"Z80MU" Z80 and CP/M 2.2 Emulator User's Guide

or

The Care and Feeding of Your Imaginary Z80

or

Fakeware For The Techie Masses!

A Guide to the Complete Z80 Emulator

COMPUTERWISE CONSULTING SERVICES, P.O. BOX 813, MCLEAN, VA 22101

Textformat angepasst an OpenOffice.org writer
25.11.2010 von Dr. Hehl Hans
Z80EMU.PDF

www.hehlhans.de/euroz80.htm

Describing the first of a two-part series of 8080, Z80, and CP/M-emulators for the IBM PC. This document describes Z80MU (version 2.1, Z80MU dated 10/30/85), a software emulation of the Z80 and CP/M 2.2. The second product - if it ever gets out the door - includes hardware emulation of the 8080, using the NEC V20 8088-compatible processor chip.

Program written by Joan Riff for Computerwise Consulting Services.
Joan Riff

COMPUTERWISE CONSULTING SERVICES, P.O. BOX 813, MCLEAN, VA 22101
"Z80MU" Z80 and CP/M 2.2 Emulator User's Guide

INTRODUCTION what is it?

Z80MU is a software emulator of the ZILOG Z80 processor, which runs on the IBM PC. It also provides an emulation of Digital Research's CP/M version 2.2 operating system.

It includes the following facilities:

Complete emulation of Z80 object code, including all six active bits within the Z80 Flags Register.

Emulation of CP/M 2.2, with the exception of hardware-specific functions.

Advanced commands for debugging Z80 software (eliminating the need for DDT.COM), including:

Illegal Opcode control (treat 'em as FAULTs or NOPs).

BDOS function trace.

Instruction TRACE and NOTRACE, with Z80 PC traceback.

Breakpoints.

Dump CP/M memory in HEX and ASCII.

Patch CP/M memory in HEX, decimal, binary, ASCII characters,
or ASCII strings.

Symbolic labels may be defined and used instead of CP/M addresses.

Z80 register and flag display and alter, including
alternate regs/flags and IFF1, IFF2, IMF, I, R regs.

CP/M memory move and find.

Intel HEX files can be properly read into CP/M memory,
as well as created from CP/M memory.

An "emulated terminal" for full-screen CP/M applications.

A full disassembler much like Ward Christensen's RESOURCE,
built right in, with:

Symbolic addresses.

Control breakpoints for Instructions, Bytes (DB), Words (DW),
Table of Words (DW), and Storage (DS).

Automatic label generation.

Comments associated with Z80 addresses.

Online help summaries.

SUBMIT file support (built right in - no need for SUBMIT.COM).

Access to all PCDOS programs and commands.

CP/M software reads and writes PCDOS files, and can be
organized with PCDOS directory structure (instead of
using CP/M's "user number" idea).

Z80MU (from now on called "the Emulator") can be used quite transparently to run CP/M applications on the IBM PC. It contains many advanced commands not found in CP/M, but there's no law that says that you have to use 'em all. By ignoring the advanced commands, you can run the emulated CP/M all day long and you'll swear that you're running CP/M.

You can also take advantage of the advanced commands and features, and have a user interface more powerful than that available with CP/M.

Advanced Z80 programmers can even ignore CP/M entirely and use the Emulator as a generic Z80 development tool for developing device controllers and other non-CP/M based Z80 code.

INTRODUCTION

At CCS, we use the Emulator to develop things like Z80-based hard disk controller card software, device-switching hardware, and even a few boring old standard CP/M applications. We have also used it to regenerate the source code for the 32K ROM in the Radio Shack Model 100 laptop computer, to figure out how it works.

The Emulator consists of a high-speed 8088 assembler subroutine (which does the actual Z80 emulation), and a 'C'-language main program which provides the CP/M-like interface to the user, the disassembler, the RESOURCE facility, and the rest of the advanced features.

Why Did We Do It?

At CCS, we have quite an investment in CP/M software. Most of that software consists of fairly esoteric software development tools, things that are hard to find in the IBM PC world. Furthermore, we continue to develop software for the Z80. The old chip just won't die, although nowadays it is used more in controller boards than as a primary system processor.

We wanted to keep these Z80 tools, yet apply our numerous IBM PC tools (especially some fantastic IBM PC editors like SPF/PC) to our Z80 development process.

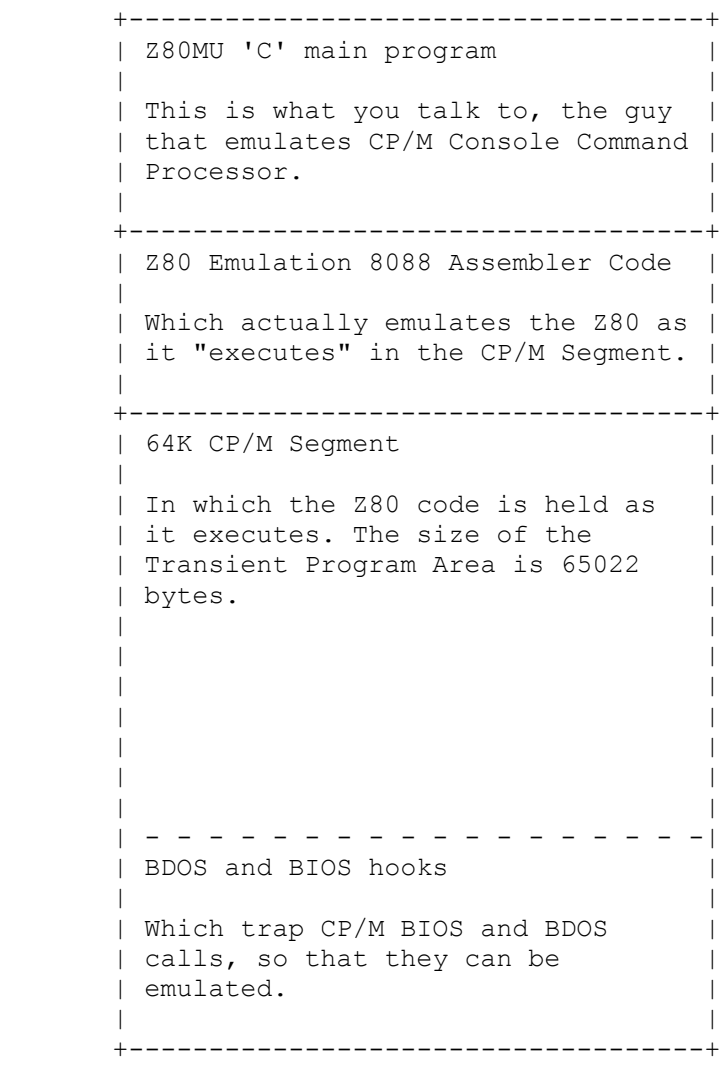
We were also keenly interested in creating source code for some object-only Z80 applications, with a view to converting them to 8088 assembler source code for re-assembly as native IBM PC programs.

We scouted around and discovered several software packages which supposedly allowed us to run CP/M on the IBM PC. Needless to say, they just couldn't cut it. The Heath User's Group (HUG) emulator, 80Mate by Vertex, and a few similar packages were considered. We weren't impressed by the speed of emulation, or by the accuracy of the emulation, or by the user interface. What's more, we kept bumping into more and more people who were in the business of Z80 development, and who kept badgering us to solve this problem "the right way" (whatever that means).

INTRODUCTION

How Does It Work?

When it runs under PCDOS, the Emulator looks like this:



The main program accepts the user's commands, and processes them. When and if it comes time to run a Z80 program, the Z80 code is loaded into the CP/M Segment, and the 8088 assembler code which actually emulates the Z80 is called to run the program there.

As the Z80 program executes, it accesses the outside world via calls to the BDOS and BIOS hooks. The code that is executed for a given BDOS or

INTRODUCTION

BIOS function actually resides within the 8088 assembler code that emulates the Z80.

What Does It Require To Run?

The Emulator runs on the IBM PC under PCDOS 2.x or above. The Emulator itself is about 93K in size. The first thing that it does when it runs is allocate 64K to be used as CP/M memory (the "CP/M Segment").

So you had better have 93K + 64K = 157K available when you run the Emulator. Note that the amount of memory required may vary as improved versions of the Emulator are released.

Some CP/M applications expect to control the screen cursor by sending control characters through the BDOS. For these programs, you may want to have the PCDOS ANSI.SYS device driver loaded via your CONFIG.SYS file.

How Accurate an Emulation Is It?

There are two aspects to the accuracy of the Emulator:

1) How accurately it emulates the Z80
2) How accurately it emulates CP/M

The Emulator emulates the Z80 almost perfectly, even down to an exact emulation of all six flag bits in the Flags Register. Even the untestable Half-carry and Add/Subtract flag bits are emulated. For faster execution, the Emulator ignores the two unused bits in the Flags Register, so these will not act exactly as they would on a real Z80.

Input/Output instructions (the IN's and OUT's) perform everything except the actual strobe of the I/O data lines. You can't very well have Z80 code accessing I/O addresses that mean something entirely different on the IBM PC. So the actual data transfer has been disabled. But any setup, auto-increment of registers, and flag effects have been emulated even for the IN's and OUT's.

The Z80 HALT instruction is used as a hook to return control to the main 'C' program, and as a call to the emulated CP/M BIOS and BDOS.

When it came to emulating CP/M 2.2, we took a less precise approach. We weren't interested in emulating the limitations of CP/M. We wanted to include many of the benefits of PCDOS. And we wanted to add many more "builtin" commands than were available with CP/M. We also demanded the largest possible TPA (Transient Program Area - the amount of memory

available to be used by a Z80 program). Yet we wanted to keep the interface very close to CP/M's.

So we decided to support "standard" CP/M applications, ones that stood a good chance of executing on a wide variety of CP/M systems and thus were hardware independent.

This we have succeeded in doing.

The user interface is just like CP/M's, so that someone used to CP/M will feel right at home.

The program interface (via the BIOS and BDOS) is exactly the same as CP/M's. There are some BIOS and BDOS functions that are hardware-specific. These are in general not supported. The differences are explained in a later section.

The handling of commands and command arguments (the command "tail") appears to the Z80 application exactly as it would on a real CP/M system. The default FCB at 5Ch is formatted with the filename implied by the first command-line argument, and the FCB at 6Ch with the second argument. The byte at 80h is set to the number of characters in the command tail, and is followed by an uppercase version of the command tail as typed by the user.

We have achieved a TPA size of 65022 bytes. This is more than is available on almost all "real" CP/M systems, including the Baby Blue Z80 add-on board (for the IBM PC) from MicroLog.

We have also built a terminal emulator into the Emulator, since many CP/M applications (especially those doing full-screen editing) assume that they are being run from an ASCII terminal.

All in all, the compatability of the Emulator is so good that we have been able to move almost all of our CP/M applications to the Emulator, and to have them run perfectly (although a tad slow).

Here is a partial list of CP/M applications that we have tested with the Emulator and found to run as they do on a "real" CP/M system:

ASM
LOAD
ED
DDT
DUMP
PIP
M80
L80
LIB

MBASIC
LASM
MAC
dBase II
WORDSTAR 3.0 & 3.3 with MAILMERGE (but not SPELSTAR!)
PMATE-80 rev. 3.02
Software Toolworks 'C' compiler
Telecon 'C' compiler
Chang Labs' MemoPlan

This is more a list of the software that we happened to have on hand than an exhaustive list of software that will run under the Emulator.

What WON'T Run Under the Emulator?

There are some CP/M programs (like STAT.COM provided by Digital Research) which are hardware-specific. These cannot be run under the Emulator, or must be run "carefully" to avoid functions that look to the hardware. STAT, for instance, starts out by interrogating the physical layout of the diskette. Since this is unsupported under the Emulator, STAT is immediately aborted by the Emulator.

WORDSTAR's SPELSTAR won't work, either. It tries to call CP/M's CCP directly. It's really quite sad. It goes to a lot of work to calculate just where in Z80 memory the CCP is, relative to the BDOS address held in location 0005h. Then it calls that address. Unfortunately, there's nothing there...

How Fast Is It?

Aye, there's the rub.

Because the Z80 used by the Emulator is an imaginary one whose instructions must be emulated in software, the effective speed of a Z80 program is considerably less than the speed of the IBM PC. One 12-cycle Z80 instruction, for example, may take from 47 to over 100 IBM PC cycles (depending on the instruction, its addressing mode, etc).

For detail on the effective speed of the emulated Z80 and what it means, see the description of the "speed?" and "howfast?" Builtin Commands later in this Guide.

We have not found the speed of the emulated Z80 to be entirely acceptable. Z80MU is the fastest software-based true Z80 emulator available today for the IBM PC. Yet we would like to have something that would run

INTRODUCTION

the standard CP/M utilities on an IBM PC at an effective cpu speed of at least 1 MegaHertz, and still have all of the subtleties (like flag updating) performed with 100% accuracy.

One solution is to run the Emulator on an IBM PC/AT with a hard disk. This is perhaps the best solution, but doesn't help those who don't have access to an AT.

An alternative solution is to use the NEC V20 processor chip in the IBM PC, and get rid of the Intel 8088. The NEC V20 is a tad faster than the 8088. More importantly, it performs full-speed hardware emulation of the Intel 8080 chip. Many CP/M utilities do not use the extended instructions offered by the Z80, and will run just fine on an 8080 chip. Such a hardware-based emulator would offer superior speed. It would, however, be limited to 8080 operation.

We are developing such an emulator. There is no firm release date.

In the meantime, you should equip your IBM PC with a NEC V20 chip to speed things up. We paid $16 apiece for ours, and there is no more cost-effective hardware upgrade for the IBM PC! See the ads in back of BYTE magazine for sources and prices.

How Do I Get My CP/M Programs Into The IBM PC?

Assuming that you want to emulate some CP/M applications on the IBM PC, the first realization is that these CP/M files don't already reside on the PC. What's more, they currently exist (by definition) on disks formatted for CP/M, not for the IBM PC. So they can't be read by a vanilla IBM PC using PCDOS. They must be copied to standard PCDOS disk files.

We have used several approaches, all with excellent results.

The first approach is to transfer the CP/M files straight from the CP/M disks to PCDOS disks using a utility that runs on the IBM PC and is capable of reading the foreign CP/M format. Such utilities include:

CONVERT (from Selfware, Inc. Fairfax, VA)
XENOCOPY (from Vertex Systems, Inc. L.A., Calif)

This approach is nice, when it works. The major disadvantages are:

Only certain CP/M formats are recognized by each of these utilities. Apple][and NorthStar Horizon CP/M disks, for

example, cannot be read on the IBM PC without special hardware.

You must have the CP/M disks at hand.

Another approach is to transfer the CP/M files to the IBM PC via a communications line. This is Joan Riff's personal favorite. If the CP/M system is at hand, then the two machines are direct-connected and cranked up to 9600 baud. If the CP/M system is not handy, then the transfer is made over the phone at whatever speeds the respective modems can handle. In either case, the CP/M system running BYE and XMODEM is controlled by the IBM PC running Crosstalk VI version 3.5 (from Micro-Stuf) and Joan Riff's excellent XMODEM (with CRC) Crosstalk RUN command.

When direct connected, files really fly across at 9600 baud.

The major advantages of this approach are:

Who cares what the CP/M disk format is? If the CP/M system can read its own files, then we can get them. This works very well when transferring Apple][and NorthStar Horizon CP/M files.

Public Domain CP/M software can be gathered just by dialing into a CP/M Bulletin Board or RCPM system. You need never know what hardware system is on the other end.

CP/M and PCDOS files are similar enough that we have never had to alter a file that was transferred using either of the above two approaches. We just download the files to the IBM PC and run 'em under the Emulator.

The biggest problem is remembering which files are PCDOS files and which are CP/M files. If you transfer a CP/M file called DUMP.COM, for example, from a CP/M system to the IBM PC's disk, you really do want to remember that it is a CP/M file (to be run with the Emulator) and not an IBM PC .COM file. If you accidentally invoke DUMP.COM from PCDOS, you will be unpleasantly surprised. The CP/M DUMP.COM file contains Z80 opcodes, which will be executed by the IBM PC as 8088 opcodes. Time to reach for the Big Red Switch...

You must run such CP/M command (.COM) files under the Emulator!

At CCS, we keep things straight by storing CP/M files under separate PCDOS directories. The "Z80PATH" environment string (explained elsewhere) makes this particularly convenient.

INTRODUCTION

The saving grace to this CP/M-to-PCDOS conversion is that it needs to be done only once for a given file. We spent quite a while transferring 10 megabytes of CP/M files to the IBM PC. But we need never do it again. Now we just run everything on the IBM PC.

How Do I Run It?

The Emulator is just another PCDOS program. There are no arguments to give it. There is no syntax. Just type

```
Z80MU
```

and bingo! - you're in CP/M.

To make things easier, you may want to copy Z80MU.EXE to one of your PCDOS "PATH" directories (if you have any). If you don't have any PATH directories set up, then just insert the floppy that holds Z80MU, start the program, and then remove the floppy. You don't need it until you want to run the Emulator again.

The next section ("The PCDOS Environment") describes the PCDOS environment that applies to the Emulator. You may want to study it before you run the thing.

What Can Go Wrong?

The Emulator is as safe a program as ever you'll find on the IBM PC. You will probably never experience any problem with it.

There is one important thing to watch out for, however:

> If the Emulator itself is ever aborted, then you should immediately reboot your IBM PC.

Why? Because the Emulator must trap the IBM PC's BREAK interrupt. When the Emulator returns to PCDOS (via the "exit" command), it restores this interrupt the way it was before. If the Emulator never gets a chance to exit gracefully, then it never gets a chance to restore this interrupt. The thing is left pointing to now-dead code somewhere in the IBM PC's memory. This is bad news for you, and good news for that big red switch on the side of your PC...

Why might the Emulator abort? Well, there is always the possibility of an Emulator bug that we haven't found. But the most likely reason is a disk error that results in the familiar message:

Abort, Retry, Ignore?

If you select Abort, then you've just aborted the Emulator and left the BREAK interrupt in limbo. So reboot to be safe.

THE PCDOS ENVIRONMENT

The Z80 Emulator runs as a normal application program under PCDOS (version 2.0 and above). There are a few things that you should keep in mind, in order to get the most out of the Emulator.

PCDOS's use of COMMAND.COM

Certain Emulator commands ("dir", "!xxxxxx", etc) are handled by calling PCDOS to perform the associated operation. The first thing that PCDOS does when called in this way is reload its COMMAND.COM file from disk. To speed things up, you should make sure that PCDOS's COMMAND.COM file is in a RAMdisk, or on a hard disk. You can tell PCDOS where to find COMMAND.COM by using the "SET COMSPEC=" command in your AUTOEXEC.BAT file.

Certain versions of PCDOS (2.0 and 2.1, and maybe others) have trouble obeying the "COMSPEC=" command. They try to reload COMMAND.COM from the boot disk, regardless of the current "COMSPEC=" parameter. If you use one of these versions of PCDOS, then you may avoid problems by keeping COMMAND.COM always available on the boot drive. Alternatively, you may apply one of the public domain COMZAP patches to fix your copy of PCDOS.

The "Z80PATH=" Environment String

The Emulator has a facility which is equivalent to the PCDOS "PATH" command. It allows you to tell the Emulator where to look for Z80 command (.COM) files.

This facility is implemented by a new PCDOS environment string, called "Z80PATH". This string is a list of fully-qualified names of directories which are to be searched when the Emulator is looking for a .COM file to load and run. The various directory names must be separated with semicolon (";") characters, as follows:

```
SET Z80PATH=c:\cpm;c:\z80\mystuff;c:\
```

This example tells the Emulator to search for Z80 programs first in the directory "CPM" on drive C:, and then (if not found there) in the directory "Z80\MYSTUFF" on drive C:, and finally (if still not found) in the root directory of drive C:.

A Z80PATH string should be defined in your AUTOEXEC.BAT file, so that it is always present when you run the Emulator.

The trailing "\" character of each directory name is optional. If it is absent, a "\" character is automatically applied to the directory name before the name is used in the search.

The Z80PATH search order is used whenever an "unqualified" program name is used as a command to the Emulator. An "unqualified" program name is a legal filename (up to 8 characters) which:

1) has no drive ID on the front of it (no ":" character), and
2) has no directory names imbedded in it (no "\" characters), and
3) is not the name of an Emulator Builtin Command.

For example, let's say that you give the following command to the Emulator:

```
Z80 A>asm dump.aaz
```

The Emulator first checks to see if the command ("ASM") is one that it recognizes - a so-called Builtin Command (see the "Builtin Commands" section). If it is not, then the Emulator acts just like CP/M and attaches a .COM extension to the command, yielding "ASM.COM". It then looks in the current PCDOS directory on the current disk (in this case, drive A:) for a file by the name of ASM.COM. If it finds such a file, then it loads it into the CP/M Segment and runs it.

If the file is not found on the current drive, then the Emulator searches the various Z80PATH directories, looking for a file with the right name (ASM.COM). The directories are searched in the order that they appear in the Z80PATH string. The first ASM.COM file that is found is the one loaded and run.

If there is no Z80PATH string defined in the PCDOS environment, then the search stops with the current disk drive's current PCDOS directory.

If no matching filename is found after all of this, then the Emulator echos the command line

```
asm?
```

indicating that it doesn't know what you mean.

Note that no search takes place if the command is "qualified". A "qualified" command includes a drive ID or a pathname, such as:

```
Z80 A>b:asm dump.aaz
Z80 A>\bin\asm dump.aaz
```

In such a case, the Emulator tries only once to open the Z80 .COM file, using the exact name given. If such a file cannot be found, then the command fails as mentioned above.

The AUTOEXEC.Z80 File

When the Emulator first starts up, it automatically executes the following command:

```
Z80 A>SUBMIT AUTOEXEC.Z80
```

If there is no file by the name of AUTOEXEC.Z80 in the current directory when the Emulator is run, then an error message is displayed and the Emulator just waits for you to enter commands from the keyboard.

If there is such a file, however, then the Emulator reads its commands from that file, until EOF. See the "submit" builtin command for more details about submit files.

This is an easy way to automate the Emulator. At CCS, we use a different AUTOEXEC.Z80 file in each work directory in order to set up the particular environment that we want to work with. The AUTOEXEC.Z80 file within the Radio Shack Model 100 directory, for example, automatically reads in the 64K Model 100 image from disk ("read 0 model100.mem"), and the disassembler control file ("control read model100.ctl"). It also sets up the disassembler format that we want ("list include A O"). So when we start up the Emulator while within that directory, the thing comes up ready to do real work.

THE PCDOS ENVIRONMENT

I/O Redirection With The Emulator

The Emulator reads its commands from the standard input as defined by PCDOS. It writes its output to the standard output that is defined by PCDOS. So regular old PCDOS I/O redirection can be used when you start the Emulator.

For example, the following PCDOS command can be used to run the Emulator and capture all Emulator output to file OUTPUT.DOC:

```
Z80MU >OUTPUT.DOC
```

You may also append output to an existing file with:

```
Z80MU >>OUTPUT.DOC
```

And the following can be used to have the Emulator read all of its commands from the file INPUT.BAT:

```
Z80MU <INPUT.BAT
```

You may combine input and output redirection, as follows:

```
Z80MU <INPUT.BAT >OUTPUT.DOC
```

This is perfect for automating the Emulator. Some of the samples displayed later in this document were captured by redirecting the Emulator's output to a file, and then editing that file into this document.

There are a few things to bear in mind, however.

First of all, remember that there are several parts of the Emulator:

The main program, which reads your commands and in general acts like the CP/M CCP. It does all I/O via PCDOS, so it is subject to I/O redirection.

The actual Z80 emulator, which does no I/O at all.

The CP/M BIOS emulator. It does I/O at the IBM PC ROM BIOS level, so PCDOS never sees what's going on. CP/M BIOS terminal I/O goes through the emulated "terminal" inside the Emulator, and then straight to the IBM PC screen. So I/O redirection does not apply there.

> The CP/M BDOS emulator. The BDOS emulator does its I/O via PCDOS, so I/O redirection does apply to it, and to any Z80 programs that use BDOS functions for I/O.

Now the question arises: Will my CP/M application obey any I/O redirection that I specify when I run the Emulator?

The answer, of course, depends on your application.

Most "standard" CP/M applications do their I/O via BDOS functions. So these will obey your I/O redirection.

Full-screen editors, in general, use the BIOS instead of the BDOS, for a lot of very good reasons. So these will automatically be exempt from your I/O redirection.

When constructing a file to be read via input redirection, remember to include all characters that are to be read either by the Emulator itself or by the CP/M application. This usually means that your input file will be a jumbled mix of Emulator and application input.

Let's say, for example, that you have a CP/M application named TEST.COM that asks for your name, prints some silly message based upon your name, and then exits back to CP/M. A complete input file to execute that program would look like this:

```
test
Joan Riff
exit
```

If this text is saved on a file called AUTONAME, we can run the Emulator, tell it to run TEST.COM, answer TEST's question, and then exit the Emulator back to PCDOS by entering the following PCDOS command:

```
Z80MU <AUTONAME
```

Please remember that all Emulator input and/or all Emulator output is redirected at once. If you redirect the output only, meaning to enter commands from the keyboard, don't be real surprised if you can't see any of the Emulator's prompts. They are being written to the redirected output file, and not to the screen.

> NOTE: When you redirect output to a disk file, your input keypresses are supposed to be sent to the output file (not to the screen). Some versions of PCDOS, however, contain a bug

> that causes your keypresses to appear on the screen instead. We have seen public domain patches to fix this bug floating around the Bulletin Boards, but can't vouch for any of them.
>
> ALSO: Input that comes from Input I/O Redirection is not echoed, so you won't see it anywhere. Input from submit files, however, IS echoed.

Using The Keyboard

As mentioned previously, the Emulator reads its input from the standard input as defined by PCDOS. So if you don't redirect the input to the Emulator, then input comes from the keyboard.

The Emulator could have done its own direct keyboard and screen I/O. This would speed things up considerably. PCDOS is notoriously slow when it comes to writing to the screen.

We decided, however, not to circumvent PCDOS when writing to the screen and reading from the keyboard. The Emulator is slower as a result. But we gain a few conveniences as a result:

> We achieve something closer to true CP/M emulation, 'cause PCDOS automatically handles ^P and ^S/^Q in a manner close to CP/M's handling of them.
>
> We get automatic I/O redirection.
>
> We get PCDOS expanding macro keys and interpreting function keys.

The F3 key, for instance, can still be used to repeat the last command entered to the Emulator. The ESC key cancels the current input line. And F1 recreates the last command one character at a time. Other PCDOS keyboard conventions (like Ctrl-NumLock, Ctrl-PrtSc, and so on) are also handled by PCDOS in a way that we're all used to.

The addition of keyboard enhancers like CED and Sidekick can confuse things, so that the function keys don't act quite right. You'll just have to experiment with it.

If you want to copy screen output to the printer, then press ^P or Ctrl-PrtSc. A second press will turn printer echo off. Remember that such PCDOS redirection applies to Emulator output (like dumps, disassemblies, etc) as well as to the output of any CP/M applications being run under the Emulator that use BDOS functions for output.

If text is scrolling off of the screen too fast to read (not real likely, with PCDOS being as slow as it is), you can pause and restart it with ^S/^Q, or Ctrl-NumLock.

The actions of ^S/^Q, ^P and Ctrl-PrtSc may vary, depending on the particular CP/M application being run. CP/M's BDOS function number 6 (Direct Console I/O), for example, is handled by CP/M without it checking for ^S or ^P. The Emulator mimics this action.

Filenames!

CP/M filenames may contain certain characters that PCDOS objects to. In general, don't use the "\" or "/" characters in filenames, or the I/O redirection characters ">" and "<", and so on.

And watch out for PCDOS device names that are perfectly innocent filenames under CP/M. Things like "CON.ASM" will fool you.

THE CP/M ENVIRONMENT

This section describes the environment set up by the Emulator, under which your Z80 programs will run.

Emulated Terminal

If a Z80 program does character I/O by invoking CP/M BDOS functions (not BIOS calls), then its input and output come from PCDOS, and this section does not apply.

When a Z80 program does character I/O by calling the emulated CP/M BIOS (not using BDOS functions, but BIOS calls), then it is communicating with an imaginary, emulated ASCII terminal which is maintained by the Emulator. The Emulator interprets ASCII codes that are sent to this "terminal", and translates them into appropriate calls to the IBM ROM to control the IBM's display.

Most ASCII characters that are sent to the emulated "terminal" are displayable characters - letters, numbers, and so on. They appear on the screen for the user to read. Other ASCII characters - called "control sequences" - are used not to display anything, but to cause the "terminal" to perform special functions like clearing the screen, switching between high- and low-intensity, and so on.

The builtin "terminal" obeys VT52 control sequences, which are the same ones used by the Heath/Zenith H19 and H89 machines when in ZDS mode. They are as follows:

ESC H Homes cursor

ESC C Advances cursor 1 char to right. Stays on same line.

ESC D Backspaces cursor one char to left. Stays on same line.

ESC B Moves cursor down 1 line, staying in same column. Screen is scrolled if necessary.

ESC A Moves cursor up 1 line, staying in same column. No scrolling occurs.

ESC I (uppercase letter "I", HEX 049h) Moves cursor up 1 line, staying in same column. Scrolling occurs if cursor was on top line.

ESC n Causes current cursor position to be returned via emulated "keyboard" as ESC Y line# column#. This control sequence is ignored (not supported) by the Emulator.

ESC j Saves cursor position for later restore via ESC k.

ESC k Returns cursor to position that was saved via ESC j.

ESC Y line# column# Direct cursor addressing sequence. Screen lines are numbered 1 to 25. Screen columns are numbered 1 to 80. Line# and column# args are obtained by adding 31 (01Fh) to the desired line or column number. Alternatively, you may think of lines as being numbered from 0 to 24, columns from 0 to 79, and the offset to add to each being 32 (020h).

To position to line 5, column 10, for example, the following is sent:

ESC Y $)

which is represented in HEX as 01Bh 059h 024h 029H and in decimal as 27 89 36 41. Note that the line# arg is obtained by 5 + 31 = 36 (024h), and the column# arg by 10 + 31 = 41 (029h).

Line# or column# args less than 32 default to 32 (i.e. - to line or column 1). An arg value that is too large defaults to the max legal value (25 for line, 80 for column).

ESC F Erases the entire screen and homes the cursor.

ESC b Erases from the start of the screen to the cursor, including the cursor position.

ESC J Erases from the cursor to the end of the screen (including the cursor position).

ESC l (lowercase letter "L", HEX 06Ch) Erases the entire line that the cursor is on, positions the cursor to the left edge of that line.

ESC o (lowercase letter "O", HEX 06Fh) Erases from the beginning of the line to the cursor (including the cursor position).

ESC K Erases from the cursor to the end of the line (including the cursor position).

ESC L Inserts a blank line before the line that the cursor is on, shifts following lines (including the cursor line) down to make room. Cursor is moved to start of new blank line.

ESC M Deletes the line that the cursor is on, scrolls following lines up to fill its place. Cursor moves to left edge of its line.

ESC N Deletes the character under the cursor, shifts remaining text to left to cover it up.

ESC @ (At-sign character, HEX 040h) Enters Insert Mode. Displayed characters cause others on the same line to be moved right to make room. This can be pretty pokey, thanks to the IBM ROM BIOS!

ESC O (uppercase letter "O", HEX 04Fh) Exits Insert Mode.

ESC p Enters Reverse Video or Highlight mode.

ESC q Exits Reverse Video or Highlight mode.

BEL (decimal 7) Beeps bell.

BS (decimal 8) Backspaces cursor one character position.

HT (decimal 9) Tabs cursor to next mod-8 column boundary

LF (decimal 10, HEX 0Ah) Advances cursor to next line, same column.

FF (decimal 12, HEX 0Ch) Clears screen and homes cursor.

CR (decimal 13, HEX 0Dh) Returns cursor to start of current line.

The other standard ASCII control characters (below 032 or 20h) and those from 128 (80h) through 255 (FFh) display various graphic symbols. See the IBM Tech Ref Manual for details.

The emulated VT52 "terminal" also translates input from the IBM keyboard. Most keypresses are returned to the Z80 program as single ASCII characters. The "extended" codes that are generated by IBM function keys, arrow keys, ALT keys, and so on, are translated into 2-byte keyboard sequences as follows:

The first byte is an ESCAPE character (Decimal 027, HEX 1Bh).

The second byte is the keyboard scan code, as defined in the IBM Tech Ref Manual.

Additionally, the NUL extended code (CTRL-@) is translated into a single ASCII character (Decimal 000).

For example, assume that the user presses the PgUp key on the IBM keyboard. The next time that the Z80 program calls the CP/M BIOS to read a keypress from the "terminal", an ESCAPE character will be returned. The time after that, a Decimal 073 (HEX 49h) will be returned. This is the scan code for the PgUp key.

The above information may be used to configure particular Z80 applications for use with the Emulator's "terminal". Remember that this emulated "terminal" only appears at the CP/M BIOS interface. Applications that do I/O via BDOS functions (which includes most of the standard CP/M utilities) do not see this behavior.

WORDSTAR, dBase II, Target Plannercalc and PMATE, for example, do at least some of their terminal I/O via the CP/M BIOS, so they must be configured for the particular terminal being used. Refer to the above information when installing such applications for use under the Emulator, using this emulated VT52 "terminal". Please bear in mind, however, that things can get very confusing when one application (like dBase II)

does terminal I/O via both the BIOS and the BDOS. Just experiment until you get something useful.

> NOTE: This emulated terminal facility may be enabled and disabled via the "terminal" builtin command, which is described in the "Builtin Commands" section.

^C and BREAK Handling

The ^C (Ctrl-C) and BREAK (Ctrl-Scroll Lock) keys are handled in a special way on the IBM PC. Under normal circumstances, these keypresses are trapped by PCDOS and cause the executing PCDOS program (in this case, the Emulator) to be aborted.

This is undesirable.

Additionally, CP/M includes an important assumption, to wit: ^C is a keypress like any other, and must be passed through all the way to the Z80 program for processing. Several standard CP/M programs (PIP.COM, M80.COM, WORDSTAR, etc) use ^C as a command, and must not be aborted when it is typed.

For this reason, the Emulator traps ^C and BREAK keypresses. It decides what to do with them as follows:

> When not running a Z80 program (i.e. - when the Emulator is accepting commands), the BREAK key does nothing and the ^C keypress does various things depending on the mood of PCDOS.
>
> When executing a Z80 program, pressing ^C causes a ^C character (Decimal 003) to be queued as keyboard input. Pressing BREAK causes the Z80 to be stopped (with an appropriate message), and a return to the Emulator awaiting your command.
>
> NOTE: If the Z80 program is in the process of reading a line of keyboard input when you press BREAK, then you may have to press RETURN (to terminate that read) before the BREAK will be recognized.

When the Emulator stops ("aborts") a Z80 program due to a BREAK keypress, it displays a message to that effect. It suspends the Z80 in perfect order, maintaining all registers, flags, etc. It then accepts Emulator commands.

At this point you may inspect registers, run another Z80 program, or

whatever. You may also continue execution of the aborted program with the "go" command.

This is a very powerful tool for debugging, which is not available with true (hardware) Z80 systems. You may in effect interrupt the Z80 at any point, actually pausing it between Z80 instructions.

This also maintains compatability with existing CP/M programs that want to read the ^C character. To exit PIP, for example, you need only press ^C and then RETURN. This ^C is passed to PIP, who interprets it as a request to exit the program.

Note that RETURN must be pressed after the ^C in some cases (like when entering a command line into PIP). This is one area of PCDOS-CP/M incompatability, for which Joan Riff offers her apologies. A Z80 program (like PIP) that requests a line of input from CP/M won't see the ^C until the entire line is terminated with RETURN. A Z80 program that asks for one character at a time, however, will see the ^C immediately.

> NOTE: When running PCDOS commands like type, dir, and so on, the BREAK key is unavailable to you. If you want to interrupt the output from such programs that are run "underneath" the Emulator, you will have to use the ^C key.

Common PCDOS and CP/M File System

The Emulator goes to great lengths to allow CP/M programs to read and write PCDOS files. Thus the PCDOS file system serves as a common environment for both PCDOS and CP/M files. This allows you to use your favorite IBM PC editor, for example, to edit source files that are then compiled within CP/M (using the Emulator) with ASM.COM, M80.COM, F80.COM, or whatever.

CP/M's "User Number" concept, however, is primitive compared to the directory structure available with PCDOS. So although the emulated BDOS supports the setting of a user number, the user number is ignored by the Emulator when it comes time to actually access files.

Likewise, the concept of a Read-Only disk drive is not necessary under the Emulator. And the fatal CP/M flaw that crops up when you change disks and forget to type ^C to Warm Boot the system has been virtually eliminated by PCDOS.

CP/M and its CCP (or lack of them)

THE CP/M ENVIRONMENT

One thing that we did in order to achieve such a large TPA size (65022 bytes) and such fast emulation was remove the console command part of CP/M (the Console Command Processor, or CCP) from the CP/M Segment. In fact, we did away with CP/M entirely.

The CCP is the part of a normal CP/M system which accepts commands from the user and processes them. The Digital Research CCP that comes with CP/M 2.2 is fairly limited in its power.

In the Emulator, it is the main Z80MU 'C'-language program which contains all of the functions of CP/M's CCP, and does a whole lot more. In fact, it is this program which emulates all of CP/M (with a lot of help from PCDOS).

For this reason, there are certain CP/M enhancements (like ZCPR) which will have a whole lot of trouble working under the Emulator. The good news is that they are largely irrelevant under the Emulator, as the Emulator itself provides a powerful increase in console power even without ZCPR.

So if you have dreams of running ZCPR (or any CP/M enhancement which counts on patching CP/M), you ought to forget about it. When using the Emulator, there's no CP/M for such programs to patch. It's all fakeware, invisible to Z80 programs.

Stick with CP/M programs which interact with the outside world via the standard, unmodified BIOS and BDOS interfaces.

CP/M 2.2 BIOS and BDOS Emulation

The Emulator tries very hard to look to Z80 programs like CP/M version 2.2, at least in terms of its BIOS calls and BDOS support functions. Most Z80 programs that are run under the Emulator will have no idea that they aren't being run on a Z80 machine running Digital Research's CP/M.

There are some hardware-specific aspects of CP/M, however, that make no sense on a PCDOS system.

Many CP/M BIOS calls, for example, deal with the physical layout and operation of the floppy disk. Some of these are ignored. Others cause a fault of the Z80 program, with the display of a message to the effect that the program invoked an unsupported BIOS call.

There are a few BDOS functions which likewise are irrelevant, and which cause the Z80 program to be aborted.

THE CP/M ENVIRONMENT

The following is a list of the various BIOS calls that are supported by CP/M 2.2, and their effects under the Emulator. Calls that are marked "unsupported" cause the Z80 program to be aborted.

The addresses listed are absolute addresses within the CP/M Segment. Note that when the CP/M segment is Cold Booted, CP/M's location zero is set to JMP BIOS+3 (FF03h - the Warm Boot vector). The only safe way for a Z80 program to locate this BIOS jump table in any CP/M system (not just the Emulator) is to look at address 0001 within CP/M's memory.

BIOS Address	
FF00h	Cold Start CP/M segment
FF03h	Warm Start CP/M segment
FF06h	Set A-reg to FFh if Emulated VT52 Terminal has keypress to be read, 00h if not
FF09h	Get keypress from Emulated VT52 Terminal to A-reg, via IBM's ROM BIOS
FF0Ch	Output C-reg to Emulated VT52 Terminal via IBM's ROM BIOS
FF0Fh	Output C-reg to LPT1: via IBM's ROM BIOS
FF12h	Output C-reg to COM1: via IBM's ROM BIOS
FF15h	Get char from COM1: to A-reg via IBM's ROM BIOS
FF18h	Home Disk (unsupported)
FF1Bh	Select Disk (unsupported)
FF1Eh	Set Track (unsupported)
FF21h	Set Sector (unsupported)
FF24h	Set DMA address (unsupported)
FF27h	Read Sector (unsupported)
FF2Ah	Write Sector (unsupported)
FF2Dh	Set A-reg to FFh if LPT1: ready for output, 00h if not, as reported by IBM's ROM BIOS
FF30h	Sector Translate (unsupported)

Similarly, here is a table of the various BDOS functions (in Decimal/-HEX), and their actions under the Emulator. Note that a CP/M BDOS function is invoked by loading the function number into C-reg, and doing a CALL 0005h. See standard CP/M documentation for detailed calling conventions.

BDOS Function	
00/00h	Warm Boot. Returns to accept more Emulator commands. Does not alter memory in the CP/M Segment.
01/01h	Read char from PCDOS standard input to A-reg.
02/02h	Send E-reg to PCDOS standard output.

03/03h Read char from PCDOS AUX: device into A-reg.

04/04h Send E-reg to PCDOS AUX: device

05/05h Send E-reg to PCDOS PRN device

06/06h If E-reg on entry = FFh, then return in A-reg next char from PCDOS standard input, or 00h if none available at this instant.
If E-reg on entry <> FFh, then send E-reg to PCDOS standard output.

07/07h Get IOBYTE to A-reg.

08/08h Store E-reg into IOBYTE.

09/09h Output string at (DE) to PCDOS standard output.

10/0Ah Input line from PCDOS standard input to (DE).

11/0Bh Set A-reg to FFh if char from PCDOS standard input is ready to be read, or 00h if not.

12/0Ch Return CP/M version to HL. Sets reg L to 22h (for CP/M version 2.2), and reg H to 00h.

13/0Dh Reset disk system. Sets DMA to 80h. Does not change selected drive to A: (like CP/M does), as this is not necessary with PCDOS.

14/0Eh Set default drive to E-reg.

15/0Fh Open file whose FCB is at (DE). Sets A-reg to 00h if successful, else to FFh.

16/10h Close file whose FCB is at (DE). Sets A-reg to 00h if successful, else to FFh.

17/11h Search for first file that matches pattern in FCB at (DE). Sets A-reg to 00h if successful, else to FFh.

18/12h Search for next file that matches last pattern used. Sets A-reg to 00h if successful, else to FFh.

19/13h Delete file(s) represented by FCB is at (DE). Sets A-reg to 00h if successful, else to FFh.

20/14h Read next sequential record from file whose FCB is at (DE). Sets A-reg to status as follows:

0 = successful
1 = reading unwritten data (EOF)
FFh = PCDOS returned error # 2: "No room in DTA for record"

Note that a short record is filled out by the Emulator with ^Z (eof) characters.

21/15h Write next sequential record to file whose FCB is at (DE). Returns status in A-reg as follows:

0 = successful
5 = diskette full
6 = PCDOS returned error # 2: "No room in DTA for record"

22/16h Create (and open) file whose FCB is at (DE). Sets A-reg to 00h if successful, else to FFh.

23/17h Rename file(s) per special FCB at (DE). Sets A-reg to 00h if successful, else to FFh.

24/18h Return login vector (bitmap of known disks) to HL. Calls PCDOS to discover number of available drives.

25/19h Return default drive number in A-reg.

26/1Ah Set DMA to DE.

27/1Bh Return allocation information (unsupported - aborts Z80 program). See note below.

28/1Ch Write-protect drive (ignored).

29/1Dh Return write-protect vector (bitmap of $R/O drives) to HL. Sets HL to zero (nobody's write-protected).

30/1Eh Set file attributes (ignored, but returns A-reg of 00h to indicate success).

31/1Fh Return physical disk information (unsupported - aborts Z80 program). See note below.

32/20h If E-reg on entry is FFh, then current user number is returned in A-reg.

If E-reg on entry <> FFh, then current user number is set to E-reg MOD 32.

This only updates the byte at CP/M address 4. The Emulator ignores this byte when accessing files.

33/21h Read random record from file whose FCB is at (DE). Returns A-reg status as follows:

0 = successful
1 = reading unwritten data
3 = (CP/M "Cannot close current extent" error): PCDOS returned error # 2: "No space in DTA for record"

Note that a partial record is filled out by the Emulator with ^Z (eof) characters.

34/22h Write random record to file whose FCB is at (DE). Returns A-reg status as follows:

0 = successful
3 = (CP/M "Cannot close current extent" error): PCDOS returned error # 2: "No space in DTA for record"
5 = (CP/M "Directory Overflow" error): Diskette full

35/23h Compute file size for file whose FCB is at (DE). Result goes into random record field of FCB.
36/24h Set random record field per FCB at (DE).
37/25h Reset drive (accepted but ignored)
38/26h (unsupported - aborts Z80 program)
39/27h (unsupported - aborts Z80 program)
40/28h Write random record with zero fill. In the Emulator, this is translated to a function 34/22h (above).

NOTE: If someone will kindly provide us with a coherent writeup of the disk parameter block and allocation information as returned by BDOS functions 31/1Fh and 27/1Bh above, then we will gladly emulate these functions in the next release.

In CP/M the BDOS routines call the BIOS routines. This is not true in the Emulator. The Emulator's BDOS functions in general invoke the corresponding PCDOS functions, and the Emulator's BIOS routines call the IBM PC ROM BIOS routines.

EMULATOR BUILTIN COMMANDS

BUILTIN COMMANDS

This section describes those commands that are recognized and acted upon by the Emulator itself. Such commands do not involve a search of the disk for a corresponding .COM file, since the Emulator recognizes them as special commands which are to be handled within the Emulator itself. This process is roughly equivalent to the handling of CP/M's CCP commands.

Command Name Conflicts

You may have a .COM (Z80 command) file that has the same name as one of these Builtin Commands. How do you tell the Emulator to run your .COM file, instead of doing its corresponding Builtin Command? All that you have to do is convince the Emulator that your command is indeed a disk file. This can be done by including a drive ID or pathname as part of your command:

```
Z80 A>a:dump foobar.asc
Z80 A>b:\bin\dump foobar.asc
Z80 A>\mystuff\dump foobar.asc
```

Alternatively, you may want to rename your .COM file so that it no longer conflicts with an Emulator Builtin Command name.

Numeric Arguments

Some Builtin Commands accept numeric arguments. These may represent addresses to be dumped, the number of pages to save, or whatever.

A numeric argument may be entered in any of several ways:

- As a HEX number. No prefix is required in this case, as this is the default numeric radix.

Examples: ffff 0 7F -5 +3FF

As a decimal number, prefixed by a period (".").

Examples: .10 .0 -.1 +.256 .65022

As a binary number, prefixed with the percent sign ("%").

Examples: %0 %1010101011110000 -%1

As an ASCII character, prefixed by the apostrophe ("'").

Examples: 'A '" '0 -'Z

As an ASCII escape sequence, prefixed by an apostrophe and back-slash ("'\").

Examples:

```
'\\            (single "\" char)
'\0            (NUL byte)
'\b or '\B     (Backspace char)
'\t or '\T     (TAB char)
'\n or '\N     (LINEFEED char)
'\r or '\R     (CARRIAGE RETURN char)
'\'            (single apostrophe ("'") char)
'\"            (single double-quote char )
'\xFF or '\XFF (byte with HEX value of FF)
```

As a label which has been defined with the "label" builtin command.

Examples: fcb1 program_start BDOS -reserved

As two or more of the above entries, connected with "+" or "-" operators.

Examples:

```
fcb1+5
program_end-table_length
'A-40+'a
table-5+offset
```

EMULATOR BUILTIN COMMANDS

The Builtin Commands are presented by functional grouping:

- The PCDOS pass-through command prefix
- CP/M Builtins that are emulated
- Emulator Builtins that are similar to CP/M's
- General Emulator commands
- Emulator DEBUG commands
- CP/M Environment and file control commands
- RESOURCE commands

BUILTIN COMMANDS: PCDOS PASS-THROUGH

Builtin:

Passes command xxxxxx to PCDOS. This gives you a way to use the usual PCDOS utilities from within the emulator, without having them interpreted as CP/M commands. Everything after the "!" character is passed as a command to PCDOS.

This command requires the reloading (by PCDOS) of the PCDOS COMMAND.COM file from disk. See the section on "The PCDOS Environment" for more detail on this subject.

Since PCDOS handles the command (and any command arguments that may be present), the standard PCDOS "PATH" environment string may apply. So may all other PCDOS conventions, like I/O redirection, wildcards, PCDOS device names, etc.

The given command is executed above the Emulator's memory. This implies that there had better be enough memory available above the Emulator to run the given command.

You may use this facility to "drop into" PCDOS for a while (perhaps to use a PCDOS screen editor on a CP/M source file), and then return to the Emulator. Use "!command" to drop into DOS, and "exit" to leave PCDOS and return to the Emulator.

This is also the primary way to take advantage of PCDOS's directory structure while within CP/M. You may issue "CHDIR", "MKDIR", and other directory-related commands directly to PCDOS. The effect of such commands carries over to the Emulator.

Example (from an actual session):

```
Z80 A>!cd \foo
Z80 A>!cd \
Z80 A>!rd \foo
Z80 A>!chkdsk b:


   362496 bytes total disk space
     1024 bytes in 1 directories
   272384 bytes in 10 user files
    89088 bytes available on disk

   423936 bytes total memory
   152384 bytes free

Z80 A>!md foo
Z80 A>!cd foo
Z80 A>stat *.*


     Volume in drive A has no label
     Directory of  A:\foo

     .         <DIR>     11-19-85   1:15a
     ..        <DIR>     11-19-85   1:15a
     2 File(s)    144384 bytes free

Z80 A>!command


Z80 A>!The IBM Personal Computer DOS
Version 2.00 (C)Copyright IBM Corp 1981, 1982, 1983

Tue 11-19-1985 1:16:00.18
A:>chkdsk


   362496 bytes total disk space
     1024 bytes in 1 directories
   217088 bytes in 17 user files
   144384 bytes available on disk

   423936 bytes total memory
   149264 bytes free
```

```
Tue 11-19-1985 1:16:16.17
A:>exit

Z80 A>!format a:
```

BUILTIN COMMANDS: EMULATED CP/M BUILTIN'S

Builtin:

Changes the default disk drive to the drive whose letter name is represented by the "d" character. The command prompt will change to

```
Z80 d>
```

to reflect the new default. Drive d's current PCDOS directory will then be the first directory searched for Z80 command (.COM) files, unless explicitly overridden by a drive prefix or pathname. Data files that are created will default to that drive and its current directory. Data files that are read will be searched for only in that drive and directory, unless explicitly overridden by the Z80 program.

Example (from an actual session):

```
Z80 A>stat b:*.*


      Volume in drive B has no label
      Directory of  B:\

     WORDSTAR     <DIR>     11-02-85   4:19a
          1 File(s)     89088 bytes free

Z80 A>dir b:\wordstar\ws*.*


      Volume in drive B has no label
      Directory of  B:\wordstar

     WSMSGS   OVR   27904  11-06-85   6:27p
     WSOVLY1  OVR   34048  11-06-85   6:30p
     WSU      COM   15872  11-06-85   6:32p
     WS       COM   15872  11-06-85   6:37p
          4 File(s)     89088 bytes free

Z80 A>b:

Z80 B>dir


      Volume in drive B has no label
      Directory of  B:\

     WORDSTAR     <DIR>     11-02-85   4:19a
          1 File(s)     89088 bytes free
```

```
Z80 B>stat a:\*.exe

      Volume in drive A has no label
      Directory of  A:\

     Z80MU    EXE   94976  11-18-85   3:31p
          1 File(s)    142336 bytes free
```

BUILTIN COMMANDS: EMULATED CP/M BUILTIN'S

Builtin: del <pattern>
         era <pattern>
         delete <pattern>
         erase <pattern>

These are identical commands. All cause the invocation of the PCDOS "DEL" command to delete files that match <pattern>.

See the PCDOS manual for details of <pattern>.

We have created several synonyms for the same command in order to make life easier for CP/M folks who are used to saying "ERA", and PCDOS folks who are used to saying "DEL" or whatever.

These commands cause PCDOS to reload COMMAND.COM, so see "The PCDOS Environment" section for further detail on that subject.

Example:

```
Z80 A>erase b:*.asm
Z80 A>del *.*
     Are you sure? y
Z80 A>delete c:\backup\foo.*
Z80 A>era foo.asm
```

BUILTIN COMMANDS: EMULATED CP/M BUILTIN'S

Builtin: dir <pattern>
stat <pattern>

Shows a directory of files matching <pattern>. This invokes the PCDOS "DIR" command, so remember about COMMAND.COM (see "The PCDOS Environment" section).

<pattern> is passed directly to PCDOS. So see the PCDOS manual if you want to know what's legal.

The CP/M STAT command (STAT.COM) cannot be emulated, because the first thing that Digital Research's STAT.COM does is invoke a hardware-specific CP/M function that means nothing on the IBM PC, and is therefore illegal within the Emulator. But the most common function of STAT.COM - displaying filenames and file sizes with "STAT *.*" or whatever - can be done with PCDOS's "DIR" command. So STAT and DIR have been made to do the same thing. If you're used to typing "STAT *.*" in CP/M, then you'll be able to do the same thing under the Emulator.

Example (from an actual session):

```
Z80 A>stat b:*.*


 Volume in drive B has no label
 Directory of  B:\

WORDSTAR     <DIR>     11-02-85   4:19a
     1 File(s)     89088 bytes free

Z80 A>dir b:\wordstar\ws*.*


 Volume in drive B has no label
 Directory of  B:\wordstar

WSMSGS   OVR   27904  11-06-85   6:27p
WSOVLY1  OVR   34048  11-06-85   6:30p
WSU      COM   15872  11-06-85   6:32p
WS       COM   15872  11-06-85   6:37p
     4 File(s)     89088 bytes free

Z80 A>b:

Z80 B>dir
```

```
           Volume in drive B has no label
           Directory of  B:\

          WORDSTAR     <DIR>     11-02-85   4:19a
               1 File(s)     89088 bytes free

     Z80 B>stat a:\*.exe


           Volume in drive A has no label
           Directory of  A:\

          Z80MU    EXE   94976  11-18-85   3:31p
               1 File(s)    142336 bytes free
```

Builtin: type <filename.typ>
ty <filename.typ>

This command is the equivalent of the CP/M "TYPE" command. The specified file is displayed on the standard output (normally the screen).

The display may be paused with either the CP/M convention of ^S/^Q or the PCDOS convention of CTRL-NUMLOCK. It may be aborted with ^C or CTRL-BREAK.

This command causes PCDOS to reload COMMAND.COM, so see "The PCDOS Environment" section for further detail on that subject.

Example:

```
Z80 A>ty b:foo.asm
Z80 A>type \source\backup\foo.doc
Z80 A>ty ctest.err
```

Builtin: rename <oldpath> <newpath>
ren <oldpath> <newpath>

Renames the file specified by <oldpath> to the name given by <newpath>.

NOTE: This is not the same syntax used by the CP/M equivalent, which is "REN <newname>=<oldname>".

This command causes PCDOS to reload COMMAND.COM, so see "The PCDOS Environment" section for further detail on that subject.

This command invokes the PCDOS "RENAME" command. See the PCDOS manual for details.

Example:

```
Z80 A>rename dbase.exe dbase.xxx
```

Builtin: save n <filename.typ>
sa n <filename.typ>

Saves n 256-byte pages of CP/M memory (starting at address 0100h) to the specified file. The data written to the file is a simple memory image. No translation is done, even if a .HEX extension is given in the filename. If you want to write a true Intel HEX file, use the write Builtin.

NOTE: This is close to the CP/M equivalent, except that the default radix for n is HEX, not decimal as with CP/M.

Example (from an actual session):

```
Z80 A>save 3 820init2.com

     Writing 3 pages (768 bytes) to file '820INIT2.COM'

Z80 A>save 0 continue.com

     Writing 0 pages (0 bytes) to file 'CONTINUE.COM'
```

BUILTIN COMMANDS: SIMILAR TO CP/M'S

Builtin: copy <from_pattern> <to_path>
co <from_pattern> <to_path>

Copies the file(s) specified by <from_pattern> to the file or directory specified by <to_path>.

This command invokes the PCDOS "COPY" command, so see the PCDOS manual for details as to what's legal.

It also causes PCDOS to reload COMMAND.COM, so see "The PCDOS Environment" section for further detail on that subject.

This command is roughly equivalent to CP/M's "PIP <outfile>=<infile>". You have the additional power of PCDOS's directory and device name support, however.

Example:

```
Z80 A>copy *.* B:
Z80 A>co b:*.asm
Z80 A>copy \bin\*.* c:\backup
Z80 A>copy *.asm combined.bak
Z80 A>co *.asm *.bak
Z80 A>co foo.asm lpt1:
Z80 A>co con autoexec.bat
```

BUILTIN COMMANDS: GENERAL

Builtin: help [command]
? [command]

Displays a brief description of the requested Builtin Command.

If no command name follows the "help" command, then a rather lengthy explanation of all commands is displayed. If you press the SPACE bar, this long listing will be interrupted at the next logical break.

The listing can be paused with ^S/^Q, or CTRL-NumLock. You can turn printer copying on before the listing starts (which is recommended) with ^P or CTRL-PrtSc.

Example:

```
Z80 A>help xreg
Z80 A>? list
Z80 A>?
Z80 A>help ?
Z80 A>help b:
Z80 A>help !
```

Builtin: illop [fault | nop]
i [fault | nop]

Specifies how the Emulator is to handle illegal Z80 opcodes.

If "illop fault" is entered, then an illegal Z80 opcode will cause a Z80 fault, meaning that the Emulator will stop executing the Z80 program and will display an error message to the effect that an illegal opcode was encountered at such-and-such and address.

If "illop nop" is entered, then an illegal Z80 opcode will simply be ignored. This is closer to true Z80 operation. Execution will continue with the next Z80 instruction after the illegal opcode.

If only "illop" is entered, then the Emulator simply reports how it is currently handling illegal opcodes.

Example (from an actual session):

```
Z80 A>i

     Illegal opcodes will act as NOP's

Z80 A>illop fault

     Illegal opcodes will FAULT

Z80 A>illop nop

     Illegal opcodes will act as NOP's
```

BUILTIN COMMANDS: GENERAL

Builtin: exit
e

Exits the Emulator, and returns to PCDOS.

Due to certain limitations within PCDOS, you should not remove any floppy disks that are being used until you have exited the Emulator via this command, unless you know that the Z80 programs which you have run have truly closed any files that have been written to.

We have used the Emulator extensively, and have frequently changed floppies while within the Emulator. We have never experienced corrupted floppy data. But then, we use only "safe" CP/M software like ASM.COM, M80.COM, L80.COM, and so on. Such programs are very good about closing files when they exit.

This warning is included not because it has ever happened to us, but because we wrote the code, and we know about certain "windows" within which a faulting CP/M program could conceivably confuse PCDOS into writing one floppy's data to another floppy, destroying the second floppy's file data and perhaps even its FAT (File Allocation Table).

Example:

```
Z80 A>exit
A>
```

BUILTIN COMMANDS: GENERAL

Builtin: speed?
howfast?

Calculates the effective speed of the imaginary Z80 that exists within the Emulator.

A sample Z80 program is loaded into the CP/M segment. It is run, and its execution is timed. This program takes up to half a minute to run on a simple IBM PC with an 8088, and correspondingly less time on faster machines (i.e. - a PC with a NEC V20 chip, an IBM PC/AT, etc).

The effective clock speed is displayed at the end of the test.

The reported speed should be taken with a grain of salt. What it means is that if you had a real Z80 running the exact instruction mix found in the test program, then that real Z80 would have to run at the reported clock speed in order to perform as fast (or slow) as the Emulator's imaginary Z80.

For example, assume that the reported effective clock is 250,000 Hz. This means that the imaginary Z80 in the Emulator is running the test program at one fourth the speed of a 1 MegaHertz Z80 (as found on a Microsoft SoftCard in an Apple, for example), one eighth the speed of a 2 MegaHertz Z80, etc.

Does this mean that your CP/M programs run under the Emulator will run at one fourth the speed of an Apple with a SoftCard? Not necessarily. The reported speed is for CPU-bound (no I/O) operation, of the exact mix of instructions found in the test program. Real CP/M programs tend to have a mix of CPU and I/O operations. I/O operations to disk are handled as fast as the IBM PC can do them. They aren't emulated, they are done. And CP/M programs whose CPU-bound operations involve a lot of register-to-register operations will be emulated faster than those requiring a lot of memory accesses.

It's a complicated relationship. Our experience has been that an assembly (using ASM.COM) runs about 1/5th the speed under the Emulator on a stock, floppy-based IBM PC than it does on an Apple with a SoftCard and 1 MegaHertz Z80. The addition of a NEC V20 processor to the IBM PC improves performance. So does using a hard disk instead of floppies. And moving the Emulator to an IBM PC/AT brings emulated performance close to that of a 1 MegaHertz Z80.

NOTE: This command causes a Cold Boot of the CP/M segment. This destroys any CP/M program that you may have had in memory.

Example (from an actual session):

```
Z80 A>speed?

     *** CP/M Segment COLDBOOTED ***

     Beginning Z80 timing test. Please wait...

     Effective Z80 clock speed is 248101 Hz

     *** CP/M Segment COLDBOOTED ***
```

Builtin: btrace [SOME | ALL]
bt [SOME | ALL]

Displays the current BDOS Trace Table (if no arguments are present), or controls the BDOS functions that will be traced.

As a CP/M program is run, the various BDOS calls that it makes are traced. This command displays the trace table as it has been left by the last CP/M program run.

Items reported include:

1) The trace table sequence number.
2) The Program Counter of the CALL to the BDOS.
3) The contents of the Z80 DE register at the entry to the BDOS handler.
4) The DMA address in effect at the instant of this BDOS call.
5) The BDOS function # (as passed in the Z80 C-reg) and a text description of the function being performed.

The BDOS Trace Table is cleared by a COLD BOOT, by the load of a new CP/M program, and at various other times when it seems logical to clear it out.

If an argument (either "SOME" or "ALL") is present, then the trace table is not displayed. Instead, the Emulator adjusts (according to the argument) the way that future traces will be made:

If "SOME" is specified, then the Console Status, Console Output, Direct Console I/O, and List Output BDOS functions will not be traced. This can help to keep the BDOS Trace Table from filling up with unimportant entries.

If "ALL" is specified, then even these console character BDOS functions will be traced.

Example (from an actual run of DUMP.COM):

```
Z80 A>btrace all

Future BDOS traces will include console character functions

Z80 A>a:dump dump.com

0000 21 00 00 39 22 15 02 31 57 02 CD C1 01 FE FF C2
0010 1B 01 11 F3 01 CD 9C 01 C3 51 01 3E 80 32 13 02
0020 21 00 00 E5 CD A2 01 E1 DA 51 01 47 7D E6 0F C2
...
...  (Much of DUMP.COM output deleted for brevity)
...
0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Z80 A>bt

BDOS TRACE TABLE:
SEQ#  Z80PC  Z80DE Z80DMA FUNCTION
----- -----  ----- ------ ----------------
  006 016BH  0030H  0080H 02H Console Output from E-reg = 0
  007 016BH  0030H  0080H 02H Console Output from E-reg = 0
  008 016BH  0020H  0080H 02H Console Output from E-reg =

...
...  (Many entries deleted for brevity)
...

 049 016BH  0030H  0080H 02H Console Output from E-reg = 0
 050 01D6H  005CH  0080H 14H Read File (Sequential), FCB at (DE)
 051 016BH  000DH  0080H 02H Console Output from E-reg = ^M
 052 016BH  000AH  0080H 02H Console Output from E-reg = ^J
 244 016BH  0020H  0080H 02H Console Output from E-reg =
 245 016BH  0030H  0080H 02H Console Output from E-reg = 0
 246 016BH  0030H  0080H 02H Console Output from E-reg = 0
 247 016BH  0020H  0080H 02H Console Output from E-reg =
 248 016BH  0030H  0080H 02H Console Output from E-reg = 0
 249 016BH  0030H  0080H 02H Console Output from E-reg = 0
 250 016BH  000DH  0080H 02H Console Output from E-reg = ^M
 251 016BH  000AH  0080H 02H Console Output from E-reg = ^J
 252 015EH  0070H  0080H 0BH Get Console Status to A-reg
 253 016BH  0030H  0080H 02H Console Output from E-reg = 0
 254 016BH  0031H  0080H 02H Console Output from E-reg = 1
 255 016BH  0037H  0080H 02H Console Output from E-reg = 7
 001 016BH  0030H  0080H 02H Console Output from E-reg = 0
```

```
      002 016BH  0020H  0080H 02H Console Output from E-reg =
      003 016BH  0030H  0080H 02H Console Output from E-reg = 0
      004 016BH  0030H  0080H 02H Console Output from E-reg = 0
      005 016BH  0020H  0080H 02H Console Output from E-reg =
     -- END OF BDOS TRACE TABLE --
```

```
Builtin:  break
          break clear [n [n...]]
          break set n [n...]
          b
          b clear [n [n...]]
          b set n [n...]
```

Manipulates the Breakpoint Table, which contains up to 50 Z80 addresses at which execution of the Z80 is to be halted, and control returned to the Emulator's command prompt.

Breakpoints are typically used in conjunction with the "read" and "go" commands, and various other debug commands.

The Breakpoint Table is cleared when a new CP/M program is loaded, when the CP/M Segment is Cold Booted, and at various other times when it seems logical to clear it.

If only "break" is entered, then the current Breakpoint Table addresses are displayed.

If "break set" is entered followed by one or more Z80 addresses, then the addresses following the command are added to the Table.

If only "break clear" is entered, then all active breakpoint addresses are removed. No execution breakpoints will occur.

If "break clear" is entered followed by one or more Z80 addresses, then only the specified addresses are removed from the Breakpoint Table.

> NOTE: When a breakpoint is encountered during execution of the Z80 code, that breakpoint's address is automatically cleared from the table.

Example:

```
Z80 A>break clear

     0 Breakpoints cleared

Z80 A>b set startup startup+3 14f 210 221

     5 Breakpoints set - 5 now in table

Z80 A>b clear 221

     1 Breakpoints cleared, 4 left

Z80 A>b

     Current Breakpoints:
     0100H  0103H  014FH  0210H
     4 Breakpoints currently set

Z80 A>b clear 210 startup

     2 Breakpoints cleared, 2 left

Z80 A>b

     Current Breakpoints:
     0103H  014FH
     2 Breakpoints currently set

Z80 A>b clear

     2 Breakpoints cleared

Z80 A>b

     Current Breakpoints:
     0 Breakpoints currently set
```

BUILTIN COMMANDS: DEBUG SUPPORT

Builtin: dump [n1 [n2]]
d [n1 [n2]]

Dumps Z80 (CP/M) memory in HEX and ASCII.

If n1 is given, then the first address dumped is n1. If no arguments are present, then the first address is the one following the last one done by a previous dump command.

If n2 is given, then the dump continues through Z80 (CP/M) address n2. If n2 is not given, then n2 is assumed to be 255 bytes beyond the starting address.

A lengthy dump may be interrupted by pressing the SPACE bar.

Example (from an actual session):

```
     Z80 A>dump 100

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0100: 21 00 00 39 22 15 02 31  57 02 CD C1 01 FE FF C2 !..9"..1 W.MA.~.B
0110: 1B 01 11 F3 01 CD 9C 01  C3 51 01 3E 80 32 13 02 ...s.M.. CQ.>.2..
0120: 21 00 00 E5 CD A2 01 E1  DA 51 01 47 7D E6 0F C2 !..eM".a ZQ.G}f.B
0130: 44 01 CD 72 01 CD 59 01  0F DA 51 01 7C CD 8F 01 D.Mr.MY. .ZQ.|M..
0140: 7D CD 8F 01 23 3E 20 CD  65 01 78 CD 8F 01 C3 23 }M..#> M e.xM..C#
0150: 01 CD 72 01 2A 15 02 F9  C9 E5 D5 C5 0E 0B CD 05 .Mr.*..y IeUE..M.
0160: 00 C1 D1 E1 C9 E5 D5 C5  0E 02 5F CD 05 00 C1 D1 .AQaIeUE .._M..AQ
0170: E1 C9 3E 0D CD 65 01 3E  0A CD 65 01 C9 E6 0F FE aI>.Me.> .Me.If.~
0180: 0A D2 89 01 C6 30 C3 8B  01 C6 37 CD 65 01 C9 F5 .R..F0C. .F7Me.Iu
0190: 0F 0F 0F 0F CD 7D 01 F1  CD 7D 01 C9 0E 09 CD 05 ....M}.q M}.I..M.
01A0: 00 C9 3A 13 02 FE 80 C2  B3 01 CD CE 01 B7 CA B3 .I:..~.B 3.MN.7J3
01B0: 01 37 C9 5F 16 00 3C 32  13 02 21 80 00 19 7E B7 .7I_..<2 ..!...~7
01C0: C9 AF 32 7C 00 11 5C 00  0E 0F CD 05 00 C9 E5 D5 I/2|..\. ..M..IeU
01D0: C5 11 5C 00 0E 14 CD 05  00 C1 D1 E1 C9 46 49 4C E.\...M. .AQaIFIL
01E0: 45 20 44 55 4D 50 20 56  45 52 53 49 4F 4E 20 31 E DUMP V ERSION 1
01F0: 2E 34 24 0D 0A 4E 4F 20  49 4E 50 55 54 20 46 49 .4$..NO  INPUT FI

     Z80 A>d primary_fcb secondary_fcb+.15

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0050:                                     01 44 55 4D             .DUM
0060: 50 20 20 20 20 43 4F 4D  00 00 80 00 80 01 00 00 P    COM ........
0070: 6D 0B 1B BF 40 F3 00 00  00 F3 00 00             m..?@s.. .s..
```

Builtin: go [n]
g [n]

Begins execution of Z80 code at address n. If n is not specified, then the starting address defaults to the current Z80 Program Counter (PC).

This is the usual way to run a program that has been read from disk. It is also used to continue execution when the Z80 has been stopped via a breakpoint, or by the user pressing the BREAK key.

> NOTE: A Z80 command (.COM) file that is invoked by typing its name is automatically run. It does not require this command in order to be executed.

Example:

```
Z80 A>go
Z80 A>g 100
Z80 A>g -.768
Z80 A>go fixit5-3
```

Builtin: find n "text"
f n "text"

Searches the CP/M Segment for the binary pattern represented by string "text". The search begins at Z80 address n, which is a numeric value as described at the start of this section.

The "text" string may (and usually does) include imbedded escape sequences, as follows:

\\	(single "\" char)
\0	(NUL byte)
\b or \B	(Backspace char)
\t or \T	(TAB char)
\n or \N	(LINEFEED char)
\r or \R	(CARRIAGE RETURN char)
\'	(single apostrophe ("'") char)
\"	(single double-quote char)
\xFF or \XFF	(byte with HEX value of FF)

The CP/M Segment address of each match is displayed as a four-digit HEX value. The search ends with CP/M Segment address 0FFFFh.

Example (from an actual session):

```
Z80 A>patch 8000 "Joan Riff"

Z80 A>find 100 "Joan"
     8000H

Z80 A>f .256 "\xcd\x05\x00"
     015EH 016BH 019EH 01CAH 01D6H
```

BUILTIN COMMANDS: DEBUG SUPPORT

```
Builtin:  patch [n]
          patch n "xxxx"
          p [n]
          p n "xxxx"
```

Either enters an interactive patch dialog which allows you to change CP/M memory a byte at a time, or else applies string patch "xxxx" to the CP/M Segment and does not enter interactive patch mode.

If n (which is a numeric value as defined at the start of this section) is specified, then patching starts at address n. If n is not specified, then patching begins with next patch location (the one above the last location patched). Note that in the patch n "xxxx" format, numeric value n is required.

The interactive patch dialog consists of:

1) a prompt which shows the next address to be patched and its current contents.

2) user responses.

User responses to the interactive patch prompt are as follows:

?<return> A question mark (followed by RETURN) to request a short help message showing available responses.

n<return> A standard numeric argument as described at the start of this section. This is the byte value to be patched into the specified CP/M address.

NOTE: This may also be a 16-bit value. If the high-order byte of the resultant n is non-zero, then this is taken to be a 16-bit value, and fills 2 bytes.

"xxx" A string of ASCII text, delimited by double-quote characters. The bytes of the string are patched into successive CP/M memory locations. The string may include ASCII escape sequences as follows:

```
\\             (single "\" char)
\0             (NUL byte)
\b or \B       (Backspace char)
\t or \T       (TAB char)
\n or \N       (LINEFEED char)
\r or \R       (CARRIAGE RETURN char)
\'             (single apostrophe ("'") char)
```

```
          \"             (single double-quote char )
          \xFF or \XFF   (byte with HEX value of FF)
```

<space><return> to leave the addressed byte unchanged, and move to next one.

<return> to exit the interactive patch mode.

;xxx A comment (everything following the semicolon is ignored).

Example (from an actual session):

```
Z80 A>patch 400


Enter '?' for help with PATCH entries


0400H (00H) = 'R
0401H (00H) = 'I+20
0402H (00H) = "ff\0"
0405H (00H) =

Z80 A>d 400 40f  400 40f

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0400: 52 69 66 66 00 00 00 00  00 00 00 00 00 00 00 00 Riff.... ........

Z80 A>p primary_fcb "JoanRiff   "

Z80 A>d primary_fcb secondary_fcb+.15

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0050:                                     4A 6F 61 6E             Joan
0060: 52 69 66 66 20 20 20 4D  00 00 80 00 80 01 00 00 Riff   M ........
0070: 6D 0B 1B BF 40 F3 00 00  00 F3 00 00             m..?@s.. .s..
```

BUILTIN COMMANDS: DEBUG SUPPORT

Builtin: xreg [rr n]
x [rr n]

Sets Z80 register or flag rr to value n.

If no args are present, then the current Z80 register values and flags are displayed.

If args are present, then the register or flag represented by rr is set to the numeric value n (whose format is described at the beginning of this section).

NOTE: The Z80 has a primary and an alternate set of registers and flags. The alternate set is indicated by appending an apostrophe ("'") to the register or flag name.

The register or flag to be set (the rr argument) must be one of the following (in either upper or lowercase):

regs:	A	F	B	C	D	E	H	L
	A'	F'	B'	C'	D'	E'	H'	L'
	AF	BC	DE	HL				
	AF'	BC'	DE'	HL'				
	IX	IY	SP	PC				
	IFF1	IFF2	IMF	I	R			
flags:	SF	ZF	HF	P/V	NF	CF		
	SF'	ZF'	HF'	P/V'	NF'	CF'		

Example (from an actual session):

```
Z80 A>xreg pc 0

Z80 A>x

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 000F 007F FEFC 0000 0000 00 00 FEFE 0000   0    0   0
0000'0000'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L0000:    C3 03 FF       JP   LFF03

Z80 A>xreg bc ffff
```

```
Z80 A>x

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 FFFF 007F FEFC 0000 0000 00 00 FEFE 0000   0    0   0
0000'0000'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L0000:    C3 03 FF       JP   LFF03

Z80 A>x c' 'A

Z80 A>xreg de .256

Z80 A>xreg AF' 55

Z80 A>xreg

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 FFFF 0100 FEFC 0000 0000 00 00 FEFE 0000   0    0   0
0055'0041'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L0000:    C3 03 FF       JP   LFF03
```

Builtin: trace [n]
t [n]

Traces a certain number (specified by the n argument) of Z80 instruction executions, displaying Z80 regs and flags after each instruction's execution.

Execution begins at the current Z80 Program Counter (PC).

If no argument is given, then n defaults to 1.

The n argument is a numeric value as described at the beginning of this section.

> NOTE: The PC of each executed instruction is saved in a circular buffer for later interpretation by the "pc?" builtin command(q.v.).

Example (from an actual trace of DDT.COM's opening lines):

```
Z80 A>read ddt.com

*** Low = 0100H  Next = 1400H
*** Z80 DMA, PC and Stack automatically set for .COM file

Z80 A>t 4

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 0FBC 0100 FEFC 0000 0000 00 00 FEFC 0103   0    0   0
0055'0041'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L0103:    C3 3D 01       JP   L013D

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 0FBC 0100 FEFC 0000 0000 00 00 FEFC 013D   0    0   0
0055'0041'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L013D:    31 00 02       LD   SP,L0200

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 0FBC 0100 FEFC 0000 0000 00 00 0200 0140   0    0   0
0055'0041'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L0140:    C5             PUSH BC

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 0FBC 0100 FEFC 0000 0000 00 00 01FE 0141   0    0   0
0055'0041'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
L0141:    C5             PUSH BC
```

Builtin: notrace [n]
n [n]

Executes a certain number (specified by the n argument) of Z80 instructions, beginning at the current Z80 Program Counter (PC). The Z80 registers are not displayed after every instruction, but are displayed after the last instruction.

If no argument is given, then n defaults to 1.

The n argument is a numeric value as described at the beginning of this section.

> NOTE: The PC of each executed instruction is saved in a circular buffer for later interpretation by the "pc?" builtin command(q.v.).

Example:

```
Z80 A>n 3

A F  B C  D E  H L  IX   IY   I  R  SP   PC   IFF1 IFF2 IMF
0A01 0F09 0130 FEFC 0000 0000 00 00 01FA FEFE   0    0   0
0055'0041'0000'0000'SF=0 ZF=0 HF=0 P/V=0 NF=0 CF=1
LFEFE:    76             HALT
```

Builtin: pctrace? [FIRST/LAST n [FULL/BRIEF]]
pc? [FIRST/LAST n [FULL/BRIEF]]

Displays the Z80 PC's which were saved during the last "trace" or "notrace" execution. This is very useful for finding out how the Z80 wound up at a particular address, or where it went from there.

The display may proceed from the oldest PC toward the newest (FIRST n), or in the reverse direction (LAST n).

The display may include only the PC itself (BRIEF) or the complete disassembled instruction at each PC (FULL).

Note that a FULL display assumes that instructions haven't been modified since they executed. All that is saved in the circular PC queue is the PC itself. For a FULL display, the current contents of whatever is at that address is disassembled.

The n argument is a numeric value as described at the beginning of this section.

The default is LAST 512 FULL.

Example:

```
Z80 A>pc?
Z80 A>pctrace? first 20 full
Z80 A>pc? last 20 brief
```

Builtin: move nlow nhigh ndest
m nlow nhigh ndest

Moves CP/M memory from one location within the CP/M Segment to another location within the CP/M Segment.

The block of memory to be moved is defined by nlow through nhigh.

The new location for the block is defined by ndest.

All three arguments are numeric values as described at the start of this section.

NOTE: The move is done either left-to-right or right-to-left, as needed. So no smearing is possible.

Example (from an actual session):

```
Z80 A>p primary_fcb "JoanRiff   "

Z80 A>d primary_fcb secondary_fcb+.15

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0050:                                     4A 6F 61 6E              Joan
0060: 52 69 66 66 20 20 20 4D  00 00 80 00 80 01 00 00 Riff   M ........
0070: 6D 0B 1B BF 40 F3 00 00  00 F3 00 00              m..?@s.. .s..

Z80 A>move primary_fcb secondary_fcb secondary_fcb

Z80 A>d primary_fcb secondary_fcb+.15

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0050:                                     4A 6F 61 6E              Joan
0060: 52 69 66 66 20 20 20 4D  00 00 80 00 4A 6F 61 6E Riff   M ....Joan
0070: 52 69 66 66 20 20 20 4D  00 00 80 00              Riff   M ....
```

Builtin: math <expression>
ma <expression>

Prints 16-bit evaluated result of expression, which is composed of numeric values (as described at the start of this section) connected with '+' or '-' operators.

The evaluated result is printed in HEX and decimal, as both positive and negative numbers.

Example (from an actual session):

```
Z80 A>math 0-7ff

     HEX: F801H  -07FFH     Dec: 63489  -02047

Z80 A>ma secondary_fcb-primary_fcb

     HEX: 0010H  -FFF0H     Dec: 00016  -65520

Z80 A>ma 'A-40+'a

     HEX: 0062H  -FF9EH     Dec: 00098  -65438
```

BUILTIN COMMANDS: CP/M ENVIRONMENT CONTROL

Builtin: args <tail>
ar <tail>

Formats default FCB's at 5Ch and 6Ch as well as default DMA at 80h per command tail, exactly as if <tail> had followed a Z80 command (.COM) filename.

This command is most useful for "filling in" a command tail for Z80 code that has been "read" into CP/M memory and which will look for command-line arguments.

For instance, you may want to debug Digital Research's DDT.COM program, while telling DDT to load file FOO.COM. You would first load DDT via

```
read 100 ddt.com
```

You would then fill in DDT's command-line arguments with

```
args foo.com
```

When executed, DDT would then see "FOO.COM" in both the first FCB at 5Ch and as a raw command tail at 80h.

Example (from an actual session):

```
Z80 A>args testfile001 file0002

Z80 A>d primary_fcb secondary_fcb+.15

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0050:                                     00 54 45 53              .TES
0060: 54 46 49 4C 45 30 30 31  00 00 00 00 00 46 49 4C TFILE001 .....FIL
0070: 45 30 30 30 32 20 20 20  00 00 00 00             E0002    ....

Z80 A>d 80 9f

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0080: 15 20 54 45 53 54 46 49  4C 45 30 30 31 20 46 49 . TESTFI LE001 FI
0090: 4C 45 30 30 30 32 0D 00  00 00 00 00 00 00 00 00 LE0002.. ........

Z80 A>args foo.c -n -b -v
```

```
     Z80 A>d primary_fcb secondary_fcb+.15

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0050:                                     00 46 4F 4F             .FOO
0060: 20 20 20 20 20 43 20 20  00 00 00 00 00 2D 4E 20      C   .....-N
0070: 20 20 20 20 20 20 20 20  00 00 00 00                      ....

     Z80 A>d 80 9f

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0080: 0F 20 46 4F 4F 2E 43 20  2D 4E 20 2D 42 20 2D 56 . FOO.C  -N -B -V
0090: 0D 45 30 30 30 32 0D 00  00 00 00 00 00 00 00 00 .E0002.. ........
```

BUILTIN COMMANDS: CP/M ENVIRONMENT CONTROL

Builtin: coldboot! ldboot!
cold!

Cold Boots CP/M memory by zeroing the 64K CP/M Segment, installing the BIOS and BDOS hooks, formatting the first page of memory just as CP/M would, etc.

When you first start up the Emulator, the CP/M Segment has already been Cold Booted.

You may use this command to clean things up when you suspect that Z80 software may have corrupted the BIOS or BDOS hooks, garbaged page zero, or whatever.

If a Z80 program exits with an Emulator message to the effect that the program requests termination via Cold Boot, then you should use this command to do it. The Z80 program probably had a good reason for asking for a Cold Boot.

Example:

```
Z80 A>coldboot!

     *** CP/M Segment COLDBOOTED ***
```

BUILTIN COMMANDS: CP/M ENVIRONMENT CONTROL

Builtin: terminal [ON | OFF]
term [ON | OFF]

Enables or disables the builtin VT52 "emulated terminal".

If ON is specified, then the emulated CP/M BIOS routines that deal with the console will perform terminal emulation as described previously. Such I/O goes straight to the IBM PC screen (via the IBM PC ROM BIOS routines), and is therefore never seen by PCDOS.

If OFF is specified, then the emulated CP/M BIOS routines that deal with the console will not emulate anything. They will simply act as if the equivalent CP/M BDOS function had been called. This means that CP/M BIOS terminal I/O will go to PCDOS for handling. This makes it available for PCDOS I/O redirection. It also allows a PCDOS emulator (like ANSI.SYS) to become the CP/M terminal emulator.

If no argument is specified, then the current state of emulation is simply reported.

Example:

```
Z80 A>term

Terminal Emulation is OFF (CP/M BIOS console goes to PCDOS)

Z80 A>terminal on

Terminal Emulation is ON (via CP/M BIOS)

Z80 A>terminal off

Terminal Emulation is OFF (CP/M BIOS console goes to PCDOS)
```

BUILTIN COMMANDS: CP/M ENVIRONMENT CONTROL

Builtin: read [n] <filename.typ>
r [n] <filename.typ>

Reads data from specified file into the CP/M Segment, at address n.

If n is absent, then it defaults to 0100.

This is the primary method used to load Z80 software for debugging. It is normally used to read Z80 command (.COM) files into memory, but this command will also read raw data files with no problem.

If <filename.typ> has a filetype of .HEX, then the file is assumed to be in standard Intel HEX format, and is loaded into memory at addresses specified by the HEX records. In such a case, the n argument may have no meaning.

The n argument is a numeric value of the sort described at the beginning of this section.

> NOTE: If a read of a file causes data to be read below 0100h or above FD00h, then the CP/M environment will be clobbered. If you're developing non-CP/M Z80 code, then who cares. If you're reading in a file that expects to call CP/M, however, then running the thing with a clobbered CP/M environment just might scramble the brains of the imaginary Z80.

This command is used by us at CCS to load test versions of our Z80 software, which we leave in .HEX format when we run L80.

Example:

```
Z80 A>read 100 820init.com

     *** Low = 0100H  Next = 0380H
     *** Z80 DMA, PC and Stack automatically set for .COM file

Z80 A>write 100 360 820init.hex

     Writing HEX records for 0100H thru 0360H (609 bytes) to file
     '820INIT.HEX'

Z80 A>read 820init.hex

     *** .HEX file Starting Address = 0100H
     *** Low = 0100H  Next = 0360H
```

Builtin: write nlow nhigh <filename.typ>
w nlow nhigh <filename.typ>

Write a block of CP/M memory to a disk file.

The bounds of the block to be written are specified by nlow and nhigh, which are numeric values as described at the start of this section.

The specified block of memory is written from the CP/M Segment to the specified file. It is written as a pure memory image, unless a .HEX extension is supplied.

Specifying a filename of type .HEX will cause an Intel HEX file to be written. The final record of the generated HEX file is the special HEX record which specifies the starting execution address of the program. This address is assumed to be nlow.

Example:

```
Z80 A>read 100 820init.com

     *** Low = 0100H  Next = 0380H
     *** Z80 DMA, PC and Stack automatically set for .COM file

Z80 A>write 100 360 820init.hex

     Writing HEX records for 0100H thru 0360H (609 bytes) to file
     '820INIT.HEX'

Z80 A>read 820init.hex

     *** .HEX file Starting Address = 0100H
     *** Low = 0100H  Next = 0360H
```

BUILTIN COMMANDS: CP/M ENVIRONMENT CONTROL

Builtin: submit <filename.typ>
sub <filename.typ>

Switches input (for Emulator commands only) to the specified file.

This is roughly equivalent to the CP/M SUBMIT.COM program, except that it is built into the Emulator.

Z80 application input via BDOS and BIOS does not get switched. Only Emulator input is switched. Submit files cannot be nested.

Input reverts to the standard input (as defined by PCDOS) when EOF is detected on the specified file.

> NOTE: If a keypress is detected during submit file processing but outside of Z80 operation (i.e. - when the Emulator is expecting a command), then the submit file is aborted. This is how you cancel a submit file - just press SPACE while it is running.

Example:

```
Z80 A>submit script
Z80 A>sub c:\cpm\autoexec.z80          sub c:\cpm\autoexec.z80
```

```
Builtin:  list [n1 [n2]] [>outfile | >>outfile]
          list prologue n1 n2 [>outfile | >>outfile]
          list include [A][O][F]
          l [n1 [n2]] [>outfile | >>outfile]
          l prologue n1 n2 [>outfile | >>outfile]
          l include [A][O][F]
```

This is a multi-purpose disassembly command, which does one of three things (depending upon the arguments that are present):

Lists disassembled Z80 object code (first form).

Generates an assembler prologue for the given range of Z80 object code (second form).

Specifies fields to be included in disassembly lines (third form).

```
Format:  list [n1 [n2]] [>outfile | >>outfile]
         l [n1 [n2]] [>outfile | >>outfile]
```

This form causes the disassembly of Z80 object code from CP/M address n1 through CP/M address n2 (both of which are numeric values as defined at the start of this section). If n2 is absent, then n2 is assumed to be n1+22. If n1 is absent, then it defaults to the next address to be disassembled (as left by last list command).

The generated disassembly is sent to (">outfile") or appended to (">>outfile") the specified output file. If an output file is not specified (no ">" character present), then the generated disassembly goes to the standard output as defined by PCDOS (normally the screen).

NOTE: Disassembly speed decreases as the size of the control table increases. It may also be slowed by PCDOS being so lethargic with screen output. When disassembling the Radio Shack Model 100's memory (which required a control table of about 42K), for instance, we were only getting about 2 lines of disassembled code per second.

Generated source code is suitable for input to M80.COM in .Z80 mode. Note, however, that any RST instructions use as their argument the final address being jumped to, not the RST number (as is customary). Also, there is no END statement supplied.

Format: list prologue n1 n2 [>outfile | >>outfile]
l prologue n1 n2 [>outfile | >>outfile]

This form generates assembler source code suitable for inclusion at the front of a Z80 source file. The generated source code is the prologue for a given block of Z80 object code, defined by CP/M addresses n1 and n2. Both n1 and n2 are numeric values as defined at the start of this section.

The generated prologue contains equates for the various ASCII control characters. Additionally, it contains equated labels for any CP/M addresses that are referenced by the specified block of Z80 code (from n1 to n2), but not contained within it.

Such a prologue is generally needed at the front of any sizeable Z80 disassembly.

The generated prologue is sent to (">outfile") or appended to (">>out-file") the specified output file. If an output file is not specified, (no ">" character present), then the generated prologue goes to the standard output as defined by PCDOS (normally the screen).

Format: list include [A][O][F]
l include [A][O][F]

Specifies the fields to be included in disassembled Z80 instructions.

If the "A" arg is present, then CP/M addresses will be included at the left of disassembled instructions.

If the "O" argument is present, then raw opcodes will be included after the address (if present) but before the Z80 mnemonic.

If the "F" argument is present, then the disassembled Z80 instruction will include a comment explaining the instruction's possible effect(s) on the Z80 flags.

Arguments after "include" may be whole words. Only the first character is checked.

If no arguments appear after "include", then the disassembled Z80 instruction will consist only of the mnemonic field.

Example:

```
Z80 A>list include addresses
Z80 A>list include
Z80 A>l prologue 100 7ff
Z80 A>list prologue 0 ffff >model100.rom
Z80 A>list 0 ffff >>model100.rom
Z80 A>list
```

NOTE: The next section contains a sample Emulator session which generates source code from object code. Refer to it for more detail about this and related Resource commands.

BUILTIN COMMANDS: RESOURCE FACILITY COMMANDS

```
Builtin:  control list [n]
          control clear
          control read <filepath>
          control write <filepath>
          control n     i | b | w | t | s | c
          c list [n]
          c clear
          c read <filepath>
          c write <filepath>
          c n     i | b | w | t | s | c
```

This is a multi-purpose command which manipulates the disassembly control table.

The disassembly control table holds CP/M addresses and any or all of the following which are associated with each address:

The data type of the Z80 object code at this address. This is called a "control break".

A symbolic label to be associated with the address.

A comment to be associated with the address.

This control table is used by the disassembler (the list command), and list tells it how to format the source code while disassembling.

The various data type control breaks that may be associated with a CP/M address are as follows:

Instructions (executable Z80 code, disassembled as mnemonics)

Bytes (disassembled as DB pseudo-ops)

Words (disassembled as DW pseudo-ops, multiple entries per line)

Table of Words (disassembled as DW pseudo-ops, one per line)

Storage (disassembled as DS with argument large enough to bring it up to next control entry)

This is how the disassembler knows which parts of your Z80 object code are instructions, which are data, and which are irrelevant buffers etc.

The specific formats are described below.

Format: control list [n]
        c list [n]

This format of the command causes the Emulator to display all control table information that is currently known to it. If numeric value n (see definition of legal numeric values at start of this section) is present, then only control information associated with CP/M addresses greater than or equal to n are listed.

The list includes any control breaks, labels, and comments associated with the various CP/M addresses.

Format: control clear
        c clear

This format of the command clears out the control table, so that nothing is known about the Z80 object code. No CP/M addresses, control breaks, labels or comments are defined after this command is given.

Format: control read <filepath>
        c read <filepath>

This format of the command causes the Emulator to read a control table from the specified filename. The file must have been created by a "control write <filepath>" command.

A currently-defined control table is cleared before the new one is read from disk. It is not possible to merge control tables using this command.

Format: control write <filepath>
c write <filepath>

This command writes the current control table to the specified file, for later input via the "control read <filepath>" command. All known data is written - CP/M addresses, control breaks, labels and comments.

Format: control n i | b | w | t | s | c
c n i | b | w | t | s | c

This format of the command is the real workhorse of control table maintenance. It associates a Z80 data type with CP/M address n, which is a numeric value as defined at the start of this section.

The argument following CP/M address n is a directive to the disassembler (the list command), and may be any one of the following (note that only the first character is required):

Instructions: switch to Z80 mnemonics when you get to this address.

Bytes: switch to DB pseudo-ops when you get to this address.

Words: switch to DW pseudo-ops (multiple per line) when you get to this address.

Table of words: switch to DW pseudo-ops (one per line) when you get to this address. This is useful for jump tables, etc, where you want the source code to be neatly arranged.

Storage: do a single DS (define storage) pseudo-op when you get to this address, and make the size field big enough to take you up to the next control break address (or the end of the disassembly, whichever is lower).

You may also specify a special argument, which is handled immediately and never gets to the disassembler:

Clear: Clear this address's control break data type. This does not remove an associated label or comment. It just undoes any control break (of one of the above types) associated with this CP/M address.

Example:

```
Z80 A>control list
Z80 A>c clear
Z80 A>c read b:\model100\model100.ctl
Z80 A>control write foo.ctl
Z80 A>c read xyz
Z80 A>control 100 Instructions
Z80 A>c 103 b
Z80 A>c table_start+1 S
Z80 A>CONTROL bios_address W
Z80 A>c FF00 Table
Z80 A>c 103 clear
```

NOTE: The next section contains a sample Emulator session which generates source code from object code. Refer to it for more detail about this and related Resource commands.

BUILTIN COMMANDS: RESOURCE FACILITY COMMANDS

Builtin: label n [label_name]
label autogen n1 n2
= n [label_name]
= autogen n1 n2

This command controls the assignment of symbolic label names to various CP/M addresses. Symbolic labels may be used as numeric values in many Emulator commands. They are also used by the disassembler when creating source code from Z80 object code.

Format: label n [label_name]
= n [label_name]

This format associates label_name with CP/M address n, which is a numeric value as described at the start of this section.

Label names may be up to 32 characters in length. They must contain only alphanumeric characters and the underscore character "_".

If label_name is absent in this command, then any existing label name associated with the specified CP/M address is simply deleted.

Format: label autogen n1 n2
= autogen n1 n2

This format causes the Emulator to automatically generate labels (of format AUTOxxxx) for all unlabeled CP/M addresses that are referenced by the block of Z80 code that starts at n1 and ends at n2 (both of which are numeric values as defined at the start of this section).

Labels that are generated are automatically entered into the current control table. Existing labels will not be altered.

This is a quick way to create labels. It can be useful for rapid generation of readable source code from Z80 object code. You should, however, define any recognizable labels before using this command. You should also make sure that you have pretty accurately defined all control breaks that apply to the specified block of Z80 code. There's nothing worse than getting a block of data confused with instructions, and having this command generate a few hundred bogus labels by misinterpreting the Z80 code.

So this is generally the last thing that you do before deciding that you have finished disassembling a complete Z80 program.

NOTE: 16-bit literals will be ignored. There's no way - short of human inspection - to tell if the 16-bit value is meant to be a Z80 address or just a binary value (like a loop counter). In previous versions of the Emulator, there were too many bogus labels being autogen'd from 16-bit literals. Now it is up to you to decide whether a particular 16-bit literal should have a label associated with it. Specifically, the following instructions' 16-bit literals are ignored:

```
LD   IX,nn
LD   IY,nn
LD   BC,nn
LD   DE,nn
LD   HL,nn
LD   SP,nn
```

Example:

```
Z80 A>= 100 program_entry
Z80 A>label 7ff program_end
Z80 A>= 0 warm_boot_jump
Z80 A>= 5C FCB1
Z80 A>= 5 BDOS
Z80 A>label autogen 100 7ff
```

NOTE: The next section contains a sample Emulator session which generates source code from object code. Refer to it for more detail about this and related Resource commands.

Builtin: comment n ["text"]
; n ["text"]

Associates source-code comment "text" with CP/M address n, which is a numeric value as defined at the start of this section. "text" may be up to 254 characters long.

If "text" is absent, then any existing comment associated with address n is simply deleted.

Note that "text" must be enclosed in double quotes. It may include escape sequences as follows:

\\	Single "\" char
\0	NUL byte
\b or \B	Backspace char
\t or \T	Tab char
\n or \N	Linefeed char
\r or \R	Return char
\'	Single quote char
\"	Double quote char
\xFF or \XFF	Byte with HEX value FF

These escape codes made be used to make the comment more readable.

When a comment is detected by the disassembler, it will be printed in one of two places:

As a line comment, after the mnemonic. Such a comment replaces the flags comment field (if present). If "text" starts with other than a "\n" escape, then the comment is printed in this manner.

As a multi-line comment, on lines before the instruction. If "text" starts with a "\n" escape, then the comment will be printed in this manner. Blank comment lines (starting with ";") are automatically provided before and after the comment, and ";" characters are inserted after every "\n" or "\r" found in the "text" string.

If you want to create a nice-looking multi-line comment, then imbed "\n" escapes as line delimiters, and "\t" escapes to line things up on succeeding lines.

If your comment is intended to be a single-line comment appended to the disassembled instruction's mnemonic, then avoid imbedded "\n" and "\r" escapes.

Example:

```
Z80 A>comment 100 "\nStart of main program"
Z80 A>; 0 "Jump to BIOS Warm, Start"
Z80 A>; 100
Z80 A>comment ff00 "\nBIOS Jump Table\n\n\t3 bytes per JMP"
```

NOTE: The next section contains a sample Emulator session which generates source code from object code. Refer to it for more detail about this and related Resource commands.

USING THE RESOURCE BUILTIN COMMANDS

This section presents an example of re-sourcing a piece of Z80 object code. It demonstrates the use of the various Emulator builtin commands that deal with regenerating source code from object code.

We were recently presented with a piece of software which (wouldn't you know it) existed only in object form on a Xerox 820 CP/M system. The owner of this little utility really wanted to move it to a 16-bit Hyperion system, cause the utility sets up a Z80 SIO and that's what the Hyperion has - an SIO.

So he was wondering - could we regenerate the source code for this little utility?

> NOTE: The first step is to read the thing into Z80 memory with the Emulator, and clear any previous resource control breaks.

```
Z80 C>read 100 820init.com

*** Low = 0100H  Next = 0380H
*** Z80 DMA, PC and Stack automatically set for .COM file
Z80 C>control clear

Z80 C>list include addresses opcodes
```

USING THE RESOURCE BUILTIN COMMANDS

NOTE: The next thing to do is take a quick look at the program, and get a general feel for what it does.

```
Z80 C>list

L0100:     11 45 02             LD     DE,L0245
L0103:     CD 40 02             CALL   L0240
L0106:     11 C8 02             LD     DE,L02C8
L0109:     CD 40 02             CALL   L0240
L010C:     CD 35 02             CALL   L0235
L010F:     FE 0D                CP     CR
L0111:     C2 16 01             JP     NZ,L0116
L0114:     3E 36                LD     A,'6'
L0116:     D6 30                SUB    '0'
```

NOTE: This isn't real promising. Let's dump the thing and look for obvious ASCII strings.

```
Z80 C>d 100 37f

ADDR  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F 01234567 89ABCDEF
----  -- -- -- -- -- -- -- --  -- -- -- -- -- -- -- -- -------- --------
0100: 11 45 02 CD 40 02 11 C8  02 CD 40 02 CD 35 02 FE .E.M@..H .M@.M5.~
0110: 0D C2 16 01 3E 36 D6 30  FE 00 DA 06 01 FE 0A D2 .B..>6V0 ~.Z..~.R
0120: 06 01 32 58 03 11 EF 02  CD 40 02 CD 35 02 FE 0D ..2X..o. M@.M5.~.
0130: C2 35 01 3E 4E FE 45 CA  47 01 FE 4F CA 47 01 FE B5.>N~EJ G.~OJG.~
0140: 4E CA 47 01 C3 25 01 32  5A 03 11 16 03 CD 40 02 NJG.C%.2 Z....M@.
0150: CD 35 02 FE 0D C2 5A 01  3E 38 D6 30 FE 07 CA 69 M5.~.BZ. >8V0~.Ji
0160: 01 FE 08 CA 69 01 C3 4A  01 32 59 03 3A 58 03 FE .~.Ji.CJ .2Y.:X.~
0170: 00 C2 79 01 3E 0F C3 B7  01 FE 01 C2 83 01 3E 0E .By.>.C7 .~.B..>.
0180: C3 B7 01 FE 02 C2 8D 01  3E 0C C3 B7 01 FE 03 C2 C7.~.B.. >.C7.~.B
0190: 97 01 3E 0A C3 B7 01 FE  04 C2 A1 01 3E 07 C3 B7 ..>.C7.~ .B!.>.C7
01A0: 01 FE 05 C2 AB 01 3E 06  C3 B7 01 FE 06 C2 B5 01 .~.B+.>. C7.~.B5.
01B0: 3E 05 C3 B7 01 3E 02 32  58 03 3A 5A 03 FE 45 C2 >.C7.>.2 X.:Z.~EB
01C0: C7 01 3E 03 C3 D3 01 FE  4F C2 D1 01 3E 01 C3 D3 G.>.CS.~ OBQ.>.CS
01D0: 01 3E 00 32 5A 03 3A 59  03 FE 07 CA EB 01 3E 60 .>.2Z.:Y .~.Jk.>`
01E0: 32 5B 03 3E C0 32 5C 03  C3 F5 01 3E 20 32 5B 03 2[.>@2\.  Cu.>2[.
01F0: 3E 40 32 5C 03 F3 3E 18  D3 06 D3 06 3E 01 D3 06 >@2\.s>. S.S.>.S.
0200: AF D3 06 3E 04 D3 06 3A  5A 03 C6 44 D3 06 3E 03 /S.>.S.: Z.FDS.>.
0210: D3 06 3A 5C 03 C6 01 D3  06 3E 05 D3 06 3A 5B 03 S.:\.F.S .>.S.:[.
0220: C6 8A D3 06 3E 47 D3 00  3A 58 03 D3 00 FB 11 3D F.S.>GS. :X.S.{.=
0230: 03 CD 40 02 C9 0E 01 CD  05 00 FE 60 D8 D6 20 C9 .M@.I..M  ..~`XVI
0240: 0E 09 C3 05 00 1A 49 4E  49 54 20 31 2E 30 20 66 ..C...IN  IT 1.0f
0250: 6F 72 20 58 65 72 6F 78  20 38 32 30 0D 0A 0A 0A or Xerox  820....
0260: 0D 0A 42 61 75 64 20 52  61 74 65 73 3A 0D 0A 31 ..Baud R ates:..1
0270: 39 32 30 30 20 3D 20 30  0D 0A 39 36 30 30 20 20 9200 = 0 ..9600
```

```
0280: 3D 20 31 0D 0A 34 38 30  30 20 20 3D 20 32 0D 0A = 1..480 0  = 2..
0290: 32 34 30 30 20 20 3D 20  33 0D 0A 31 32 30 30 20 2400  =  3..1200
02A0: 20 3D 20 34 0D 0A 20 36  30 30 20 20 3D 20 35 0D  = 4.. 6 00  = 5.
02B0: 0A 20 33 30 30 20 20 3D  20 36 0D 0A 20 31 31 30 . 300  =  6.. 110
02C0: 20 20 3D 20 37 0D 0A 24  0D 0A 53 65 6C 65 63 74   = 7..$ ..Select
02D0: 20 62 61 75 64 20 72 61  74 65 20 20 20 20 20 20  baud ra te
02E0: 20 20 20 20 20 28 31 2D  39 29 3A 20 36 08 24 0D      (1- 9): 6.$.
02F0: 0A 53 65 6C 65 63 74 20  70 61 72 69 74 79 20 20 .Select  parity
0300: 28 4F 64 64 2C 20 45 76  65 6E 2C 20 4E 6F 6E 65 (Odd, Ev en, None
0310: 29 3A 20 4E 08 24 0D 0A  53 65 6C 65 63 74 20 77 ): N.$.. Select w
0320: 6F 72 64 20 6C 65 6E 67  74 68 20 20 20 20 20 20 ord leng th
0330: 28 37 20 6F 72 20 38 29  3A 20 38 08 24 0D 0A 43 (7 or 8) : 8.$..C
0340: 6F 6D 6D 75 6E 69 63 61  74 69 6F 6E 73 20 70 6F ommunica tions po
0350: 72 74 20 73 65 74 2E 24  00 00 00 00 00 00 00 00 rt set.$ ........
0360: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
0370: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
```

NOTE: Well, there are some strings that are terminated with dollar signs. This is a heavy hint that these strings are to be displayed by BDOS function 9. Let's start our control table by filling in what we know so far...

```
Z80 C>c 100 instructions

Z80 C>= 100 startup

Z80 C>c 245 b

Z80 C>= 245 init_msg

Z80 C>= 2c8 baud_prompt

Z80 C>= 2ef parity_prompt

Z80 C>= 316 databits_prompt

Z80 C>= 33d wrapup_msg
```

NOTE: Now let's look at the first code bytes again, and see if they make any more sense.

```
Z80 C>l 100


                STARTUP:
L0100:     11 45 02             LD     DE,INIT_MSG
L0103:     CD 40 02             CALL   L0240
L0106:     11 C8 02             LD     DE,BAUD_PROMPT
L0109:     CD 40 02             CALL   L0240
L010C:     CD 35 02             CALL   L0235
L010F:     FE 0D                CP     CR
L0111:     C2 16 01             JP     NZ,L0116
L0114:     3E 36                LD     A,'6'
L0116:     D6 30                SUB    '0'
```

NOTE: So the routine at 0240h does something with a "$"-terminated string. Need we guess?

```
Z80 C>l 240

L0240:     0E 09                LD     C,TAB
L0242:     C3 05 00             JP     L0005

                INIT_MSG:
                     DB     SUB,"INIT 1.0 for Xero"
L0245:     1A 49 4E 49
L0249:     54 20 31 2E
L024D:     30 20 66 6F
L0251:     72 20 58 65
L0255:     72 6F
```

NOTE: First of all, note how the above disassembly changed from instructions to data, just as we told it to. Note also that this routine at 0240h is simple enough to document...

```
Z80 C>= 5 bdos

Z80 C>; 240 "\nPrint $-terminated string at (DE)"

Z80 C>c 240 i
```

```
Z80 C>= 240 print_string

Z80 C>l 240


                PRINT_STRING:

                ;
                ; Print $-terminated string at (DE)
                ;
L0240:     0E 09                LD     C,TAB
L0242:     C3 05 00             JP     BDOS

                INIT_MSG:
                     DB     SUB,"INIT 1.0 for Xero"
L0245:     1A 49 4E 49
L0249:     54 20 31 2E
L024D:     30 20 66 6F
L0251:     72 20 58 65
L0255:     72 6F
```

NOTE: OK, one routine down. Referring back to our disassembly of the startup code, let's see what else we can figure out.

```
Z80 C>= 106 get_baud_rate

Z80 C>l startup


                STARTUP:
L0100:     11 45 02             LD     DE,INIT_MSG
L0103:     CD 40 02             CALL   PRINT_STRING

                GET_BAUD_RATE:
L0106:     11 C8 02             LD     DE,BAUD_PROMPT
L0109:     CD 40 02             CALL   PRINT_STRING
L010C:     CD 35 02             CALL   L0235
L010F:     FE 0D                CP     CR
L0111:     C2 16 01             JP     NZ,L0116
L0114:     3E 36                LD     A,'6'
L0116:     D6 30                SUB    '0'
```

NOTE: We have another routine to decipher - at 0235h.

```
Z80 C>l 235

L0235:     0E 01                LD     C,SOH
L0237:     CD 05 00             CALL   BDOS
L023A:     FE 60                CP     '`'
L023C:     D8                   RET    C
L023D:     D6 20                SUB    ' '
L023F:     C9                   RET

               PRINT_STRING:

               ;
               ; Print $-terminated string at (DE)
               ;
L0240:     0E 09                LD     C,TAB
L0242:     C3 05 00             JP     BDOS

               INIT_MSG:
                    DB     SUB,"INIT 1"
L0245:     1A 49 4E 49
L0249:     54 20 31
```

NOTE: Notice how the disassembly continued way past the end of our routine. That's because we didn't give an ending address. At any rate, the routine at 0235h appears to call BDOS to get a keypress, then force it to uppercase. It contains a slight bug, but our job right now is to re-source it, not fix it.

```
Z80 C>= 235 get_bdos_keypress

Z80 C>c 235 i

Z80 C>; 235 "\nReturn next keypress as Uppercase char in A-reg"

Z80 C>; 237 "Use BDOS to get next keypress"

Z80 C>; 23a "Is it lowercase char?"

Z80 C>; 23c "No, return it as-is"

Z80 C>; 23d "Yes, convert to uppercase"
```

```
Z80 C>l 235


                GET_BDOS_KEYPRESS:

                ;
                ; Return next keypress as Uppercase char in A-reg
                ;
L0235:    0E 01               LD    C,SOH
L0237:    CD 05 00            CALL  BDOS    ;Use BDOS to get next
                                                keypress
L023A:    FE 60               CP    '`'     ;Is it lowercase char?
L023C:    D8                  RET   C       ;No, return it as-is
L023D:    D6 20               SUB   ' '     ;Yes, convert to uppercase
L023F:    C9                  RET

                PRINT_STRING:

                ;
                ; Print $-terminated string at (DE)
                ;
L0240:     0E 09                 LD     C,TAB
L0242:     C3 05 00              JP     BDOS

                INIT_MSG:
                     DB     SUB,"INIT 1"
L0245:     1A 49 4E 49
L0249:     54 20 31
```

NOTE: OK, our little routines are understood and documented. Back to the main code, and add comments that clarify things.

```
Z80 C>; 100 "Give intro screen"

Z80 C>; 106 "Ask for baudrate value" ; 106 "Ask for baudrate value"

Z80 C>; 10f "RETURN only?"

Z80 C>; 111 "No, look at keypress"

Z80 C>; 114 "Yes, use default value"

Z80 C>= 116 edit_baud_rate
```

```
Z80 C>l get_baud_rate 12b


                GET_BAUD_RATE:
L0106:   11 C8 02           LD     DE,BAUD_PROMPT ;Ask for baudrate
                                                        value
L0109:   CD 40 02           CALL   PRINT_STRING
L010C:   CD 35 02           CALL   GET_BDOS_KEYPRESS
L010F:   FE 0D              CP     CR          ;RETURN only?
L0111:   C2 16 01           JP     NZ,EDIT_BAUD_RATE ;No, look at
                                                        keypress
L0114:   3E 36              LD     A,'6'       ;Yes, use default
                                                        value

                EDIT_BAUD_RATE:
L0116:   D6 30              SUB    '0'
L0118:   FE 00              CP     NUL
L011A:   DA 06 01           JP     C,GET_BAUD_RATE
L011D:   FE 0A              CP     LF
L011F:   D2 06 01           JP     NC,GET_BAUD_RATE
L0122:   32 58 03           LD     (L0358),A
L0125:   11 EF 02           LD     DE,PARITY_PROMPT
L0128:   CD 40 02           CALL   PRINT_STRING
L012B:   CD 35 02           CALL   GET_BDOS_KEYPRESS
```

NOTE: And so on, and so on, and so on....

We'll not present all of the output here. Although the session actually lasted less than half an hour, the output is huge. Instead, let us skip ahead to the point where we've finished defining our control table.

NOTE: At this point - having done all of this work to construct documentation of the object code - we should save the control table to disk!

```
Z80 C>control write 820init.ctl
```

NOTE: Having done that, we must prepare things for the actual generation of the .ASM file. Specifically, we should exclude addresses, opcodes, and flags from disassembly source lines.

```
Z80 C>list include
```

NOTE: Now we can disassemble the object code to disk as source code. First we write the prologue, and then the program itself:

```
Z80 C>list prologue 100 360 >820init.asm

Z80 C>list 100 360 >>820init.asm
```

NOTE: Having done that, let's test things by running the .ASM file through M80.COM...

```
Z80 C>m80 820init.rel,820init.prn=820init.asm
%No END statement
%No END statement

No Fatal error(s)
```

That's it. We're done. The assembler version of 820INIT.COM has been written and verified.

For your enjoyment, we have included with Z80MU the various 820INIT files that were used or created by the above session. Feel free to examine all of these files, especially 820INIT.ASM and 820INIT.PRN. They give you a good idea of the quality of source code that can be recreated from a given object program.

Appendix A: Layout of CP/M Segment

Appendix A: Layout of Simulated CP/M Segment (as set by Cold Boot)

```
0000:     JMP  0FF03H         ;to our fake BIOS
0003:     DB   ?              ;IOBYTE
0004:     DB   ?              ;Login Byte (drive, User Number)
0005:     JMP  0FEFEH         ;to our fake BDOS
005C:     DB   ?              ;Default FCB
0080:     DB   ?              ;Default DMA and Command Tail
0100:     DB   ?              ;start of TPA
FEFD:     DB   ?              ;last byte of TPA
FEFE:     HALT                ;our BDOS hook
FEFF:     RET                 ;return from BDOS
FF00:     JMP  0FF80H         ;BIOS COLD BOOT vector
FF03:     JMP  0FF82H         ;BIOS WARM BOOT vector
FF06:     JMP  0FF84H         ;BIOS console status
FF09:     JMP  0FF86H         ;BIOS console input
FF0C:     JMP  0FF88H         ;BIOS console output
FF0F:     JMP  0FF8AH         ;BIOS list output
FF12:     JMP  0FF8CH         ;BIOS punch output
FF15:     JMP  0FF8EH         ;BIOS reader input
FF18:     JMP  0FF90H         ;BIOS home disk
FF1B:     JMP  0FF92H         ;BIOS select disk
FF1E:     JMP  0FF94H         ;BIOS set track
FF21:     JMP  0FF96H         ;BIOS set sector
FF24:     JMP  0FF98H         ;BIOS set DMA address
FF27:     JMP  0FF9AH         ;BIOS read sector
FF2A:     JMP  0FF9CH         ;BIOS write sector
FF2D:     JMP  0FF9EH         ;BIOS list status
FF30:     JMP  0FFA0H         ;BIOS (unimplemented)
FF33:     JMP  0FFA2H         ;BIOS (unimplemented)
FF36:     JMP  0FFA4H         ;BIOS (unimplemented)
FF39:     JMP  0FFA6H         ;BIOS (unimplemented)
FF3C:     JMP  0FFA8H         ;BIOS (unimplemented)
FF3F:     JMP  0FFAAH         ;BIOS (unimplemented)
FF42:     JMP  0FFACH         ;BIOS (unimplemented)
FF45:     JMP  0FFAEH         ;BIOS (unimplemented)
FF48:     JMP  0FFB0H         ;BIOS (unimplemented)
FF4B:     JMP  0FFB2H         ;BIOS (unimplemented)
FF4E:     JMP  0FFB4H         ;BIOS (unimplemented)
FF51:     JMP  0FFB6H         ;BIOS (unimplemented)
FF54:     JMP  0FFB8H         ;BIOS (unimplemented)
FF57:     JMP  0FFBAH         ;BIOS (unimplemented)
FF5A:     JMP  0FFBCH         ;BIOS (unimplemented)
FF5D:     JMP  0FFBEH         ;BIOS (unimplemented)
FF60:     JMP  0FFC0H         ;BIOS (unimplemented)
FF63:     JMP  0FFC2H         ;BIOS (unimplemented)
FF66:     JMP  0FFC4H         ;BIOS (unimplemented)
```

Appendix A: Layout of CP/M Segment

```
FF69:     JMP  0FFC6H         ;BIOS (unimplemented)
FF6C:     JMP  0FFC8H         ;BIOS (unimplemented)
FF6F:     JMP  0FFCAH         ;BIOS (unimplemented)
FF72:     JMP  0FFCCH         ;BIOS (unimplemented)
FF75:     JMP  0FFCEH         ;BIOS (unimplemented)
FF78:     JMP  0FFD0H         ;BIOS (unimplemented)
FF7B:     JMP  0FFD2H         ;BIOS (unimplemented)

FF80:     HALT                ;BIOS COLD BOOT hook
FF81:     RET
FF82:     HALT                ;BIOS WARM BOOT hook
FF83:     RET
FF84:     HALT                ;BIOS console status
FF85:     RET
FF86:     HALT                ;BIOS console input
FF87:     RET
FF88:     HALT                ;BIOS console output
FF89:     RET
FF8A:     HALT                ;BIOS list output
FF8B:     RET
FF8C:     HALT                ;BIOS punch output
FF8D:     RET
FF8E:     HALT                ;BIOS reader input
FF8F:     RET
FF90:     HALT                ;BIOS home disk
FF91:     RET
FF92:     HALT                ;BIOS select disk
FF93:     RET
FF94:     HALT                ;BIOS set track
FF95:     RET
FF96:     HALT                ;BIOS set sector
FF97:     RET
FF98:     HALT                ;BIOS set DMA address
FF99:     RET
FF9A:     HALT                ;BIOS read sector
FF9B:     RET
FF9C:     HALT                ;BIOS write sector
FF9D:     RET
FF9E:     HALT                ;BIOS list status
FF9F:     RET
FFA0:     HALT                ;BIOS (unimplemented)
FFA1:     RET
FFA2:     HALT                ;BIOS (unimplemented)
FFA3:     RET
FFA4:     HALT                ;BIOS (unimplemented)
FFA5:     RET
FFA6:     HALT                ;BIOS (unimplemented)
FFA7:     RET
```

```
FFA8:     HALT                ;BIOS (unimplemented)
FFA9:     RET
FFAA:     HALT                ;BIOS (unimplemented)
FFAB:     RET
FFAC:     HALT                ;BIOS (unimplemented)
FFAD:     RET
FFAE:     HALT                ;BIOS (unimplemented)
FFAF:     RET
FFB0:     HALT                ;BIOS (unimplemented)
FFB1:     RET
FFB2:     HALT                ;BIOS (unimplemented)
FFB3:     RET
FFB4:     HALT                ;BIOS (unimplemented)
FFB5:     RET
FFB6:     HALT                ;BIOS (unimplemented)
FFB7:     RET
FFB8:     HALT                ;BIOS (unimplemented)
FFB9:     RET
FFBA:     HALT                ;BIOS (unimplemented)
FFBB:     RET
FFBC:     HALT                ;BIOS (unimplemented)
FFBD:     RET
FFBE:     HALT                ;BIOS (unimplemented)
FFBF:     RET
FFC0:     HALT                ;BIOS (unimplemented)
FFC1:     RET
FFC2:     HALT                ;BIOS (unimplemented)
FFC3:     RET
FFC4:     HALT                ;BIOS (unimplemented)
FFC5:     RET
FFC6:     HALT                ;BIOS (unimplemented)
FFC7:     RET
FFC8:     HALT                ;BIOS (unimplemented)
FFC9:     RET
FFCA:     HALT                ;BIOS (unimplemented)
FFCB:     RET
FFCC:     HALT                ;BIOS (unimplemented)
FFCD:     RET
FFCE:     HALT                ;BIOS (unimplemented)
FFCF:     RET
FFD0:     HALT                ;BIOS (unimplemented)
FFD1:     RET
FFD2:     HALT                ;BIOS (unimplemented)
FFD3:     RET
FFD4:                         ;rest reserved for scratch use
```

Appendix B: Bugs and Future Plans

This section describes known bugs and otherwise strange Emulator activity. Please help us to improve the product by sending us your own bug reports, with as much detail as possible (including programs which demonstrate the bug).

These bugs will be fixed as time allows and user interest demands.

Issue: The disassembly (list) command can become quite pokey. As the control table grows, disassembly speed decreases. When disassembling the Model 100's ROM, for instance, the poor disassembler is faced with a control table that is something bigger than 42K in size. It may have to search most of the table for each memory reference. This yields a disassembly speed of a line or two a second. Boo! The control table search routines should be changed from a sequential to an indexed method.

Author's Response: Agreed. Will rewrite these for next release.

Issue: Some commands (notably those that accept a range of CP/M addresses) get confused if they are asked to wrap around the high end of the 64K CP/M segment.

Author's Response: Have applied fixes for most glaring problems. Would appreciate specifics (i.e. - what commands fail and how) to help track down the rest.

Issue: Internal I/O redirection (effected by the Emulator, not PCDOS) should be available for any Emulator command, not just the list command.

Author's Response: Agreed. Will look into it.

Appendix B: Bugs and Future Plans

Issue: The patch command ought to allow multiple entries per line, and ought to accept arguments on the patch command line without dropping into interactive patch mode.

Author's Response: Have provided left-handed solution by allowing a patch string on same line as patch command. True fix (with multiple byte values as independent args on command line) will take a while to implement.

Issue: The Emulator lacks a mini assembler. It needs one.

Author's Response: Agreed. We just don't have the memory left in the 64K Lattice code segment to do it right. Maybe after some major rework...

Issue: The Emulator's main program ought to be rewritten in Microsoft C version 3.0. This will speed it up considerably, and will make it a lot smaller.

Author's Response: Agreed. The major effort is in reworking the assembler subroutines. There are a lot of places where Lattice's calling convention and register usage is assumed.

Issue: It would be nice to somehow state to the Emulator that what is in CP/M memory is actually an 8080 or 8085 program. This would allow the 8085's RIM and SIM instructions, for example, to be properly disassembled (instead of being misinterpreted as Z80 relative jumps).

Author's Response: Agreed. The biggest obstacle to doing this is the same old bugaboo - no memory left in 64K code segment. Will put this off until we free up a couple of K in the code segment.

Appendix B: Bugs and Future Plans

Issue: The loading of Z80 .COM files to be executed - in fact, any file load that is done directly or indirectly by the "read" command - is just too slow.

Author's Response: Agreed! This should be redone to bypass Lattice C's slow I/O routines. This problem may go away if and when the Emulator is rewritten in Microsoft 'C' version 3.0.

Issue: The disassembler can get confused when a multi-byte opcode crosses a control break. In general, it reverts to DB pseudo ops up to the control break when this happens. Also, it's not too smart when disassembling code with addresses up around the 64K segment boundary.

Author's Response: I have tried to locate and eliminate peephole problems. The nature of the problem, however, goes to the very core of the disassembler as designed. This is not quickly (or cheaply) fixable.

Issue: There are too many CP/M applications which abort because they use BDOS functions 27/1Bh and 31/1Fh. These BDOS functions should be supported, even if they mean little on a PCDOS system.

Author's Response: Agreed. The only reason that they haven't been emulated is that I have yet to find a sensible writeup of just exactly what the various data formats are. DRI's writeup stinks. If someone can provide a clear explanation of just what they truly mean, then I'll emulate them.

Issue: There are more than 600 distinct instructions in the Z80. Have they all been validated as to the accuracy of the emulation?

Author's Response: So far, all we've done is run Z80 stuff and try to guess that it has run fine. All opcodes have been desk-checked. Not all opcodes have been tested. Most, in fact, have not even been executed. Would somebody please create a definitive test program? The diagnostics that we've tried (like Supersoft's) have proven to be inaccurate.

Appendix B: Bugs and Future Plans

Issue: There should be an option to allow pagination of disassembled object code.

Author's Response: Agreed. Will look into another list include option to specify pagination.

Issue: Expressions are currently limited to "+" and "-" operators. They should allow "*" and "/" also, and maybe even AND, OR, XOR, shifts, etc.

Author's Response: I disagree. The amount of work involved isn't justified by the benefit.

Issue: The Z80 IN and OUT instructions should have access to real 8088 I/O ports on the IBM PC.

Author's Response: Over my dead body.