looking at the register display. First off, note that DEBUG displays numbers (and expects them to be entered) in hexadecimal notation — exclusively. If you don't know the hexadecimal (base 16) numbering system, take time out right here to learn more about it — otherwise, you'll be lost.

Notice that DS, ES, SS, and CS all contain the same value (2044h in our example). This will always be the case for COM programs or plain vanilla files. The actual segment address may (probably will) be different on your system. It will even be different depending on what type of memory-resident utilities or device drivers you have loaded before invoking DEBUG. The address shown in these segment registers is the first available load address. Remember, the PSP is loaded at this segment address, and then the program is loaded. Look at the value for IP (the instruction pointer) in Figure 3. The fact that it says 100h should be no surprise at this point.

There are two other registers which are significant to our patching goal. Registers BX and CX will contain the length of the program which was loaded. The actual length is BX:CX. For programs which are less than 64K (65536) bytes long, BX will be zero. In the example shown in Figure 3, the length of the program we loaded is

0001:2F00h (which is the same as 12F00h). This translates to 77568 decimal bytes. The length of the program will be very important when it comes to searching for patch locations.

Okay, now we're ready to examine the individual DEBUG commands.

R — Register Contents (Display or Change)

This is the command we used above to display the contents of all the CPU registers. You can also use this command to change the register values. Enter 'R' immediately followed by the two letter name of the register you want to change. For example, type 'RCX' if you want to change the CX register. After you hit RETURN, the current register value will be displayed, and you will be allowed to type in a new value.

You probably will not need to change the values of any registers when patching programs. The most important use for the 'R' command, for our purpose here, is to determine how long the program is, and to find the segment load address.

D — Dump (or Display) Contents of Memory

The (D)ump command is used to display the hex values of memory. The syntax is:

```
D[address] [L value]
```

The address you specify is the start memory address for the bytes to display. The address may be composed of a segment and offset, or simply the offset. If just an offset is specified, the current data segment (value in DS) will be used. If no L value is specified, 128 (80h) bytes will be displayed. Here are some examples:

```
D500
```

This command will dump 128 bytes beginning at offset 500h in the current data segment.

```
D3044:2000 L 100
```

This command will dump 100h bytes beginning at segment address 3044h, offset 2000h. Notice that this is offset 2000h into the second 64K of the program, assuming that the initial data segment is 2044h, as shown in Figure 3.

If you simply type 'D' with no arguments, the next 128 bytes (since the last Dump command) will be displayed.

The Dump command will be useful to us for finding particular locations in the file to patch. Sections of the program which contain ascii text may be quickly examined by using the Dump command.

E — Enter or Change Memory Locations

The 'E' (Enter) command is used to view a particular memory location, and to change it. The syntax is:

```
Eaddress
```

where 'address' is the memory location where you want to begin viewing or changing values. DEBUG also allows you to include a list of hex values on the 'E' command line, but we won't need to worry about that feature. As an example of how the Enter command is used, suppose we want to change several bytes beginning at offset 520h in the program. We would issue the command:

```
E520
```

followed by a return. DEBUG will display the present value of the byte at offset 520 in the data segment, and then allow us to change it. If we decide we just want to skip this byte, we can simply hit the spacebar. After a new value is entered, or the spacebar is struck, the next byte is displayed for our review. This continues until we hit the RETURN key. As with the dump command, the address used with the Enter command may include a segment prefix if the memory address isn't within the first 64K of the program.

The Enter command will be our main tool for patching programs, so you must understand exactly how it works. If doubt still lingers, fire up DEBUG and play with it a bit. The best way to get the hang of it is practice, practice, practice!

S — Search Memory

The Search command is the second-most powerful tool we have in our quest for proper patches. (The most important tool is, or should be, your brain). The proper syntax for the Search command is:

```
Srange list
```

'range' specifies where the search is to begin, and how many bytes to search. This may be done by specifying a start and end address, as in this example:

```
S100 500 50 61 75 6C
```

which will search from offset 100h to 500h for the occurrence of four bytes with the values 50h, 61h, 75h and 6Ch ('Paul'). We could have also used a segment prefix for the start address, like this:

```
S3044:0 FFFF 5A 31 30 30
```

This example would search 65535 bytes beginning at paragraph 3044h for the four specified bytes. Another way of telling DEBUG the search range is by specifying the actual number of bytes to search, instead of the end address. For example:

```
S100 L 400 50 61 75 6C
```

This command tells DEBUG to search 400h bytes beginning at offset 100h. This example has exactly the same results as our first Search example.

One important time-saving feature you'll want to note is that the 'list' of bytes to search for may be given as an ascii string of characters, instead of a list of hex bytes. The ascii search string should be enclosed in single quotes, like this:

```
S3044:0 FFFF 'Z100'
```

This is exactly the same command as our second example above, except it is a lot simpler to enter.

DEBUG's search range is limited to 0FFFFh bytes, so if you have a program which is longer than 64K, you'll need to search it in several steps. For instance, suppose the program is 192K long, and our registers after starting DEBUG are as shown in Figure 3. In order to search the entire file for the string 'Heath', these three commands would be required . . .

```
S2044:0 FFFF 'Heath'
S3044:0 FFFF 'Heath'
S4044:0 FFFF 'Heath'
```

While DEBUG is searching memory, the segment:offset of each match is listed on the screen. Be prepared to use Control-S if you are searching a large range, because the list of match addresses may quickly scroll off the screen.

It doesn't hurt anything if you tell DEBUG to search a range which extends beyond the limits of the currently loaded program. But be sure to check the program length to make sure that any matches you find fall within the program area. Another thing to keep in mind is that DEBUG's searches are case sensitive. This means that searching for 'Heath' will not find an occurrence of 'heath', since the first character is not capitalized. A good practice when searching for words which may be capitalized, is to start the search string with the second letter. For instance, if you are looking for the copyright notice in a program, search for 'opyright'.

A couple of important notes about DEBUG versions . . . Some of the older versions of DEBUG may not allow you to enter your search list as an ascii string. Check your documentation (or just try it) to determine if your version falls into this category. Also, some versions of DEBUG may impose a search limit of 8000h bytes, instead of 0FFFFh. If you have one of these versions, and you need to search an entire 64K block of code, you'll have to do it in two steps.

U — Unassemble 8088 Code Instructions

The Unassemble command allows us to see the CPU instructions that make up the program. Without the 'U' command, the program is nothing but a bunch of hex bytes. We will use the Unassemble command extensively to determine where patches need to be made. The syntax for this command is:

```
U[address] [L value]
or
U[range]
```

As with the Search command, 'range' may consist of a start and end address, or a start address and the number of bytes to unassemble. If only the start address is given, 32 bytes will be unassembled beginning at the specified address. Or you can specify the number of bytes to unassemble with the 'L' option. If the 'U' command is given with no arguments, 32 bytes are unassembled beginning at the current position of the instruction pointer. If the instruction pointer (IP) has not been altered since the last 'U' command, then the unassembly will continue from the point last unassembled. Unlike the Dump

```
-r
AX=0000  BX=0001  CX=2F00  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2044  ES=2044  SS=2044  CS=2044  IP=0100  NV UP EI PL NZ NA PO NC
2044:0100  01711F        ADD       [BX+DI+1F],SI
```

Figure 3

and Search commands, the Unassemble command (and Assemble command) use the value of the code segment (CS) if no segment prefix is given. This may all be a bit confusing, so let's have a few examples:

```
U100 150
```

This command causes the bytes between 100h and 150h of the current code segment to be unassembled.

```
U3044:0 L 50
```

This example will unassemble 50h bytes in paragraph 3044h, beginning with offset 0.

```
U
```

If this command (Unassemble with no arguments) were issued immediately after the previous example, it would cause the next 20h bytes to be unassembled.

A — Assemble 8088 Code Instructions

The Assemble command is the counterpart to the Unassemble command. It allows you to enter 8088 mnemonic instructions, and will convert them into byte values. In other words, it allows you to enter assembly language program instructions. The syntax is simple:

```
A[address]
```

If an address is specified, assembly of instructions will begin at that address. The address may contain the segment, as well as offset. If no address is specified, assembly will begin at offset 100h in the current code segment, or at the last address where instructions were assembled.

The Assemble command is another method we have for changing the program code. It may be preferable in some cases to use the Assemble command to patch programs, instead of the Enter command. This is particularly true when we need to change large areas of ascii text, since we can use the Assemble command to enter DB instructions like this:

```
DB 'Z-100 Survival Kit'
```

This is much easier than looking up all the hex values for the ascii characters, and entering them one at a time with the Enter command.

W — Write File to Disk

After we have found our patches, and made our changes, the 'W' command is used to write the file back to disk. The Write command may be used with arguments which will cause it to do all kinds of neat tricks, but for our purposes, all we need to know is 'W', followed by RETURN.

One thing you should take note of, however, is that the actual number of bytes written to disk is controlled by the BX:CX registers. Remember way back at the beginning, when a program is loaded, that the BX:CX registers contained the program length. DEBUG uses this value to determine how many bytes to write. So make sure that registers BX and CX don't get changed between the time you load your program and the time you write it back to disk.

Q — Quit, and Return to DOS

It doesn't get much simpler than this. Hit 'Q' and RETURN. Your done! DEBUG doesn't give you a second chance on this one, so make sure you have saved your changes by using the Write command.

One tip for the time when you do inadvertantly quit without saving your patched program . . . If you invoke DEBUG again, immediately, and without any command line parameters, you'll find that your program is still there, intact. However, the program length will not be properly recorded in the BX:CX registers at this point, so you'll have to be sure and set them to the proper length before issuing the Write command to save your work.

Get Ready for the Good Stuff!

We now have most of the preliminaries out of the way. In the next issue of Survival Kit, we will begin probing deeply into the why's and how's of making ZPC patches. In the mean time, practice using DEBUG until you know these commands we have discussed. Your time will be well spent.

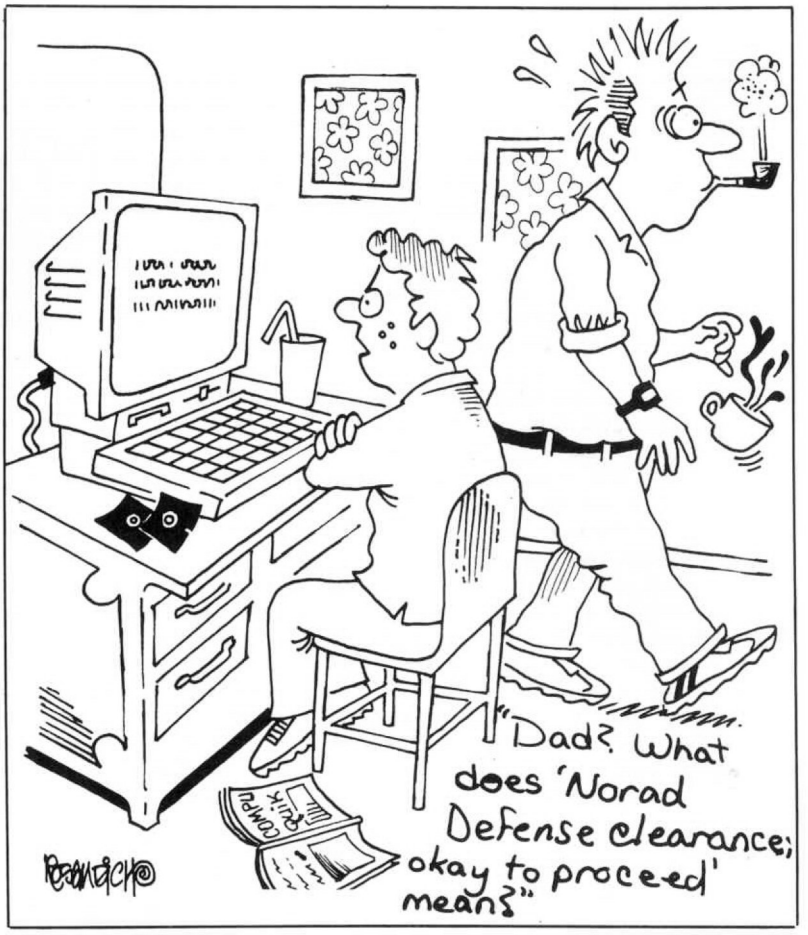
Until then, keep in touch!

*

Z-100 Survival Kit

#11

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655



Patching Programs to Run Under ZPC

The last two installments of Z-100 Survival Kit have dealt with the subject of patching programs for use under HUG's ZPC emulator program. In the last issue we covered some of the preliminaries, like how to use DEBUG. This month's column will conclude the discussion with specific details of what types of problems we encounter in PC programs that require patching to run under ZPC.

Surveying the Scope of the Problem

It seems to me, the most logical way to go about this project is to start from the top and work down. By that, I mean to say that we first need to understand the major reasons why patches are required, and then consider some of the strategies for making the patches. Then, as space permits, we can go into some examples of actual patches that might be applied to a program. I'm sure that a comprehensive treatment of the subject, that covered every aspect of patching programs for ZPC, would easily fill a large book. And I'm also sure that the book would never make the 'best seller' list. We'll just have to do the best we can, with the resources and space that is available.

Patch Logic — Why Some PC Programs Won't Run Under ZPC

ZPC is a software solution to the problem of PC compatibility for the Z-100. And although the emulation provided is amazingly complete, considering the differences between a Z-100 and IBM-PC, some things just can't be emulated. These 'un-emulatable' things tend to fall into specific categories. See Figure 1.

Port Accesses
Unsupported Interrupts
The Graphic Character Table
BIOS Data Segment Accesses
Specific Hardware Constraints

Figure 1
General Patching Categories

Far and away the most common reason a program won't run under ZPC is because it makes accesses to ports which don't exist on a Z-100. The I/O port map of an IBM-PC doesn't bear any resemblance to that of a Z-100. So any port accesses made by PC software are potential troublemakers.

Another area of concern is the handling of interrupts. Since interrupts are processed by software interrupt routines, there is a great potential for software emulation of interrupts, and in fact, this is how ZPC provides most of its PC compatibility. But there are a few PC interrupts that cannot be supported because of hardware incompatibilities.

The graphic character table used by many PC programs is held in PC memory at an address already used by the Z-100's MTR-100 monitor ROM, which precludes moving a copy of the table there for an emulated environment. This problem is easily overcome by patching the address of the table in PC programs.

Some programs access the IBM-PC BIOS data area directly to learn about system parameters and addresses. Where possible, ZPC has tried to replicate the PC data area, but there are still problem areas.

And finally, there are some hardware devices and accessories for PC compatible computers that just don't lend themselves to software emulation. Good examples would be EGA/VGA video cards, sound generation hardware, and direct access of I/O hardware by PC programs.

A Note About ZHS Circuits and ZPC

Many of the patches we will be discussing may be unnecessary if you have a ZHS circuit (Scottie board) installed in your Z-100. This simple hardware device, described in previous issues of REMark, allows many PC compatible programs to run under ZPC, which would otherwise require patches. My discussion here, however, will assume that no Scottie board is installed.

General Patching Strategy

Before we dive into the actual process of locating patches to correct specific problems, an explanation of our overall strategy would be in order. The basic methods we will use will be the same, regardless of what particular patch we are going to make.

The hardest part of the patching pro-

cess may be in deciding what type of patches are required. There will be some clues. For instance, if text on the screen is unrecognizable, that's a good indication that the address of the graphic character table needs patching. But in many (if not most) cases, the program will simply hang, or crash. Trying to decide what to look for will be difficult, and may be a matter of experience and intuition, more than anything else.

Once we know what we're looking for, we'll use DEBUG to do the searching. Sometimes the search will be as simple as looking for a text string. But other times, we'll need to use the unassemble command to find the byte values of likely Assembly Language instructions. In many cases, there will be more than one way the Assembly code we're searching for could be written. This will require some skill in Assembly Language, and a great deal of patience.

As an example, we may want to search for accesses to the PC ports. These accesses might be done with Assembly Language instructions like this:

```
MOV   DX, 3D8
OUT   DX, AL
```

We can use the DEBUG unassemble command to determine that these Assembly instructions have the following byte values:

```
BA D8 Ø3     MOV   DX, 3D8
EE           OUT   DX, AL
```

Now we can use the DEBUG search command to search for occurrences of these bytes, like this:

```
SØ FFFF BA D8 3 EE
```

This sounds simple enough. (If it doesn't, then you need to go back and read Z-100 Survival Kit #10, which was a quick DEBUG tutorial.) But there is a big catch. The Assembly code doesn't have to look like that shown above. The DX register could be loaded from another address. Or the MOV DX instruction might be separated from the IN instruction by other code.

Of course, you could be sure of finding all of the port accesses by simply searching for all occurrences of the port access instructions:

```
IN    AL, DX
IN    AL, (port number)
OUT   DX, AL
OUT   (port number), AL
```

The problem with this is that you would have to sort out the long list of prospective candidates to see which ones are significant, and which ones are simply random data that happen to have that byte value.

I think you can begin to see why a fair amount of Assembly Language knowledge is necessary to become proficient in ZPC patching.

After the patch location (or locations) is found, then we need to make the changes to the program using the DEBUG enter or assemble commands. The program can now be written back to disk and tested, if desired. Of course, you should also make a note of the changes you have made, so that you will not have to go through this laborious procedure again in the future. And if you are finally successful in getting the program to run under ZPC, you should send a copy of the patches to me, so that I can spread the news around a bit.

Port Access Patches

The way a program communicates with its hardware environment is by reading and writing to ports. Ports are accessed by a particular port address. In the Z-100, all of the I/O ports have addresses which must range between 0 and 0FFh. In an IBM PC, the I/O ports may have addresses anywhere between 0 and 3FFh. Special instructions are used to communicate with ports. And port addresses should not be confused with RAM memory addresses.

The Z-100 has no I/O ports in common with the IBM PC. (See Figures 2 and 3 for a breakdown of the Z-100 and IBM PC ports.)

000-00F	DMA Controller
020-021	Interrupt Controller
040-043	Timer
060-063	PIA, Keyboard
080-083	DMA Page Registers
200-207	Game I/O Adaptor
210-217	Expansion Unit
278-27F	Parallel Printer 2
2F8-2FF	Secondary COM Port
320-32F	Hard Disk Controller
378-37F	Parallel Printer
3BC-3BF	Monochrome Display Adaptor
3C0-3CF	EGA Adaptor
3D0-3DF	CGA Adaptor
3F0-3F7	Floppy Disk Controller
3F8-3FF	Primary COM Port

Figure 2
IBM PC/XT Port Usage

So strictly speaking, ANY port address by a PC program is a likely troublemaker. We are, however, lucky in two respects. First of all, direct port accesses are not necessary if a program takes advantage of the IBM PC BIOS services, so not many programs make port accesses except for the video ports. Secondly, many of the common port addresses do not conflict with the Z-100 ports, or cause no trouble if they do. If there is an attempt to write to a non-existent port, nothing happens.

There is one major problem that is caused by the two different addressing schemes used by the Z-100 and IBM PC. Whenever a PC port with an address of greater than 0FFh is accessed, the high byte is ignored, causing an access to the port whose address is indicated by the least significant byte. In other words, all four pages of PC ports are mapped into the single page of Z-100 ports.

A8-AB	Primary Hard Disk Controller
AC-AF	Secondary Hard Disk Controller
B0-B7	Primary Floppy Controller
B8-BF	Secondary Floppy Controller
D8-DB	Video Control Port
DC-DD	CRT Controller
DE	Light Pen
E0-E3	Parallel Port
E4-E7	Timer
E8-EB	Serial Port A
EC-EF	Serial Port B
F0-F1	Slave Interrupt Controller
F2-F3	Master Interrupt Controller
F4-F5	Keyboard
FB	Timer Status
FC	Memory Control Latch
FD	High Address Latch
FE	Swap Port
FF	DIP Switch

Figure 3
Z-100 Port Usage

If you want to become an expert at the art of ZPC patching, you'll need to compile all the information you can about how each I/O port in the IBM PC and the Z-100 is programmed, and what each bit of each byte does. This way, you can go through a particularly troublesome patching problem, and analyze which port accesses are benign, and which are serious.

Reading the Port Status

One of the most common problems associated with port accesses involves programs that try to read the status of a port. Reading a port will never cause anything to crash directly — it is a harmless act. The problem comes when the program insists on waiting for a particular port status to change.

A good example of this is when a program wants to write directly to video RAM on a CGA display. On some IBM PC models, if the video memory is accessed at the same time the screen is being refreshed, interference will appear on the screen. In order to avoid this problem, many programs check the video status register (port 3DAh). Bit 0 of this port indicates that it is okay to access video RAM. The code used to check the video status may look like this:

```
      MOV   DX, 3DA
LOOP: IN    AL, DX
      TEST  AL, 1
      JZ    LOOP
```

The problem here is that when the program tries to read port 3DAh, it really reads port 0DAh, which is the CRT-C address latch port on a Z-100. Chances are good that bit 0 of the address latch port will never satisfy the test in the code above, so the program will loop forever. The result, from the operator's point of view, is that the computer is locked up.

The patch that is required to correct this situation is really simple. Simply replace the IN, TEST, and JZ instructions with NOP's. Or you could just replace the JZ instruction with NOP's, since reading the status isn't going to hurt anything. Or you could replace the IN instruction with a JMP to the end of the loop. Whichever way you do it, the result is that we eliminate testing the port status. This simple example should demonstrate that there may be several ways to make a patch to a program.

Now I know a few of you are asking "how will the program operate with this code removed?" That's a good question, and one which you will always have to consider when making patches. In this particular example, the patched out code won't make any difference, because the Z-100 doesn't have any problem with interference when writing to the screen. But in other cases, the status read from the port may be used by subsequent instructions. If this is the case, you may have to tailor your patch to provide the desired response. In our example above, suppose that the program saved the status byte and used it for some other test. Then we might make a patch that looks like this:

```
MOV   DX, Ø3DA
MOV   AL, 1
JMP   LOOPEND
```

Writing to a Port

A completely different set of problems crop up when a program tries to write data to a port. Now, instead of just being a harmless access which may result in an endless loop, we have the program trying to tell the hardware what to do. And in every case, the expected hardware is not there. The best we can hope for is that the port instruction will be ignored.

Suppose that a PC program wants to change the graphics mode, and elects to do it by writing directly to the video mode control register (port 3D8h) instead of using the BIOS services. The access to port 3D8h will be mapped to port 0D8h in the Z-100, which is the video control register. It is only a coincidence that both of these port addresses happen to be video control registers. The sad fact is that any access to the PC's video mode port is going to make strange things happen to the Z-100's screen display.

There are hardly any programs that write to port 3D8h to change the video mode, since a lot of other overhead is required in addition to this simple register access, and since it is so easy to use the PC's ROM BIOS to make the mode change. But there are other video ports that are used commonly, and can cause just as disastrous of effects.

The solution for cases which cause problems is simply to patch out the offending instructions with NOPs, and hope for the best. Obviously, since the PC program was trying to write something to a port, there may be ramifications if it doesn't get done. In some cases, the program will work correctly under ZPC anyway.

The purpose of this discussion has been to try to give you a 'feel' for the reasoning behind the patches. The explanation has been conspicuously shy of concrete examples, mainly because it would be impossible to cover all the possibilities. And I don't want to mislead you into thinking that you can look for a few fixed samples of code. You MUST understand the logic of what needs to be patched . . . then you can worry about the specific code fragments that need to be located.

Reading the ZPC User's Manual is an important part of the learning process. Along with a description of which interrupts and other services that ZPC emulates, are discussions of program patching. For some of you who would like to see some concrete examples of program patches, the ZPC manual has a few.

Before we finish our discussion of patching accesses to I/O ports, I need to call your attention to the section "Patching Programs: the Keyboard Interrupt" in the ZPC manual. This is required reading for all prospective patchers. The keyboard port (port 60h) naturally does not exist at that port address in the Z-100, but it is such an important service, that some provision had to be made to emulate it. This was handled by using two interrupt handling routines; INT 90h and 91h. Generally speaking, you can patch any instructions which read from ports 60h and 61h with a software interrupt to INT 90h or 91h, respectively, and the code returned in register AL will be a value appropriate for the original port access.

Unsupported Interrupts

Another thing which may cause problems when running PC programs under ZPC is an unsupported interrupt. Of course since the program is really running on a Z-100 computer, there won't be any unsolicited PC hardware interrupts to worry about, but the program may issue software interrupts that cause problems.

Figure 4 shows a list of IBM PC/XT interrupts.

Int	Type	Description
00	System	Divide by Zero
01	System	Single-Step
02	System	Non-Maskable Interrupt
03	System	Breakpoint
04	System	Overflow
05	BIOS	Print Screen
08	Hardware	Timer
09	Hardware	Keyboard
0B	Hardware	COM2
0C	Hardware	COM1
0D	Hardware	Hard Disk
0E	Hardware	Floppy Disk
0F	Hardware	LPT1
10	BIOS	Video Services
11	BIOS	Equipment Check
12	BIOS	Memory Size
13	BIOS	Disk I/O
14	BIOS	Serial I/O
15	BIOS	Cassette I/O
16	BIOS	Keyboard I/O
17	BIOS	Parallel I/O
18	BIOS	Resident BASIC
19	BIOS	Bootstrap
1A	BIOS	Timer
1B	BIOS	Keyboard Break
1C	BIOS	Timer Tick
1D	BIOS	Address of Video Parameters
1E	BIOS	Address of Disk Parameters
1F	BIOS	Address of Graphics Characters
20	DOS	Program Terminate
21	DOS	Function Request
22	DOS	Terminate Address
23	DOS	Control-C Address
24	DOS	Fatal Error Address
25	DOS	Absolute Disk Read
26	DOS	Absolute Disk Write
27	DOS	Terminate/Stay Resident

Figure 4
IBM PC/XT Interrupts

00	Hardware	Divide by Zero
01	Hardware	Single-Step
02	System	Non-Maskable Interrupt
03	System	Breakpoint
04	System	Overflow
05	BIOS	Print Screen
20	DOS	Program Terminate
21	DOS	Function Request
22	DOS	Terminate Address
23	DOS	Control-C Address
24	DOS	Fatal Error Address
25	DOS	Absolute Disk Read
26	DOS	Absolute Disk Write
27	DOS	Terminate/Stay Resident
40	Hardware	Parity Error
41	Hardware	Processor Swap
42	Hardware	Timer
43	Hardware	Slave IC
44	Hardware	Serial Port A
45	Hardware	Serial Port B
46	Hardware	Keyboard/Light Pen/Vertical Retrace
47	Hardware	Parallel Port

Figure 5
Z-100 Interrupts

The interrupt table for a Z-100, running in native Z-100 mode is shown in Figure 5.

You should note that in both of these interrupt table descriptions, there are other interrupts which have been defined. The ones shown are those which are most commonly used.

Of these, the first five (0 through 4) are defined by Intel for an 8088 system, and therefore, are equivalent between a PC and Z-100. Interrupt 5 is for support of the Print Screen key, and will typically not be found in a program. Interrupts 6h through 0Fh are generated by the PC hardware, and are difficult to emulate with software, but ZPC does synthesize an interrupt 9h to indicate keyboard activity.

Interrupts 10h through 1Fh are all vectored to entry points in the PC's BIOS ROM, and are typically called via a software interrupt in a program. These interrupts are very commonly used, and luckily, they are undefined in the Z-100's normal interrupt table (compare with Figure 5), and therefore, available to use for emulation purposes. ZPC does a good job in emulating these interrupt services. The specific interrupts which are supported by ZPC are listed in the ZPC User's Manual.

The Interrupts between 20h and 27h are MS-DOS defined interrupts, and are compatible between the PC and Z-100.

Even though many of the interrupts have been emulated successfully by ZPC, many have not. It is these cases where you will receive the infamous "WILD INTERRUPT" message when attempting to run PC software. If you run into a program that uses an unsupported interrupt, about the only thing you can do is patch the instruction out with NOPs and hope for the best. If you're lucky, the interrupt call will not affect the operation of the program, and everything will work okay. But if the program is trying to perform Disk I/O through the BIOS, or other critical functions, the program will almost certainly crash.

The Graphic Character Table

Whenever a PC program writes text on the screen while in graphics mode the text character font designs are looked up in a table in PC memory. This is known as the graphic character table, and it is located at address F000:FA6E. It would be nice if this table could be put in that location for emulation purposes, however, the Z-100's MTR-100 ROM occupies that space already. (An experimental MTR-100 ROM version has been independently developed which includes this table, but it is not yet available — see Z-100 LifeLine Issue #4 for further discussion.)

The cure for this problem is relatively simple. Simply find where the PC program is accessing this table and patch the address so that it points to the table ZPC has provided, which is at memory location B000:0000. There are a number of different instructions that a PC program might use to access this table, but it is usually pretty easy to find the table references by searching for the FA6E address. Remember when you are conducting your search with DEBUG, that the byte order that Intel uses places the least significant byte first in a word. Therefore, you should search for the bytes 6Eh, 0FAh, in that order.

BIOS Data Segment Accesses

In a real IBM PC, the segment beginning at paragraph 40h is used by the BIOS as a data segment. This area is normally used as a BIOS jump table by the Z-100, but Pat Swayne has figured out how to make this area available for use by ZPC for PC BIOS data. Most of the data values maintained in this area by ZPC are typical of what you would find on a real PC, and won't cause any problems. But if you run into a program that is trying to manipulate the PC BIOS type ahead buffer directly, or trying to determine hardware status, it will have problems. About the only thing you can do in these cases is patch the program so that it returns with the expected value. This may or may not affect a cure.

Specific Hardware Constraints

Luckily, most PC software is fairly well behaved. That is to say that most software uses the BIOS or DOS for communication with the hardware. The major exception is direct access of the video RAM, but ZPC has solved this problem nicely.

Some programs, however, will insist on communicating with the hardware directly. Any attempt to communicate directly with the video ports, serial or parallel ports, interrupt controller, DMA controller, keyboard chip, etc. must be patched. In most cases, patching out an access to a hardware device will cause the program to crash anyway. The most likely candidates for patching are the video port accesses. If you have a ZHS circuit with COM ports installed, direct accesses of the COM ports should be okay.

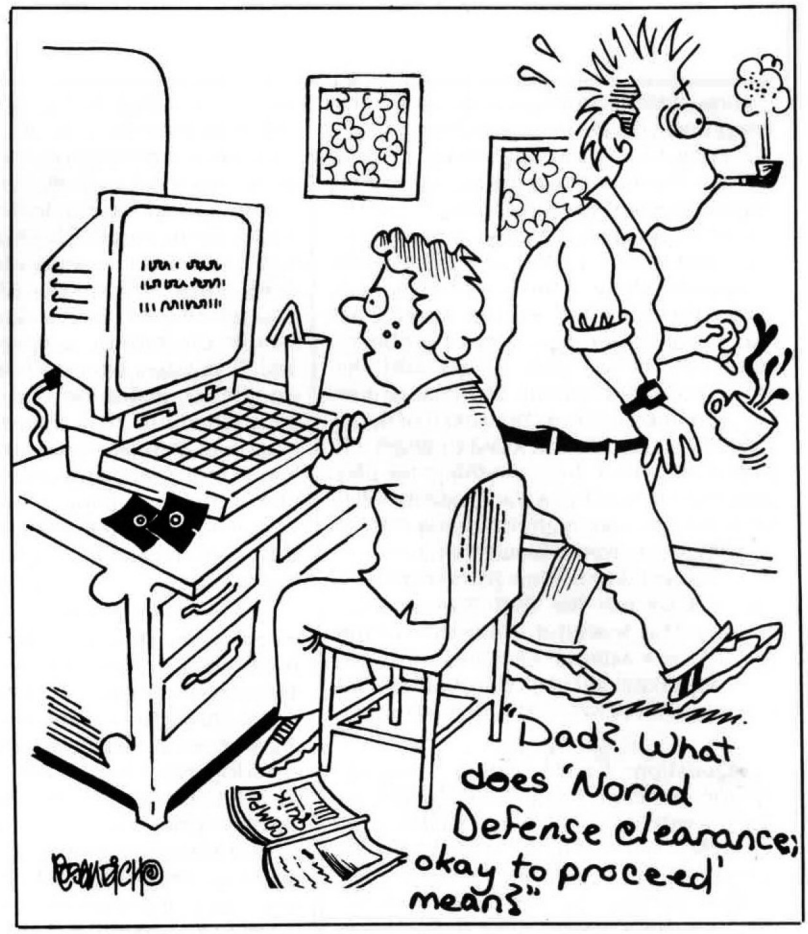
Tough Dogs

When you consider how many things could prevent a program from being run successfully under ZPC, it is incredible that any programs work at all. As it turns out, most programs can be made to work under ZPC, if you spend enough time trying. And you will find that most programs only require simple patches for video port accesses or the graphic character table.

There are times, however, when extraordinary measures may be necessary. If you have gone through a program and patched everything in sight that looks illegal and the program still refuses to run, you may want to reconsider just how important it is to use this program on the Z-100. If you still want to proceed, then you can try to single-step through a program, using DEBUG or SYMDEB, until you find the cause of the problem. This can (probably will) be a laborious task which requires real programming expertise. Even so, if a program uses overlays, is self modifying, or is memory resident, you may be destined for failure. *

Z-100 Survival Kit #12

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655



This issue I'll devote the entire column to questions and answers I've received from Z-100 users. Here we go . . .

* * *

Question: *How can I write a keyboard input routine in Assembly, 'C', or Pascal, which will respond to the non-ASCII keys on the keyboard. Also, how can I determine if the SHIFT, CONTROL, and CAPS LOCK keys are down?*

Answer: This is a pretty broad question, about which an entire column could be written. I'll try to give a few pointers here. First of all, if you are using the 'C' language, or Pascal, or any high level language, standard library functions should be available to read the keyboard. Today's modern language implementations usually give the programmer several different types of character input routines, each of which may be useful in different situations. Some languages attempt to filter unwanted characters from the input stream (BASIC is notorious for this) and make reading non-ASCII characters difficult. But most languages allow you to read the actual characters which are read by the DOS input routine. Obviously, if you are using Assembly Language, you won't have any filtering problems since you will be using DOS function calls for keyboard input.

In any case, if you need to use non-ASCII keys in your program, it is usually easier if you disable the key expansion feature. This is done by sending an ESC ? y (hex codes 1B 3F 79) to the console. When key expansion is disabled, a single key code is returned for each key that is pressed. (See the Z-100 User's Manual, Appendix B, for key codes and escape sequences.) The default DOS mode (key expansion enabled) causes many of the keys to return an ESC code, followed by another key code. It is easier to handle special non-ASCII keys when they only return one unique key code. If your program does disable it again before exiting member to enable it again before exiting to DOS. This can be done by sending ESC ? x (hex codes 1B 3F 78) to the console.

In its normal ASCII scan mode of operation, the keyboard encoder considers the SHIFT, CONTROL, and CAPS LOCK keys to be modifier keys, and they do not generate a key code when they are pressed. This makes it impossible to monitor the state of these keys, as is done in an IBM-PC. There is a way that you can tell if the keys are down, and that is by using the up/down (event driven) mode of the keyboard. In up/down mode, the SHIFT, CONTROL, and CAPS LOCK keys generate separate up and down codes when they are pressed and released (just like any other key). In order to take advantage of this feature, your entire keyboard input routine would need to be written to use up/down mode. This would be a challenging project which is beyond the scope of this question and answer section. Even using up/down mode, you will not be able to tell whether it is the left or right SHIFT key that is down, like on a PC clone. The keyboard encoder chip itself would need to be reprogrammed to make this possible.

* * *

Question: *I'm using the BIOS__CONOUT routine to output text to the Z-100 screen. But I'm having trouble positioning the text on the screen. I've tried changing the HORZ__CHAR and VERT__LINE variables in the MTR-100 data segment before writing text to the screen, but the text still appears right where it left off the last time. Plus, when control returns to DOS, the cursor is still at the same spot it was before my program took over. What do you suggest?*

Answer: I'm not sure why the text location doesn't change when you manipulate the HORZ__CHAR and VERT__LINE variables in the MTR-100 data segment . . . I would need to look at your code. The D__CRT and S__CRT routines in the MTR-100, which are eventually called by the BIOS__CONOUT routine, do use these variables to determine the next character position on the screen. Likewise, the problem with the cursor is also mysterious. Whenever BIOS__CONOUT is called to output a character, the cursor is automatically updated. There is more to this than meets the eye. I suspect that you may be using a wrong value for the offset to the variables in the MTR-100 data segment, or you may be using some other routine to output the characters on the screen. For instance, if you are calling the MTR-100 DFC (Display Font Character)

routine directly from your program, the symptoms you describe would result.

Sorry I can't give definitive answers without seeing some code, but there is an important point to note here. The MTR-100 ROM program is a very complex program that makes it possible for the Z-100 to operate. Many of the functions provided by the MTR-100 are intertwined, so that changing one thing causes problems elsewhere. If you plan to get into the MTR-100 data segment and change values, you need to know how each of those variables is used by the ROM program. Of course, it won't hurt anything to play around a bit. But if you want to write reliable software, you might be better off doing things the 'well-behaved' way.

The best way to change the cursor location is to use the ESC Y escape sequence. This is well documented in the Z-100 User's Manual. Using ESC Y automatically takes care of all the overhead associated with cursor and text positioning.

* * *

Question: *I have written a pop-up memory resident utility which needs to save the existing screen when it takes control. When 64K video RAM chips are installed, there is supposed to be enough memory for two pages of video memory, so I thought I might avoid using a big chunk of system RAM for buffer space by using this second page of video memory to save the screen. The problem is that when I move the contents of the screen memory from the beginning of each video plane to the top half of the video plane, it overwrites the existing screen starting about 16 lines down. Can you tell me what I'm doing wrong? Here is some of the code I'm using to save the screen:*

```
        MOV     AX, 0E000h              ; point to green video plane
        MOV     DS, AX                  ; point DS and ES to video plane
        MOV     ES, AX                  ;
        MOV     SI, 0                   ; SI points to first page
        MOV     DI, 8000h               ; DI points to second page
        MOV     DX, 225                 ; will move 225 scan lines
NEXTLINE:
        MOV     CX, 80                  ; 80 bytes at a time
        CLD                             ;
        REP     MOVSB                   ; move 80 bytes now
        ADD     SI, 48                  ; skip to next scan line
        ADD     DI, 48                  ;
        DEC     DX                      ; decrement scan line count
        JNE     NEXTLINE                ; repeat until done
```

Answer: I'll give you an 'A' for effort, but your description of the problem, and the code sample, indicates that there are a lot of things you don't understand about the Z-100 video layout.

First of all, let me say that you are right in assuming that 64K video RAM chips will provide more than enough memory for two pages of video memory. A simple calculation shows that 225 scan lines of 80 bytes each only takes 18,000 bytes of memory, so theoretically at least, there should be enough room for three pages of video using 64K chips. But unfortunately, the way the Z-100's video memory is organized precludes this possibility. First of all, each 80 byte scan line includes 48 additional bytes at the end which are not used. This makes the math faster for scrolling and address calculations, since each line is 128 bytes long. You have taken this into account in your sample program by adding 48 to the DI and SI registers after each line is moved. Another complication in the video mapping scheme is that each group of nine scan lines (representing a text row) is followed by 7 non-displayed scan lines. This makes the beginning of each text row start at an even 800h byte boundary. This was also done to accommodate faster text scrolling.

When you take this odd video mapping scheme into account, you find that the normal 25 line text screen on the Z-100 appears to take 51,200 bytes of video RAM. This figure is arrived at by multiplying 16 scan lines/text row (9 displayed, 7 non-displayed) by 128 bytes/scan line by 25 text lines. The question naturally arises: "How can you address up to 51,200 bytes of memory, in a system that may only have 32K RAM chips?" The answer lies in the way the video memory is mapped between the CPU and the CRT-Controller. This subject is way too complicated to discuss here. Suffice it to say that the video RAM mapping module allows the CPU to see video memory in a way that is convenient for scrolling, while at the same time allowing the CRT-Controller to access the memory in a manner appropriate to screen refreshing. For those of you who want to know more, the Z-100 Technical Manual has an in depth (but barely understandable) discussion of this mapping scheme.

The end result of all this video memory mapping business is that you can't simply move 32K bytes from the start to the end of each video plane. It is possible to write to 'page one' or 'page two' of video memory, but it is done by changing the value of the video address latch. In other words, the second page of video memory is accessed at exactly the same memory address as page one, but the value of the address latch changes the actual location for the memory access. This is pretty deep stuff which needs to be explained at length. If there is interest, I'll cover it more thoroughly in a future installment of Z-100 Survival Kit.

But let's get back to your original problem of saving the screen for a pop-up utility. Even if you knew everything there was to know about saving the existing screen in the second page of video memory, this may not be the best way to go. What if your memory-resident program was popped-up while you are running a program that uses interlaced video? Since the high-resolution screen used in interlace mode uses more video memory, there would not be enough room to save the entire screen in video memory. You would also be precluded from using your pop-up utility with programs that make use of two pages of video memory. I'd say the best all-around solution, especially if you are writing your program for others to use, would be to byte the bullet and simply reserve a block of system RAM large enough for the screen buffer.

* * *

Question: *I would like to learn more about Assembly Language programming, and using MS-DOS function calls on the Z-100. There are lots of books available that cover this topic for PC compatibles, but nothing for the Z-100. Any suggestions?*

Answer: Any book which covers Assembly Language programming for the IBM-PC will be useful for Z-100 users, as well. The key here is that both the IBM-PC and the Z-100 have an 8088 CPU, and they both use the MS-DOS operating system. All of the Assembly Language instructions, and all of the DOS functions are identical between the IBM-PC and the Z-100.

A good place to start is by reading Pat Swayne's "Getting Started with Assembly Language" series in the last few issues of this magazine. Or, if you prefer a book, these are good tutorial introductions:

Assembly Language Primer
for the IBM-PC & XT
Robert Lafore, © 1984 the Waite Group
Published by New American Library

Peter Norton's Assembly Language Book
for the IBM-PC
Peter Norton and John Socha, © 1986
Brady Communications Co.
Published by Prentice Hall Press

Even though these books say they are for the IBM-PC and XT, almost everything in them is also applicable to the Z-100. Another valuable reference is:

The iAPX88 Book
Intel Corporation © 1981 or later
Published for Intel by Reston Publishing Company

More advanced books about MS-DOS and Assembly programming would include:

Advanced MS-DOS
by Ray Duncan, © 1986 Ray Duncan

Published by Microsoft Press

MS-DOS Developer's Guide
John Angermeyer and Kevin Jaeger, © 1986 the Waite Group
Published by Howard W. Sams & Company

Tricks of the MS-DOS Masters
John Angermeyer, Rich Fahringer, Kevin Jaeger, and Dan Shafer
© 1987 the Waite Group, Published by Howard W. Sams & Company

MS-DOS Papers
Waite Group, © 1988 the Waite Group
Published by Howard W. Sams & Company

Memory Resident Utilities, Interrupts, and Disk Management with MS-DOS
Michael Hyman, © 1986 Michael I. Hyman
Published by Management Information Source, Inc.

There will be portions in these, and other books, which will not be applicable to the Z-100. Many authors also include a discussion of how to access the IBM-PC BIOS routines from Assembly Language. This type of info won't do you any good because the Z-100 has BIOS routines which are different from the IBM-PC.

Information about the Z-100's BIOS can be found in the Heath MS-DOS Programmer's Utility Pack (sometimes referred to as the PUP). If there is any book that can legitimately lay claim to being the Z-100 programmer's bible, this is it. Any serious Z-100 programmer MUST have this reference guide.

⁂ ⁂ ⁂

Question: *I know that the Z-100 maintains a font table in system RAM which it uses for text characters. I also know that this table is created by copying the table in the MTR-100 monitor ROM. Can I replace the RAM version of the table with my own character font?*

Answer: Sure, that's why the font table is moved into RAM memory to begin with. As a matter of fact, if you have an ALTCHAR.SYS file on your boot disk, it is already being done for you. The Z-100 version of DOS will automatically look for a file by that name in the root directory, and if found, that font will be copied into memory and used for screen text. This feature is commonly used to gain access to the H-19 style block graphics characters, but any font could be copied to ALTCHAR.SYS for loading at boot time.

After you have booted up, the font may be changed at any time by using the DOS FONT program. This program will allow you to change fonts, to redesign an existing font, or to create an entirely new font. The FONT program also lets you change the keyboard mapping. See your Z-100 MS-DOS manual for more information about this program.

If you want a program to load a special font, this is also possible. One way would be to have your program EXEC the DOS FONT program to load the new font. Or your program can load the font itself. In order to create and load your own font, you'll need to know the exact layout of the font table. This is shown in the source listing for the MTR-100 ROM (which is included with the Z-100 Technical Manual set). To find the start address of the font table in memory, follow this procedure:

1. Find the start address of the MTR-100 data segment. This is stored as a double-word pointer at address 0:3FCh in the interrupt table.
2. The double-word pointer to the start of the font table is at offset 6Fh in the MTR-100 data segment.

⁂ ⁂ ⁂

Question: *There are beginning to be a lot of cheap 8 inch drives available. Which ones will work with the Z-100?*

Answer: About the only thing I can say for sure is that the 8 inch drive interface of the Z-207 floppy controller is designed for use with a standard 50 pin Shugart compatible (SA801 or SA851) drive. I expect that other drive manufacturers could tell you if their drive meets this specification. The controller and BIOS software will support single- or double-sided drives. Be careful when you go shopping, because some of the older Heath drives (lovingly referred to as boat anchors) are not double-sided, double-density, and have a limited storage capacity. The more recent style drives allow 1.25 megabytes of storage, and still command a fair price (although that situation will change dramatically as more people switch to 5-1/4 and 3-1/2 inch high density drives.

⁂ ⁂ ⁂

Question: *How do I calculate the video address for a pixel on the screen, or for a text character on the screen?*

Answer: The video RAM offset for a text character at row R, column C, may be calculated as follows:

```
VOS = (R * 2048) + C
```

This equation assumes that the row and column indices begin with row 0, column 0, and that the Z-100 is programmed for a standard 640 × 225 resolution screen with 25 text lines. The VOS number calculated above is the offset to the top byte in the text character. Each text character consists of 9 bytes of displayed information. Each successive byte of the character design is offset 128 bytes. This means that the ninth (and last) byte of the character design will be written to VOS +1024. Keep in mind that each plane of video memory may need to be updated independently, depending on the foreground and background colors of the font, and the status of the video control register bits.

When the MTR-100 ROM program writes text to the screen, it actually writes 11 bytes of information for each character (to the green plane only). The additional two bytes are not displayed on the screen. Byte number 10 is the ASCII code for the character, and byte number 11 are the character attributes.

The video RAM offset for a single pixel at coordinate X, Y may be calculated as follows:

```
TR = INT(Y / 9)
VOS = (TR * 2048) + ((Y - (TR * 9)) *
   128) + INT(X / 8)
```

Additionally, the bit number of the pixel in the byte may be calculated as:

```
BIT = 7 - (X - INT(X / 8) * 8)
```

Again, we are assuming that a standard 640 × 225 screen is being used, and that the origin for the X and Y coordinates is 0,0 at the top left of the screen. We're also assuming that bits are numbered with bit 7 as the most significant bit in the byte.

There are more efficient ways of doing the arithmetic for these calculations. Typically, shift operations and modulo arithmetic should be used instead of multiply and divide instructions. But you get the idea.

⁂ ⁂ ⁂

Question: *The source listing for the MTR-100 ROM, and the documentation in the Programmer's Utility Pack, show quite a few different variables in the ROM data segment. I understand what many of them do, but some are elusive. There are some that I can't find referenced in the ROM code. What are they used for?*

Answer: Most of the variables in the MTR-100 monitor ROM data segment are used by the ROM to hold system status flags, address pointers, or to pass information to other routines. I'm sure Heath didn't plan for any of them to be changed by user programs, although clever programmers can do tricks by playing with them. Modifying the MTR-100 variables is definitely not for the faint-hearted, however.

Not all of the variables are used in current versions of the MTR-100 ROM. For instance, many of the variables in the COLOR structure are not used. Apparently, these variables are holdovers from earlier versions, or might even have been included in a trial version, and then never used. Being a programmer myself, I know that it is easy to forget to go back and remove unneeded trash before a product is released.

Not only are some of the documented variables unused, but there are many undocumented variables that are used. Only the variables up to about offset 300h are documented by Heath, but the data segment is 400h bytes, almost all of which is used for some purpose or another. Most of the undocumented variables are simply temporary storage locations which are used to hold loop indices or transient results.

⁂ ⁂ ⁂

Question: *The Z-100 Technical Manual gives a procedure to clear the Z-100*

screen by using the CLRSCRN bit of the video control port. One of the steps involves waiting for 16.7 milliseconds to elapse. How can this be done?

Answer: There are two basic ways you can delay for the right amount of time, as described in the Technical Manual. You can use the timer, or you can wait for two consecutive vertical sync pulses. The way you count the video sync pulses is by hooking into interrupt vector 5Ah. This is a software interrupt generated by the BIOS when the vertical retrace interval begins. Your program should hook into interrupt 5Ah, wait for two interrupts, and then restore the interrupt vector to its original value.

The other way of forcing a delay is by using the system timer. It would probably be easier to use the timer that MS-DOS maintains, than to access the interval timer itself. This can be done by getting the DOS time, and then looping until 16.7 seconds have elapsed.

I personally think that either of these techniques is overkill when it comes to clearing the screen. It is easier, and still fast enough to simply let the 8088 CPU do the screen clearing by writing zeros to all memory locations. If you are willing to write code which allows your program to continue executing while it waits for the 16.7 seconds to elapse, then that's a different story. But if you're going to loop and wait anyway, you might as well just clear the screen manually, and forget about the CLRSCRN feature of the video control port.

Here is a sample code fragment which will clear the Z-100 screen (all three video planes) regardless of whether you are using normal or interlaced video.

```
IN      AL, 0D8h          ; get video port status
MOV     AH, AL            ; save status
AND     AL, 8Fh           ; enable all multiple access bits
OR      AL, 80h           ; enable CPU access of video RAM
OUT     0D8h, AL          ; so all video planes are written
MOV     CX, 0E000h        ; get green plane
MOV     ES, CX            ;
MOV     DI, 0             ; begin at start of video plane
MOV     CX, 8000h         ; will clear 8000h words of memory
CLD                       ;
REP     STOSW             ; do it to it
MOV     AL, AH            ; get original video port status
OUT     0D8h, AL          ;
```

If we ignore the possibility of interrupts and video arbitration, this routine will clear the screen in about 60 milliseconds, which is fast enough for me. I mean how often do you clear the screen in a program anyway? The routine could be further optimized by only clearing the displayed portion of the video memory, at the expense of some additional code.

* * *

Question: *Is anyone out there still willing to repair Z-100 computers?*

Answer: That's getting to be a real good question. At the risk of offending Zenith Data Systems (referred to as *.* on the January cover of REMark), I know that many of the Heath/Zenith Computer & Electronics Centers are shying away from Z-100 repairs. I'm in a position where I receive a lot of feedback about this type of thing, and I have received numerous reports of service being refused for such minor technicalities as the existence of an FBE memory expansion or a CDR speed-up kit. In other words, some of the Heath/Zenith centers are using the existence of non-Zenith modifications as an excuse to refuse service. This didn't use to be the case. But it seems to depend almost entirely on the management at the local store. I've also gotten reports of Heath Stores charging enormous up front fees (like $85.00) just to open the case and look. If you have a good relationship with the local Heath Store, and you can stand all those Apples looking at you, then by all means that's where you should take your Z-100.

Another alternative, providing you or a friend doesn't repair computers, is to try an independent Zenith Data Systems dealer. Again, this is a hit and miss situation, and you'd better check out the local dealer before you trust him with your baby. Very few ZDS dealers even do any service on computers. Don't even bother calling the local Zenith TV shop. Some of the ZDS dealers that have been serving the Heath community for years would be a good bet — you know — the ones who have advertised in REMark over the years. Good examples are First Capital Computer, Payload Computers, and Quikdata, Inc. I'm not sure about First Capital or Payload, but I do know that Quikdata will repair Z-100s, even if you didn't buy it from them originally.

The only other alternative, and one which is getting to be more popular as time goes by, is to simply junk your old Z-100, and buy another one. The going price for used dual-floppy Z-100s seems to be about $250. Of course, most used models are loaded with goodies, so the price might be higher for a particular system. And in the future, the price is going to go through the floor as bunches of Z-100s start coming back into the private sector through government auctions. Yep, I'd say if there is anything major wrong with your Z-100, you might be just as well off buying another, and keeping the old one for spare parts. ✱

Z-100 Survival Kit #13

Paul F. Herman
3620 Amazon Drive
New Port Richey, FL 34655

I Just Bought a Z-100 . . . Now What?

This month's column is going to be a guide for all the folks who are buying used Z-100s for the first time. It will answer questions which are elementary for most of us, but which are of paramount importance to someone who has never seen, or heard of, a Z-100 computer before. The buyer of a used Z-100 is immediately at a disadvantage, because the machine may not come with the proper documentation or software. In fact, it may not even work. I've been fielding about two or three calls a week from people who just bought a Z-100 and don't know what they've got, or what to do with it.

Where Do They All Come From?

Many used Z-100s are purchased from folks who are upgrading (yes, I hear the hisses . . .) to bigger and better machines. Generally, in these cases, the purchaser is in pretty good shape because the machine will come with a pile of software and technical literature accumulated by the first owner. And if there are any difficulties, the new owner can usually ask the seller for help.

But MOST used Z-100s being purchased right now are bought at an auction or surplus outlet for a price ranging from $50 to $250. The buyers typically think they are getting an IBM-PC compatible computer (aren't all computers IBM compatible?) and don't know what to do when it won't boot their borrowed copy of PC-DOS. A large number of Z-100s are now beginning to show up at government auctions, and are being sold to the highest bidder — without any software — not even DOS.

Take an Interest in New Z-100 Owners

Many of you who have read up to this point are beginning to say that this doesn't have anything to do with you, because you already have a Z-100, and you are not a novice. That may be true, but you should realize that these new Z-100 owners need our help to figure out how to use their new computers. And we need their continued support to extend the useful life of the Z-100. If a new user can't get his Z-100 working and doing useful things, he will throw it away and buy a PC clone instead. On the other hand, if we help him discover the capabilities of the Z-100, he will continue to use it and may contribute to the Z-100 community in the future. When you consider that tens of thousands of Z-100s are owned by the government, and will be auctioned off in the next few years, the level of help and support the new buyers receive may have a drastic effect on the future of the Z-100.

The Typical Scenario

My company (Paul F. Herman Inc.) has become something of a clearing house for Z-100 information. The Heath Users' Group refers most of the questions they receive about the Z-100 to us, and Heath/Zenith is referring many of their Z-100 technical questions to us as well — especially those coming from the military.

The typical caller starts out by saying that he just bought a Z-100 (or several) for next to nothing, and needs some information. The questions these new buyers ask are not hard to answer, but they do take some time to explain properly. And I get asked the same questions over and over, several times a week — thus the reason for this edition of Z-100 Survival Kit.

I would like to address some of the more commonly asked questions in this column. And then I would like to finish by going through a step-by-step procedure to help you figure out if your Z-100 is operating properly, and to help you get your system on line — even if you don't have any documentation.

Is the Z-100 an IBM-PC Compatible Computer?

In a word, no. But before you get discouraged, some additional explanation is in order. When I say that the Z-100 is not IBM-PC compatible, this is to say that it will not run all of the software that you can buy for a PC clone. However, both the Z-100 and the IBM-PC use the MS-DOS operating system, and the same CPU chip, so many programs WILL run on both machines. These programs are generally referred to as "Generic DOS" programs. There are very few commercial software programs which fall into this class, but there are many public domain and shareware programs that are generic DOS, and

which will run on the Z-100.

Can the Z-100 Run IBM-PC Software?

There are several approaches to using IBM-PC software on the Z-100. First of all, many "PC programs" are really not IBM-PC specific, but are programs which will run on any MS-DOS computer, including the Z-100 (see previous question). In order to fall into this category, a program has to display only text (no graphics), use only ASCII keyboard input (text character keys or control codes), and access peripheral devices using MS-DOS function calls. It will be almost impossible to tell if a program is a generic DOS program without trying it.

For PC-specific programs, there is still hope for running them on the Z-100. The most economical approach, and the logical first alternative, is to try HUG's ZPC software emulator program. This program allows you to use a surprising number of IBM-PC programs on the Z-100. Many programs will run under ZPC without any problem. Others may require modifications, called patches, before they will perform correctly. For this reason, the ZPC software solution may not be a good choice for casual users unless the application programs you need will run without patching. ZPC may be ordered directly from the Heath Users' Group (see phone numbers in the front of this magazine). The ZPC program requires at least 768K of RAM memory (the full load) for the most successful emulation.

At least two companies have developed hardware solutions for the Z-100 PC compatibility problem. Gemini Technologies has a product called the Gemini Emulator Board, and UCI Corporation manufactures the UCI Easy-PC Emulator. These hardware modifications to the Z-100 allow just about any PC compatible program to be run on the Z-100, as long as it uses text or CGA graphics modes. Availability of these hardware emulator systems may be limited — check some of the suppliers who advertise in this magazine for current ordering information.

What Version of DOS Does the Z-100 Use?

The latest version of DOS available for the Z-100 is v3.1. But keep in mind that this must be a version which is designed to work on the Z-100. You can't take your brother-in-law's copy of MS-DOS 3.1 and expect it to work on the Z-100. As far as I know, the only place in the world that still sells DOS for the Z-100 is QuikData Inc. (a regular advertiser in REMark). Other Zenith Data Systems dealers may have some copies too. Supplies are limited, so don't delay if you need to get a copy of DOS for your Z-100.

I am currently involved in negotiations with Heath/Zenith and Microsoft to try to make MS-DOS for the Z-100 available on a continuing basis — only the future will tell if this project is successful.

What Do All of the Rear Panel Connectors Do?

Here is a listing of each connector on the back panel, and a description of its use:

J1 — This is a female DB-25 connector which serves as a DCE (Data Communications Equipment) port. It was originally intended to be used as a serial printer port, although many serial printers are more conveniently connected to J2.

J2 — This is a male DB-25 DTE (Data Terminal Equipment) serial port. It may be used for serial modems, printers, or other devices. This port is roughly equivalent to the COM1 port on PC compatibles. Both serial ports in the Z-100 (J1 and J2) are similar, and differ primarily in the gender of the connector and the pin-outs. Most devices can be used on either port if you have a null-modem gender changer.

J3 — A parallel printer port which uses a female DB-25 connector. This is a standard parallel output port similar to the LPT1 port on PC compatibles.

J4 — A modular phone jack which is used as a light pen connector. Use of a light pen with the Z-100 will require special software which knows how to interface directly with the light pen.

J9 — This is a female DB-9 connector used for RGB video output to a color monitor. Most CGA compatible color RGB monitors should work okay with the Z-100, and should come with this type connector.

J14 — An RCA phono jack used for monochrome video output to a monochrome monitor. This jack will be missing on All-In-One models, since the composite monochrome monitor is built in.

J16 — If installed, this should be a 50-pin connector for attaching a Shugart compatible 8-inch floppy disk drive. However, if the Z-100 has previously been attached to a Bernoulli Box, tape backup, or other special equipment, J16 may be used as a 50-pin SCSI bus connector.

There are knockouts for many more connectors on the back panel of the Z-100, but stock machines will only have those listed above. The existence of additional DB-25 or other types of connectors probably means that a multi-port I/O card (Z-204) or other accessory cards are installed.

If one of the rear panel knockouts has been replaced by a small slide switch, the switch is most likely used to change the Z-100 between 4 MHz and 7.5 MHz operation.

Can I Add a Hard Disk to the Z-100?

Sure, no problem. There are a variety of different ways of adding a hard disk, and there are a number of vendors who still provide this type of support. Check with one of the vendors who advertises in this magazine for more information.

Can I Read and Write PC Compatible Disks with the Z-100?

Yes, both the Z-100 and IBM-PCs use the standard 360K double-sided, double-density format. These disks are interchangeable between machines. If you find that you cannot read disks created in a PC compatible machine, there are several things to check:

1. Make sure the PC compatible disk is a standard 360K format disk. Many PC computers (especially the ATs) use a high density 1.2 Mb format which cannot be read by the Z-100.
2. Check your version of DOS. If you are using MS-DOS v1 (also known as Z-DOS on the Z-100), you will not be able to read disks created with version 2 or above of MS-DOS. This is because version 2 and higher of MS-DOS uses 9 sectors per track, instead of the 8 sector format used by Z-DOS.
3. In some cases, inability to read a known-good diskette may be caused by hardware problems, such as a drive which is not aligned properly, or a bad controller board. If the Z-100 seems to work just fine with its own disks, but refuses to read disks created on other machines, you may have an alignment problem.

If PC disk compatibility is a primary concern, and you need to read high density or 3.5 inch formats, software is available for the Z-100 which will allow the use of these special floppy drives.

Getting Your Z-100 Going Without Any Documentation

Many Z-100s are being sold these days without any documentation or user's manual. It is easy to understand why the new purchaser would have difficulty figuring everything out. The remainder of this column is a step-by-step guide for getting a Z-100 up and running, with or without documentation.

Power-Up Check

The only thing you'll need for this check is the Z-100 itself, and a video monitor. If you have the "All-In-One" model, your monitor is built into the computer. If you have the "Low-Profile" model, you'll need either a composite monochrome monitor, or a CGA compatible RGB color monitor. Plug composite video monitors into jack J14. Plug RGB monitors into connector J9. Sorry, the Z-100 won't work with a TV set (former Commodore 64 owners ask this question from time to time).

Oh, there is one other thing you'll need — a power cord for the Z-100. You should have received this with the computer, but the cord is removable, so it could be missing. The power connector

on the back of the Z-100 is a standard type used by many computers. If you need a cable, try a local electronics parts house.

Now plug the Z-100 in, and turn it on. (Can't find the switch? Just stop right here — you probably should not be playing with computers.)

You should hear one or two BEEPs (depending on ROM version). You should also hear a noisy fan coming up to speed. No Fan noise or BEEPs? This probably means the power supply is dead — a very expensive problem — find the guy who sold you this thing before he disappears. Yes, the Z-100 has a fuse, but it is considered to be a non-serviceable part (no this is not a joke, it is Heath/Zenith's way of selling power supplies). At any rate, if the fuse is blown, you've probably got other problems, so best to get your money back, if possible.

If you hear the fan, but no BEEPs, then there is something wrong with the internal electronics. Could still be the power supply, but before you give up completely, take the cover off (see below for instructions) and try wiggling all the sockets and connectors to see if that corrects the problem. Still no luck? I guess you're up the proverbial creek without a paddle.

If the Z-100 does BEEP at you, all is well so far . . . skip the next section.

How To Take Off The Cover

If you need to get inside the Z-100, the cover is easy to remove, but only if you know how. Look at the back of the computer, and on each side you should see metal rails sticking out. Grab these and pull them toward the back of the machine and lift the lid at the same time. You may need to use a screwdriver or a pair of pliers to get them moving if they're stuck. The lid should just lift off.

After the cover is off, you'll have access to quite a bit of the internal electronics of the machine. You still won't be able to get at some of the boards without further disassembly, but I don't want to get too involved here. If you have the guts and the desire, go for it! Z-100s are easy to take apart and put back together — just make sure you remember which connectors go where.

The Hand Prompt

(**Note:** Some Z-100s with a Winchester disk (hard disk) installed, may be set for automatic booting. If this is the case, but you would still like to follow along with our discussion, try hitting the DELETE key during the auto-boot sequence, and you should be returned to the hand prompt.)

After the BEEPs, you should be able to see a prompt on the video monitor that resembles a hand with a pointing finger. If you don't, check your video connections again, and make sure you are using the proper type of monitor. If you still don't get anything, it sounds like problems with the video board in the computer. This could be something simple like a connector which fell off in the machine, or it might be more serious. If you want to have a look inside, proceed at your own risk.

If you get a video display, but it is distorted or out of sync, check the adjustments on your monitor first. If the problem can't be corrected by adjusting the monitor, you may have to fiddle with the jumpers on the video board. Jumpers are provided to select the vertical and horizontal sync polarity, and the type of RGB synchronization. Most monitors are pretty standard these days, so this should not normally be a problem.

System Information

Now it's time to find out something about the configuration of this Z-100. When the hand prompt is displayed on the screen, press the 'S' key. The computer should display a few lines of information about how much memory is installed, what type of video memory is used, and if the system is color or monochrome. It may also tell you what size memory chips are used, and if you have an 8087 numeric coprocessor installed.

Now, press the 'V' key. This will tell you what version of the monitor ROM you have. If you have a Winchester installed, or plan to add one later, you MUST have version 2.5 or later of the monitor ROM.

Press the HELP key. You should see a list of all the valid ROM commands. You can play with some of these if you like — won't hurt anything. The exact details of how to use most of them will be left as an exercise for the user. One, which may be particularly useful, is the TEST command. If this option does not appear on your list of commands, then you must have a real old version of the ROM — don't worry about it for now.

If this option is available, try it. You should get a second menu showing the different tests which are available. Options should include a disk read test, keyboard test, memory test, and power up test. There's no need to run these tests right now, but make a mental note that they are available, if needed.

Booting Up

If you've gotten this far, you can take confidence that most of the computer is functioning as it should. The only major parts that could still cause problems are the disk drives and controllers.

If there is a hard disk (Winchester) installed in the Z-100, try just typing 'B' and RETURN to see if the Winchester is set up as the default boot device. If the DOS sign-on message appears, you're home free — the previous owner must have left the system software on the hard disk.

If your system does not have a Winchester, or if the Winchester boot attempt failed, we'll have to boot from a floppy disk. Find your MS-DOS (or Z-DOS . . . Ughhh!) distribution disk, and insert disk #1 in floppy drive A. Drive A is usually the one on the left (systems with full-height drives), or the one on top (All-in-One Z-100s, or systems with half-height drives). If you have a Winchester Z-100, you only have one to pick from, and it must be drive A.

Now try typing 'B', followed by RETURN. Or, if you have a Winchester system, you may have to type 'B', then 'F1', then RETURN. The drive A access light should come on, and the system should boot up and display the DOS banner.

If the computer waits for a long time and then displays "DEVICE ERROR", you may have hardware problems with the drive or controller board. If the computer just hangs forever, crashes back to the hand prompt, or does other crazy things, you're probably trying to boot with an improper version of DOS. Remember, you must be using Z-DOS or MS-DOS for the Z-100! If you get the message "NO SYSTEM", this means the disk you are trying to boot is not bootable. If you have an unlabeled two-disk set of DOS disks, try the other disk.

Configuration

If you have successfully booted DOS, you're just about home free. One other thing that will be necessary before you can use any printers or other peripheral devices is the DOS configuration. This is not normally necessary with PC compatible computers (or is done with the MODE command), but on the Z-100, you MUST configure DOS for the devices you will use.

Find the CONFIGUR program on one of your DOS disks. Run this program and follow the instructions. Typically, you would want to configure DOS to use a parallel printer as device PRN, and maybe a serial printer or modem as device AUX. Before exiting the CONFIGUR program, be sure to write the changes to DISK and MEMORY. This is an option on the main CONFIGUR menu. This configuration process must be done for each bootable DOS system disk you use, including each bootable hard disk partition.

To check out the configuration, try just copying some text to the printer using DOS. This can be done as follows:

1. At the DOS prompt, type: COPY CON PRN . . . followed by a RETURN.
2. When the cursor goes to the next line, type in some characters, like "Testing 1,2,3" and hit RETURN.
3. Enter a Control-Z character. This is done by holding down the Ctrl key and hitting 'Z' at the same time. Hit RE-

Continued on Page 24

Continued from Page 9

TURN.

4. The text you entered should be printed on the PRN device. You can also do this for the AUX device.

Accessing an Already Bootable Winchester

At this point, you're basically in business. You should now be able to use any of the programs that you have for the Z-100. But if you have a Winchester system, and you were not able to boot onto the Winchester, there is still work to be done.

Try typing 'B', followed by 'F3', a colon (:), and then your name, followed by RETURN. What this does is tells the computer that you want to boot-up ('B') from the Winchester ('F3') onto the partition with your name. Needless to say, it will not find a partition with your name (unless you're name is something like Ms. Dos).

You should receive a message that says "Error — Partition Not Found. Hit RETURN to continue". Follow instructions, and hit RETURN. You should now see a list of the valid Winchester partition names. Try booting to each of the partitions that are listed by using the procedure above, but use the partition name, instead of your name. Hopefully, you will find one that is bootable.

If you don't get a list of partition names, this generally means that the Winchester needs to be prepared from scratch. This involves running the PREP program, and the PART program. These utilities are included with the Heath/Zenith Winchester utilities, and their use is a bit beyond the intended scope of this month's Z-100 Survival Kit column.

Accessing Winchester Partitions

If you have been unsuccessful at booting onto the Winchester, you will need to access it by booting from a floppy drive. Make sure you have gone through the procedure described above, and written down the names of the partitions.

To access the Winchester partitions, you will need a Heath/Zenith utility program that assigns drive letters to the partitions. The name of the assignment program will be ASSIGN if you are using Z-DOS or MS-DOS v2. The name will be ASGNPART if you are using DOS v3.

Boot up on your system floppy, and run the assignment program using the following syntax:

```
ASGNPART Ø:pname d:
```

or...

```
ASSIGN Ø: pname d:
```

where:

0 = the Winchester unit number
pname = name of partition to assign
d: = drive letter to assign

The drive letter you assign should be E, F, G, or H. After executing this command, you should be able to get a directory of the Winchester partition. If not, this generally means that the partition has not been formatted, so you will need to use the DOS FORMAT program to format the partition. Remember to use the '/S' FORMAT switch for any partitions you want to be bootable.

Wrapping It Up

I hope this information will help you get your 'new' used Z-100 on line. After getting it working, one of the next things you should do is start a search for any documentation you can find about the Z-100. This would include the Z-100 Users Manual, Z-100 Technical Manual Set, MS-DOS reference manual, Programmer's Utility Pack, and back issues of REMark and SEXTANT magazines. There are lots of other options, DIP switches, and jumpers in the Z-100 which I have not even mentioned in this column. Many of these will have an effect on the way the system operates, and how useful it is for you.

Till next time . . . keep in touch!