CDL DEBUG I and II

User's Manual

Sept. 30, 1979

Z-80 and 8080/8085 Debugging Tool

Operates on Z-80 only

Table of Contents

1 Introduction to DEBUG I and II

2 Overview of ZBUG

- 2.1 Data Format
  - 2.1.1 Data MODE
  - 2.1.2 Data RADIX
    - 2.1.2.1 Data Display
    - 2.1.2.2 Data Type-in
    - 2.1.2.3 Address Display
- 2.2 Examining and Modifying Data
  - 2.2.1 Memory Data
    - 2.2.1.1 Display Data
    - 2.2.1.2 Replace Data
    - 2.2.1.3 Examine/Modify Data
    - 2.2.1.4 Search for Data
  - 2.2.2 Registers and Flags
- 2.3 Program Execution and Breakpoints
  - 2.3.1 Executing a Program
  - 2.3.2 Breakpoint During Execution
- 2.4 Tracing and Traps
  - 2.4.1 Tracing a Program
  - 2.4.2 Traps During Tracing

3 Starting Out

- 3.1 ZBUG's Operating Environment
- 3.2 Executing ZBUG
- 3.3 A Sample Session

CDL Debug I and II User's Manual
Table of Contents

4 The Commands - A Detailed Description

4.1 C - Calculate
4.2 D - Display
4.3 E - Examine
4.4 F - Fill
4.5 G - Goto
4.6 I - Instruction Interpret
4.7 L - List ASCII
4.8 M - Mode
4.9 O - Open File
4.10 P - Put String
4.11 Q - Quit
4.12 R - Radix
4.13 S - Set Trap/Conditional Display/Wait
4.14 T - Trace
4.15 X - Examine Register/Flag
4.16 Y - Search
4.17 Z - Zap CP/M fcb's

5 Going Beyond the Basics

5.1 The ZBUG Expression
5.1.1 _ Operator
5.1.2 +, -, !, and ^ Operators
5.1.3 *, /, @, &, <, and > Operators
5.1.4 ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE Operators
5.1.5 +, -, #, @, \, ^, and ! Unary Operators
5.1.6 "Symbols"
5.1.7 "Constants"
5.1.7.1 Numbers
5.1.7.2 Strings
5.1.7.3 Registers and Flags
5.1.7.4 Instructions
5.2 Advanced Ideas

Appendixes

A A Quick Reference to the Commands

B Error Messages

Section 1

Introduction to DEBUG I and II

Debug I is a subset of Debug II and throughout this manual a distinction will be made only when there is a difference between the two. Debug I and II each consist of two seperate programs:

Debug I

DEBUGI.COM----------ZBUG.IMG

Debug II

DEBUGII.COM----------ZBUG.REL

DEBUGI.COM is a program which loads the working debugger, ZBUG.IMG. DEBUGII.COM likewise loads ZBUG.REL. In this manual, reference to ZBUG pertains to the working portions of Debug II. It also pertains to Debug I unless specifically stated otherwise. The terms "Debug I" and "Debug II" refer to the overall debugger--i.e. both the loader and working debugger.

ZBUG is CDL's dynamic debugging utility, designed to facilitate assembly language programming. ZBUG, when used with CDL's Macro Assembler, provides a powerful and versatile set of techniques for developing assembly language programs.

ZBUG provides standard debugging tools including memory and register examination/modification and program execution with breakpoints. However, ZBUG extends these common tools considerably with user-controlled data formatting, powerful expression evaluation for user-entered data, and extremely flexible trap capability with tracing.

This manual is intended as a guide for the user of ZBUG, beginning and experienced. For this reason, the sections of the manual are organized for general ease of access to basic and specific information.

Section 2 is provided as a guide to the first-time user, describing the basic features of ZBUG in a very general manner. It is not intended as a complete description, by any means, but is intended to give the flavor of ZBUG's possibilities.

Section 3 helps to demonstrate the use of ZBUG, indicating how to start execution of ZBUG, and giving a sample session to show the use of some of the basic commands. This section is intended to show the user how to begin using ZBUG as a useful tool.

Section 4 is a reference quide to the commands available in ZBUG, offering a complete description of each command's use and operation.

Section 5 is primarily concerned with giving the more experienced user an idea of the flexibility of the ZBUG command set. Included is a detailed discussion of the arithmetic expression capabilities of ZBUG for data type-in, and an advanced "session" with ZBUG to demonstrate ZBUG's muscle in handling different debugging situations. It is hoped that this section will lead to a more intimate understanding of ZBUG's abilities.

Finally, two appendixes are provided: a quick reference guide to the ZBUG command set, and a list of ZBUG's error messages.

This manual also documents ZBUG.IMG, a subset of ZBUG.REL which occupies less memory. Notes throughout the manual define the ZBUG.REL features not available in ZBUG.IMG.

Please note that this manual assumes that the ZBUG user is familiar with the Zilog Z80 CPU's register and flag organization, the Z80 instruction set, the Computer Design Labs Macro Assembler (the Z80 instruction mnemonics), and CP/M (tm-Digital Research) or Computer Design Labs' TPM (tm-CDL) Operating System. References are made to the following publications:

"Z80-CPU Technical Manual"
Zilog
10460 Bubb Road
Cupertino, California 95014

"An Introduction to CP/M Features and Facilities"
"CP/M Interface Guide"
Digital Research
Post Office Box 579
Pacific Grove, California 93950

"Macro I User's Manual" or "Macro II User's Manual"
Computer Design Labs
342 Columbus Ave.
Trenton, New Jersey 08629

Section 2

Overview of ZBUG

The following is a discussion of the main features of ZBUG's operation. It is intended to be a general introduction to the capabilities of ZBUG and a description of its use. For a complete description of each command available, please refer to Section 4.

The description of ZBUG is broken up into four general categories: data format, data examination and modification, execution and breakpoints, and tracing and traps. Each section mentions the commands offered in the particular category, and makes reference to the appropriate sub-section(s) of Section 4.

2.1 Data Format

Data can be interpreted in many ways. The length of the data (in 8 bit bytes for the Z80), its numeric representation, and whether considered as instructions or not are each important to the proper interpretation of data. ZBUG provides means for the user to interpret or specify data in different ways.

2.1.1 Data MODE

In ZBUG, memory data can be considered as a list of "cells" of a fixed length of one to four bytes, or as a list of Z80 instructions, each of varying length of one to four bytes.

In order to specify the manner or MODE memory data is to be displayed or accepted by ZBUG, the user employs the "M" command (see Section 4.8). This command sets the mode in which memory data is to be displayed/accepted, until overridden (see Sections 4.8 and 5.2) temporarily or the "M" command is used again to change the mode to another "default".

The modes that may be specified by the "M" command are byte, word (two byte), three byte, double word (four byte), and instruction. With all but instruction mode, data displayed and accepted is numeric, while in instruction mode, it is in CDL Z80 instruction mnemonics (refer to the CDL "Macro I/II Assembler User's Manual) with numeric operands as applicable.

Note that MODE and the "M" command are associated only with memory data. As registers and flags are of a fixed length, they may be considered as having a fixed mode - byte or word (two byte) for registers, and bit for flags.

2.1.2 Data RADIX

Numeric data, regardless of the mode in which it is considered, must have an understood RADIX in order to be interpreted properly. ZBUG provides the means to specify the radix for three different types of data - contents of memory "cells" or registers displayed by ZBUG, any numeric data typed by the user, and addresses of memory "cells" as displayed by ZBUG. The radix of each type of data may be separately specified by the user via the "R" command (see Section 4.12) and may be ASCII (with the exception of the address type), binary, decimal, hexadecimal, octal, split octal (3 digits per byte), and relative (signed) decimal.

The radixes specified by the "R" command determine the "default" used by ZBUG regarding the specific data types, and remain in effect until overridden (see Sections 4.12 and 5.2) temporarily or reset by another use of the "R" command.

Debug I displays all data in hexadecimal, and therefore does not provide the "R" command. Data type-in may be in hexadecimal or ASCII (see Section 5.1).

2.1.2.1 Data Display

All numeric memory data (such as the contents of a "cell" in modes 1 (byte) through 4 (four byte), an instruction operand in mode instruction, or the contents of a register) are displayed by ZBUG in the radix set by the "RD" variation of the "R" command (see Section 4.12). This radix may be temporarily overridden in certain cases by a special application of the "R" command (see Sections 4.3, 4.12, 4.15, and 5.2), but always reverts to the radix last set by the "RD" command.

2.1.2.2 Data Type-in

All numeric data typed by the user is assumed to use the default radix set by the "RT" variation of the "R" command. This radix may be temporarily overridden with the use of the radix operator and/or modifier (see Sections 5.1, 5.2, and the CDL "Macro I/II Assembler User's Manual), but always reverts to the radix last set by the "RT" command.

2.1.2.3 Address Display

Each address displayed by ZBUG, whether the address of a "cell", an instruction, or the address operand of an instruction, is displayed in the radix set by the "RA" variation of the "R" command. This radix may only be changed by the application of the "RA" command.

Addresses displayed by ZBUG are in one of two basic froms: absolute (the address displayed represents an actual physical address), and relative (the address displayed is a displacement relative to some absolute address).

ZBUG provides two pairs of "relocation registers" (see Sections 4.15 and 5.2) which are used by ZBUG to calculate

relative addresses. The addresses contained in these registers (the RR - 'RR and DR - 'DR pairs) may be set to the beginning and ending addresses of any area of memory. If an address value lies between the beginning and ending addresses found in either pair of relocation registers, ZBUG will subtract the beginning address from the address to be displayed, forming a relative offset. This offset is then displayed, followed by a single quote (') if the offset is relative to the RR - 'RR pair, or a double quote (") if relative to the DR - 'DR pair.

If an address cannot be relocated to either base pair, or if ZBUG is commanded not to display relative addresses (via the "RA" command), the address is displayed as an absolute value.

Debug I will always display addresses as absolute, and does not feature the RR or DR relocation register pairs.

If ZBUG finds that an address to be displayed lies between its own bounds, the address will be displayed as an offset relative to itself, followed by a pound sign (#).

2.2 Examining and Modifying Data

With the ability to specify the mode and radix data is to be displayed and accepted, the standard facilities of data examination and modification are greatly enhanced. ZBUG provides several commands facilitating manipulation of memory and register data.

2.2.1 Memory Data

ZBUG has six different commands for the manipulation of memory data - two for displaying data, two for replacing data, one for examination/modification of data, and one for searching for data.

2.2.1.1 Display Data

The "D" command (see Section 4.2) is used to display sequential "cells" (or instructions) in the current mode (as set by the "M" command) and radix (the "RD" command) along with their addresses (in the radix set by the "RA" command).

Debug I displays both the data and their addresses in hexadecimal only.

The "L" command (see Section 4.7) is used to display ASCII printable data only, along with the associated address.

2.2.1.2 Replace Data

Using the "F" command (see Section 4.4), an area of memory may be filled with a numeric constant in the mode set by the "M" command (except for instruction mode - in this case, byte mode is used).

The "P" command (see Section 4.10) makes it easy to enter an ASCII string anywhere into memory.

The "O" command is used to load a CP/M format "COM" or "HEX" file (refer to Digital Research "An Introduction to CP/M Features and Facilities") or a CDL TPM "HEX" or "REL" file (refer to CDL "Macro I/II Assembler User's Manual) for debugging purposes.

The "Z" command is used to reproduce the effects of entering a command string at the CP/M command level (see Section 4.17, and the Digital Research "CP/M Interface Guide"). With the aid of this command, the CP/M fcb's TFCB and TFCB+16, and the buffer TBUFF are set (or cleared) as defined in the "CP/M Interface Guide". This command is not avaiable in Debug I.

2.2.1.3 Examine/Modify Data

One of the most powerful commands in ZBUG's repertoire is the "E" command (see Section 4.3), which provides the means to examine and optionally modify memory. The "cell" is displayed, and modifying data accepted, in the current mode and data display radix, either of which may be overridden (see Sections 4.3, 5.1, and 5.2) temporarily. With this command, the "cell" currently under examination or opened may optionally be changed (merely by typing a replacement value or instruction), re-examined in a different mode/radix, or closed. Closing a "cell" can be followed the opening of the next sequential one, the last sequential one, a "called" one, a "returned from" one, or none - all with one keystroke (see Sections 4.3 and 5.2). With this command, a single memory cell or many may be examined and changed, a single instruction or a number of instructions examined and/or entered.

Debug I will display both the data and addresses in hexadecimal always. A cell may be re-examined in a different mode, but not a different radix.

2.2.1.4 Search for Data

With the "Y" command (see Section 4.16), a string of "cells" or instructions (depending on the mode set by the "M" command) may be searched for in memory. ZBUG displays the addresses of each occurrence of the string found.

2.2.2 Registers and Flags

The "X" command (see Section 4.15) is provided to examine and optionally modify the contents of a machine register or flag (see Section 4.15, and the Zilog "Z80-CPU Technical Manual") or a ZBUG psuedo register (see Sections 4.15, 5.1, and 5.2). The contents of a register are displayed in the radix set by the "RD" command, which can be overridden (see Sections 4.15 and 5.2) temporarily by special application of the "R" command. Flag values are always displayed as a 0 or 1.

Debug I displays register values in hexadecimal, and the special application of the "R" command does not apply.

2.3 Program Execution and Breakpoints

ZBUG implements the standard "goto with breakpoints" in the form of the "G" command (see Section 4.5).

2.3.1 Executing a Program

With the "G" command, complete transfer of control from ZBUG to the address specified is made - i.e., the machine is no longer under ZBUG's supervision. Until a breakpoint is reached, the user's program has complete control at normal execution speed.

2.3.2 Breakpoints During Execution

The use of the "G" command includes the setting of up to seven individual software breakpoints by ZBUG before tranfer of control to the user program is effected. These breakpoints take the form of a "restart 6" instruction (RST 6 - or 0F8 hex), and ZBUG assumes the user program does not use this instruction and does not modify locations 030-032 hex in memory.

2.4 Tracing and Traps

Perhaps the most useful and powerful features of ZBUG are its tracing and trap capabilities.

2.4.1 Tracing a Program

Utilizing the "T" command (see Section 4.14), it is possible to execute a program while under ZBUG's full supervision. The user may specify the number of instructions to be traced. Unless interrupted (by a trap condition, invalid instruction, or user intervention), ZBUG will execute the program, simulating actual execution at a speed of 250-2500 times slower (depending on various conditions).

While tracing, ZBUG may be instructed to display the instructions executing, and the values of certain machine registers modified by those instructions.

Debug I will always display the executing instructions and modified machine registers.

2.4.2 Traps During Tracing

The "S" command (see Section 4.13), in combination with the trace capability of ZBUG, provides the ability to trace a program until any one of four arbitrary conditions occurs. With the S command, up to four boolean expressions (see Sections 5.1 and 5.2) may be saved. Each of the saved expressions are evaluated after each instruction traced. If any one of the evaluated expressions returns a non-zero value, ZBUG halts tracing, notifying the user of the trap.

As traps are determined and controlled by arbitrary boolean expressions, traps may be set to monitor register/flag/memory value compared to a constant or other register/flag/memory value(s). A register or flag may be compared to its own value prior to the last instruction execution, in order to monitor a change in value. It is possible to trap when a register/flag/memory value reaches a certain value (or range of values), or when a specific instruction is about to be executed.

The potential of this feature is limited only by the user's understanding and use of the conditional expressions that may be devised.

Debug I does not provide the trap features found in Debug II, and tracing interruption is left to user intervention or invalid instructions.

Section 3

Starting Out

This section is provided to introduce the user to ZBUG. Included in the discussion following is a description of the environment under which ZBUG runs, how to execute ZBUG in order to debug an existing program, and basic debugging operations using a few simple commands.

3.1 ZBUG's Operating Environment

ZBUG is designed to operate under the Digital Research CP/M Operating System or the Computer Design Labs' TPM Operating System. The files DEBUGII.COM and ZBUG.REL must be present on the currently logged-on drive (or drive A) in order to run ZBUG.

The program DEBUGII.COM is initiated at the CP/M (TPM) command level (refer to the Digital Research "An Introduction to CP/M Features and Facilities"), which loads ZBUG from the ZBUG.REL file. ZBUG is loaded, following CP/M (TPM) convention, as high in memory as allowed by the operating system, leaving any low memory free to contain the program to be debugged.

All ZBUG disk and console I/O is performed using CP/M (TPM) system functions (refer to the Digital Research "CP/M Interface Guide").

ZBUG.REL currently occupies approximately 13.25 K bytes, including all data areas.

ZBUG.IMG operates in the same manner as ZBUG.REL, with the files DEBUGI.COM and ZBUG.IMG replacing DEBUGII.COM and ZBUG.REL.

ZBUG.IMG occupies approximately 9.25 K bytes.

3.2 Executing ZBUG

There are two methods of invoking ZBUG....

While at the CP/M (TPM) command level, the command:

A> DEBUGII <cr>

will cause the file DEBUGII.COM to be loaded and executed. DEBUGII will display ZBUG's current size (in bytes) and load address (which indicates the size of free memory), in hex. DEBUGII will then load the ZBUG.REL file, relocating it to the highest available memory address. If ZBUG.REL is not located on the currently logged-in drive, DEBUGII will look for it on drive A. DEBUGII will finally pass control to ZBUG, which will display a sign-on message and prompt, after which the user may begin entering commands to ZBUG.

The alternate method of executing ZBUG is as follows. Again at the CP/M (TPM) command level, the command:

A> DEBUGII <name> <cr>

will cause DEBUGII to load the file <name>.COM at location 100 hex (after signaling a successful load of ZBUG), and display the size (in bytes), the load address, and the end address, before passing control to ZBUG. This action is the same as executing ZBUG in the previous manner, and immediately entering the following command to ZBUG:

* o <name> <cr>

Please note that in the examples above, <name> is a CP/M (TPM) filename (i.e., [device :] name [. extension]). With the ZBUG "O" command, a file with the extension "COM", "HEX", or "REL" may be loaded, as described in Section 4.9. If DEBUG loads the file, however, only files with the "COM" extension are loaded. DEBUG ignores any extension entered at the CP/M (TPM) command level, changing it to "COM".

If DEBUG encounters an error while loading either ZBUG or the optionally specified "COM" file, a short error message will be displayed, and control returned to the operating system.

3.3 A Sample Session

This following is a sample workout with ZBUG. A few basic commands will be explored, with the idea of presenting ZBUG's usefulness without getting bogged down with the more involved features.

For this session, assume that the following is a listing of an assembled program which exists on the logged-in drive as the "COM" file PRINT.COM:

```
                        ;
                        ;       A program to list the 10 bytes
                        ;       starting at location 80 hex on the
                        ;       console in hex
                        ;
                                .pabs
                                .phex
0100                    .loc    ^H100
0100  21 0080   print:  lxi     h,^H80  ;start at ^H80
0103  060A              mvi     b,10    ;   and list 10 bytes
0105  7E        loop:   mov     a,m     ;get a byte
0106  CD 010F           call    hex     ;   and list in hex
0109  23                inx     h       ;bump to next byte
010A  10F9              djnz    loop    ;   and loop if more
010C  C3 0000           jmp     0       ;leave when done
010F  F5        hex:    push    psw     ;save byte
0110  1F                rar             ;rotate
0111  1F                rar             ;   to get
0112  1F                rar             ;      high
0113  1F                rar             ;         nibble
0114  CD 011F           call    nibble  ;print high nibble
0117  F1                pop     psw     ;restore byte and
0118  CD 011F           call    nibble  ;   print low nibble
011B  3E20              mvi     a,' '   ;print a space
011D  1804              jmpr    output  ;   and return
011F  E60F      nibble: ani     ^H0F    ;mask out nibble
0121  C630              adi     '0'     ;   and add ASCII zero
0123  C5        output: push    b       ;save <BC
0124  E5                push    h       ;   and <HL
0125  5F                mov     e,a     ;CP/M convention
0126  0E02              mvi     c,2     ;   to print character
0128  CD 0005           call    5       ;      on console
012B  E1                pop     h       ;restore <HL
012C  C1                pop     b       ;   and <BC
012D  C9                ret             ;return
0100                    .end    print
```

The following is a verbatim listing of a terminal session. The user's input is in lower case alphabetics, the computer's response in uppercase. Comments of the related user and computer actions are enclosed in curly brackets ("{ ... }"). The constructions "<cr>" and "<lf>" in the session are user-typed carriage return and line feed, respectively.

{ Starting at the CP/M (TPM) command level, we'll try executing PRINT, to see what happens.... }

```
A>print <cr>
00 00 42 55 47 2= 34 20 20 44
A>
```

{ Obviously, something appears to be wrong - where did the "2=" come from ? }

```
A>DEBUGII print <cr>

ZBUG's length - ^H0351F and
load address  - ^H03AE1

ZBUG is loaded....
loading COM file....
^H00200 bytes loaded -- (from ^H00100 to ^H002FF)
```

CDL Z80 CP/M DYNAMIC DEBUGGER VERSION 1.00 <NOTE: Signon for your version may not be exactly as the above line.>

```
*
```

{ First, let's take a look at location 80 hex to see the 10 bytes we're supposed to be listing.... }

```
* e 80 <cr>
0080:   10      <lf>
0081:   D8      <lf>
0082:   C3      <lf>
0083:   2A      <lf>
0084:   05      <lf>
0085:   CD      <lf>
0086:   41      <lf>
0087:   00      <lf>
0088:   CA      <lf>
0089:   2A      <lf>
008A:   05      <cr>
```

{ Okay, now let's execute it to see the results, but with a breakpoint at the jump back to CP/M }

```
* g 100, 10c <cr>
10 =8 <3 2: 05 <= 41 00 <: 2: *** BREAK (0) --> 010C: JMP
0000
```

{ Yes, folks - something is definately amiss. For each hex digit greater than 9 (as near as we can tell so far), we have a garbage character - ":" for A, "<" for C, and "=" for D. Let's

start it up again, stopping after getting the second byte (the first with a problem) from memory. }

```
* g 100, 10f <cr>
*** BREAK (0) --> 010F: PUSH    PSW
```

{ That was the first, loop for the second.... }

```
* g, 10f <cr>
10 *** BREAK (0) --> 010F:      PUSH    PSW
```

{ Okay, now look at that byte, in the A register }

```
* xa <cr>
D8      <cr>
```

{ Yes, it's the right byte, alright. Now go until we've isolated the top nibble - the one with the problems... }

```
* g, 11f <cr>
*** BREAK (0) --> 011F: ANI     0F
```

{ ... and look again. }

```
* xa <cr>
0D      <cr>
```

{ Right - the nibble in set up properly. So, we'll go ahead and mask it and add the '0'... }

```
* g, 123 <cr>
*** BREAK (0) --> 0123: PUSH    B
```

{ ... and look at the result - in ASCII since we're ready to print out what's left. Note that in Debug I we can't do this - we'd have to stay in hex }

```
* xa <cr>
3D      ;ra <cr>
'='     <cr>
```

{ There's the problem. We now remember that a 0D hex added to a '0' (30 hex) is not the 'D' (44 hex) we wanted. All we have to do is add a check to see if the character we're going to print is greater than a '9' (39 hex), and add enough to get the proper character. We'll need to know what that extra value is so.... }

```
* c 'D'-3d <cr>
07
```

{ We see that to get a 'D' instead of a '=', we have to add a 7. Now, we'll modify the program to include the check and add. First, we should take advantage of ZBUG's assembler/disassembler capabilities, and get into instruction mode. }

```
* mi <cr>
MODE: INSTRUCTION
```

```
    { Now, let's change that "nibble" routine.... }

    * e 11f <cr>
    011F:   ANI     0F      <lf>
    0121:   ADI     30      <lf>

    { We'll  put the  check right  on top  of where  the "output"
routine is, making it garbage.... }

    0123:   PUSH    B       cpi     '9'+1 <lf>
    0125:   MOV     E,A     jrc     .+4 <lf>
    0127:   STAX    B       adi     7 <lf>

    { Now we must relocate the "output" routine.... }

    0129:   DCR     B       push    b <lf>
    012A:   NOP             push    h <lf>
    012B:   POP     H       mov     e,a <lf>
    012C:   POP     B       mvi     c,2 <lf>
    012E:   JRNZ    0150    call    5 <lf>
    0131:   MOV     B,H     pop     h <lf>
    0132:   MOV     D,H     pop     b <lf>
    0133:   JRNZ    018B    ret <cr>
    *

    { Okay, now to fix up the relative jump to "output"... }

    * e 11d <cr>
    011D:   JMPR    0123    jmpr    129 <cr>

    { ... and  try it out,  stopping again before  running off to
CP/M. }

    * g 100, 10c <cr>
    10 D8 C3 2A 05 CD 41 00 CA 2A *** BREAK (0) --> 010C: JMP
0000
    *

    { Fine, all's well.  We'll leave ZBUG... }

    * q <cr>

    A>

    { ... and save it. }

    A>save 2 print.com <cr>
    A>

    { One last time.... }

    A>print
    00 00 42 55 47 2D 34 20 20 44

    { All fixed ! }
```

Section 4

The Commands - A Detailed Description

The following is a listing of the commands ZBUG provides, with a detailed description of the use and operation of each. This section is intended for use as a detailed reference guide to ZBUG.

The format of each command is as follows:

Command character - Command name
Command format
Description
Example

The following symbols are used to describe the format to the various ZBUG commands:

<CR> represents a carriage return,
<LF> a line feed,
<BS> a backspace (control-H),
<ESC> an escape (or altmode)
<FF> a form feed (control-L)

[...] means contents are optional,
{ ... } means contents are mandatory,
... ¦ ... means "or" - i.e., a choice can be made
...]+ -or- ...}+ means one or more
...]* -or- ...}* means zero or more

Please note that ZBUG is only blank (' ') sensitive where expressly stated. Normally, blanks may be used freely during type-in to facilitate easier reading. Note also that more than one command may be typed on the same line, separated by semicolons (';') and terminated by a <CR>. ZBUG will attempt to execute each command in order of appearance, unless a) an error occurs, or b) an "E", "X", or "Y" command has been executed, after which any remaining commands will be ignored.

Unless explicitly defined otherwise, all constructions of the form "< ... >" in the command descriptions (such as <expression>, <address>, <count>, etc.) are properly formed ZBUG expressions (see Section 5.1).

In each of the command examples, the ZBUG prompt ("*") is followed by the user input (in lower case alphabetics), and then by any ZBUG response. Unless otherwise stated, the mode is byte and the radixes (address display, data display, and default type-in) are hexadecimal.

4.1 C - Calculate

C <expression>

Calculate the value of <expression> and display in the current mode and data display radix.

The <expression> is evaluated and its value displayed in the current mode and data display radix. If the <expression> is omitted, no value is displayed.

Debug I will display the resulting value in hexadecimal.

```
{ calculate 1+2 }

* c 1+2 <cr>
Ø3
*

{ calculate 3*5 }

* c 3*5 <cr>
ØF
*
```

4.2 D - Display

D [<address>] [, <count>]

Display memory in the current mode and data display radix.

Starting at <address>, display <count> sequential cell addresses (in the current address display radix) and cell contents (in the current mode and data display radix). If <address> is omitted, the default is Ø. If <count> is omitted, the default is 1. ZBUG formats the display, placing an address and one, two, four, or eight values per line (depending on mode and data display radix).

Debug I will display both the data and addresses in hexadecimal.

```
{ display cell at Ø }

* d <cr>
ØØØØ:   F8
*

{ display 5 cells starting at 1ØØ }

* d 1ØØ, 5 <cr>
Ø1ØØ:   37 23 4Ø F7 EA
*

{ set instruction mode (see Section 4.8), then
 display 4 cells (instructions) starting at 2ØØ }
```

```
* mi <cr>
MODE:  INSTRUCTION
* d 200, 4 <cr>
0200:   ANI     04
0202:   JRNZ    2010
0204:   POP     PSW
0205:   RET
*
```

4.3 E - Examine

E [<address>]

Open cell at <address> for examination and modification.

Open the cell at <address>, displaying <address> in the current address radix, and the contents of the cell in the current mode and data display radix. Accept from the user an optional replacement value - assumed to be in the current mode - followed by a valid closing character. Note that if the current mode is instruction, the user may type an instruction (mnemonic/operand(s) sequence). When accepting an instruction as a replacement value, ZBUG is blank sensitive, as a blank or tab MUST separate a mnemonic from any operand(s) required by the particular instruction.

Debug I will display both the cell data and address in hexadecimal.

After the replacement value has been typed, or instead of it, the user must close the location to continue on. To close the location, he/she may do one of the following:

type a <CR> to close and exit the E command,

type a <LF> to close and open the next sequential cell,

type a comma (',') to close and open the next sequential cell on the same line (not valid in instruction mode, as a ',' separates instruction operands),

type a <BS> to close and open the last sequential cell (in instruction mode, the mode is first changed to byte (see * Note)),

type a <ESC> to close and open the cell pointed to by the last value (or address, if instruction mode) typed or displayed, pushing the "return" address of the next sequential cell on the "call" stack,

type a <FF> to close, popping the "return" address from the "call" stack and opening that cell,

type a semicolon (";") followed optionally by "Mn" (see Section 4.8) and/or "Rn" (see Section 4.12), followed by a <CR>, to optionally change the current mode and/or data display radix temporarily (see * Note) and reopen the cell. (Note that the

"Rn" option is not available in Debug I)

Note that, for each newly opened cell, ZBUG goes to a new line, displays the cell's address (in the current address display radix) and the cell's contents (in the current mode and data display radix).

* Note: The changing of the mode and/or data display radix as indicated above changes same only until a) it is changed again in the same manner or b) a cell is closed with a <CR>, exiting the E command and restoring the mode/radix of before. Note also that the radix cannot be changed in Debug I.

Note that the special value "." ("here") always contains the address of the cell currently open, and on exit from the E command, contains the address of the last cell opened. For more information regarding ".", refer to Section 5.1.

```
{ examine cell at 100 }

* e 100 <cr>
0100:   C3      <cr>
*

{ examine and modify cells... }

* e 100 <cr>
0100:   C3      <lf>
0101:   00      05 <lf>
0102:   34      <cr>
*

{ set instruction mode, and try a few things... }

* mi <cr>
MODE:  INSTRUCTION
* e 100 <cr>
0100:   JMP     3405    <lf>
0103:   CALL    2156    <lf>
0106:   JRNZ    0113    <esc>
0113:   CPI     41      ;ra <cr>
0113:   CPI     'A'     <lf>
0115:   JRZ     0134    <ff>
0108:   ANI     '^A'    ;rh <cr>
0108:   ANI     01      ANI     02 ; <cr>
0108:   ANI     02      <cr>
*
```

Please refer to Section 5.2 for more discussion.

4.4 F - Fill

F [<address>] , [<count>] , <value>

Fill cells with constant.

Starting with <address>, fill <count> sequential cells in the current mode (if instruction mode, assume byte) with the value <value>. If <address> is omitted, default to Ø. If <count> is omitted, default to 1.

```
{ fill starting at 100 with 4 '!''s }

* f 100, 4, '!' <cr>
*

{ put a Ø in cell 100 }

* f 100, , Ø <cr>
*
```

4.5 G - Goto

G [[<start>] [, <break>]*]

Goto <start> with breakpoints at <break>.

Clear any previously set breakpoints. For each <break> address (ZBUG provides up to seven) indicated, set a software breakpoint. Begin execution (complete transfer of control) at the address <start>. If a breakpoint is encountered, interrupt execution and notify the user of the break, indicating which trap occurred (Ø - 7) and displaying the address and associated instruction. If no breakpoints were specified, do not set any. If <start> was omitted, default to the address in the program counter (register <PC).

ZBUG utilizes a "restart 6" (RST 6 -or- ØF8 hex) instruction for software breakpoints, which require that ZBUG place a jump instruction at locations ØØ3Ø to ØØ32 hex for proper operation of breakpoints. This implies that the user program must NOT use this instruction or modify these locations, or undefined actions may result.

When a break has been encountered and the user notified, ZBUG places the address of the break (kept in the program counter - register <PC) in the special value ".". For more information, refer to Section 5.1.

```
{ start execution at 100 }

* g 100 <cr>

{ start execution at the address in the program counter, with
a breakpoint at 340 }

* g, 340 <cr>
*** BREAK (Ø) --> Ø340: CALL    2351
*
```

4.6 I - Instruction Interpret

I <instruction>

Interpretively execute the <instruction>.

After "assembling" the <instruction>, interpretively execute it.

This command provides the capability of executing an arbitrary instruction, in order to effect the actions determined by the particular instruction. Unless the instruction performs a transfer of control (i.e., jumps, calls, returns, restarts, or indirect register jumps), the program counter will not be changed.

ZBUG effectively traces the one instruction and returns to the user.

Note that if the <instruction> is a call or restart instruction, the associated return address pushed on the stack will be within ZBUG which, when transfered to by a matching return (or any other means), will cause ZBUG to notify a breakpoint, and return to the user.

This command is not available in Debug I.

{ execute an increment register A instruction checking the contents before and after }

```
* x a <cr>
Ø3      <cr>
* i inr a <cr>
* x a <cr>
Ø4      <cr>
*
```

{ execute a push register pair H&L instruction and check the stack pointer }

```
* x sp <cr>
Ø546    <cr>
* i push h <cr>
* x sp <cr>
Ø544    <cr>
*
```

4.7 L - List ASCII

L [<address>] [, <count>]

Starting at <address>, list <count> ASCII characters.

Starting at <address>, display <count> sequential printable ASCII charaters (if nonprinting, print a "."), preceding each group of up to 32 characters with the address (in the current address display radix) associated. If <address> is omitted,

default to Ø. If <count> is omitted, default to 1.

ZBUG formats the display, so that for every line displayed contains one address and up to 32 ASCII characters.

Debug I will display the address in hexadecimal.

```
{ list the 23 characters starting at 100 }

* l 100, 23. <cr>
Ø100:   ...A.+;@....%ØAa*:....j
*

{ list the character at Ø }

* l <cr>
ØØØØ:   .
*
```

4.8 M - Mode

M [<modifier>]

where <modifier> is { B | W | I | 1 | 2 | 3 | 4 }

Set current mode.

Set the current mode to byte ("B" or "1"), word or double byte ("W" or "2"), triple byte ("3"), double word or four byte ("4"), or instruction ("I"), and display a message reflecting the change. If the mode modifier is omitted, display the current setting.

A special application of this command is available during the execution of the "E" (examine) command. In place of the replacement value accepted by ZBUG (or immediately following it), the user may type: ";Mn", where "n" is one of the modifiers described above. This changes the current mode for the remainder of the execution of the command, unless changed by another application of this feature.

When ZBUG is first executed, the mode is set to byte.

```
{ change mode to instruction }

* m i <cr>
MODE:  INSTRUCTION
*

{ change mode to word (two byte) }

* m 2 <cr>
MODE:  WORD
*

{ display the current mode setting }
```

```
* m <cr>
MODE:  BYTE
*
```

4.9 O - Open File

O <filename> [, <bias>] [, <relocation>]

Open <filename> for debugging.

Load CP/M disk file <filename> into memory, with optional bias of <bias> and relocation (if a CDL ".REL" file) of <relocation>. If <bias> is omitted, default to Ø. If <relocation> is omitted, default to 1ØØ hex.

<filename> is a CP/M filename of the form:

[<drive> :] <name> [. <extension>]

where the extension is "COM" for a binary image file, "HEX" for an Intel-compatible "hex" object file (whether binary or ASCII), and "REL" or a CDL relocatable object file (binary or ASCII). If the extension is omitted or not "COM", "HEX", or "REL", a "COM" type file is assumed.

Files are loaded to the actual physical addresses found as follows:

COM files:	The <bias> (defaulting to Ø if omitted) plus 1ØØ hex.
HEX files:	The <bias> (defaulting to Ø if omitted) plus the load address supplied in the HEX format.
REL files:	The <bias> (defaulting to Ø if omitted) plus the <relocation> (defaulting to 1ØØ hex if omitted).

If the file <filename> cannot be found, or if an error has occurred while reading it, or the file attempts to be loaded within ZBUG's bounds, an error will result, and ZBUG will discontinue loading.

During the loading, ZBUG displays the starting load address and the ending load address. If an error is encountered in a "HEX" or "REL" file's format while loading, the address of the last byte loaded will be displayed.

Note that ZBUG is blank sensitive within a filename, assuming any blank encountered terminates the filename.

{ open COM file TEST, loading it starting at 1ØØ }

```
* o test <cr>
Ø1ØØ:   LOAD ADDR
Ø5FF:   END ADDR
```

```
*

{ open file TEST.REL, loading and relocating it at 200 }

* o test.rel, , 200 <cr>
0200:   LOAD ADDR
0342:   END ADDR
*

{ open COM file FILE from drive A }

* o a:file <cr>
0100:   LOAD ADDR
037F:   END ADDR
*
```

4.10 P - Put String

P [<address>]

Put an ASCII string into memory starting at <address>

After displaying the <address> in the current address display radix, accept ASCII charaters from the user, storing them in sequential memory bytes starting at <address>, until a control-D is typed. If <address> is omitted, default to 0.

Debug I will display the address in hexadecimal.

```
{ put a string at 100 }

* p 100 <cr>
0100:   this is a string <control-D>
*

{ put a string at 0 }

* p <cr>
0000:   example <control-D>
*
```

4.11 Q - Quit

Q

Exit ZBUG, returning to CP/M.

```
{ quit and go to CP/M }

* q <cr>
A>
```

4.12 R - Radix

R [<type> [<modifier>]]

where <type> is { A | D | T }
and <modifier> is { A | B | D | H | O | R | S }

Set current address display, data display, default type-in radix.

Set the current address display (<type> "A"), data display (<type> "D"), or default type-in (<type> "T") radix to either ASCII ("A"), binary ("B"), decimal ("D"), hexadecimal ("H"), octal ("O"), relative or signed decimal ("R"), or split octal ("S"), and display a message reflecting the change. If the radix modifier is omitted, display the current setting for the radix type <type>. If the radix type is omitted, display the current settings of each radix type.

A special application of this command is available during the execution of the E (examine) and X (examine register/flag) commands (see Sections 4.3 and 4.15). In place of the replacement value accepted by ZBUG (or immediately following it), the user may type: ";Rn", where "n" is one of the radix modifiers described above. This changes the data display radix for the remainder of the execution of the command, unless changed by another application of this feature.

Note that the ASCII ("A") radix is not valid for the address display radix type.

An additional flexibility is provided for the address radix application. If the radix is being modified, and the command is followed by an "A" (i.e., "RAHA"), ZBUG recognizes that addresses will be displayed as absolute values, and not relocated (see Sections 2.1.2.3, 4.15, and 5.2).

When ZBUG is first executed, each of the radix types are set to hexadecimal, and relative addresses are permitted (i.e., the "A" option for the address radix is not in effect).

This command, and all its variations, are not available in UZBUG.

```
{ change data display radix to hexadecimal }

* r dh <cr>
DATA DISPLAY RADIX : HEXADECIMAL
*

{ display the current address display radix setting }

* r a <cr>
ADDRESS DISPLAY RADIX : BINARY
*

{ set the address radix to hexadecimal, specifying absolute
addresses only }
```

```
* r aha <cr>
ADDRESS DISPLAY RADIX : (ABSOLUTE) HEXADECIMAL
*

{ display the current settings of each of the radix types }

* r <cr>
ADDRESS DISPLAY RADIX : BINARY
DATA DISPLAY RADIX : HEXADECIMAL
DEFAULT TYPE-IN RADIX : DECIMAL
*
```

4.13 S - Set Trap/Conditional-Display

S [D] [*] [<id> [, <expression>]]

where <id> is { Ø | 1 | 2 | 3 }

Set trap/conditional display.

This command performs several different functions, depending on the options used. Each is described separately below.

S [<id> [, <expression>]]

Set trap <id> to boolean expression <expression>. If the <expression> is omitted, display the expression set currently for trap <id>. If <id> is omitted, display the expressions currently set for each trap.

S* [<id>]

Reset trap <id>, removing its currently set expression, and effectively clearing the trap. If <id> is omitted, display the id's of each cleared trap.

SD [<id> [, <expression>]]

Set conditional-display <id> to boolean expression <expression>. If the <expression> is omitted, display the expression saved currently for conditional-display <id>. If <id> is omitted, display the expressions currently set for each conditional-display.

SD* [<id>]

Reset conditional-display <id>, removing its currently set expression, and effectively clearing the conditional-display. If <id> is omitted, display the id's of each cleared conditional-display.

For each trap or conditional-display set, ZBUG saves the <expression> specified, after surrounding it with parentheses and preceding the resulting expression with a unary radix change operator for the current default type-in radix. This is to insure the user that the expression will be evaluated during tracing in the default radix active when the expression was

originally entered, regardless of any subsequent later changes with the R command. Each expression entered, whether trap or conditional-display, must be no more than 59 characters long, including any spaces or tabs contained within.

During tracing (see Section 4.14), each expression saved is evaluated after every instruction traced. The effects of and uses of both types of expression (trap and conditional-display) are described further in Section 4.14 and 5.2.

This cammand, and all its variations, is not available in Debug I.

```
{ set trap Ø to fire when register A changes }

* sØ,<a ?ne <!a <cr>
*

{ set trap 3 to fire when the next instruction to be traced is a pop register pair D&E }

* s3, !. ?eq [pop d] <cr>
*

{ display all currently set traps }

* s <cr>
(Ø)     ^H(<A ?NE <!A)
(3)     ^H( !. ?EQ [POP D])
*

{ display all currently cleared trap id's }

* s* <cr>
1 2
*

{ set conditional-display Ø to display if the program counter is between 1ØØØ and 1296 }

* sdØ,(<pc ?ge 1ØØØ) & (<pc ?le 1296) <cr>
*

i{ display currently set conditional-displays }

* sd <cr>
(Ø)     ^H((<PC ?GE 1ØØØ) & (<PC ?LE 1296))
*

{ display currently cleared conditional-displays }

* sd <cr>
1 2 3
*
```

4.13.1 SW - Set Wait

SW [<count>]

Set a delay time of <count> * 10 msec (at 2 MHZ) for tracing.

Set tracing delay time of <count> (default type-in radix is alway decimal for this command) centi-seconds. If <count> is omitted, display the current setting in decimal centi-seconds.

During tracing, ZBUG will wait for the count set by this command after displaying an instruction and before executing it. This gives the user time to see what is about to be executed, and interrupt ZBUG before anything happens if desired. The count is considered by ZBUG to be between 0 and 255 centi-seconds, giving the user up to just over 2.5 seconds to make the decision to interrupt, or enough time to follow the trace with a listing.

```
{ set delay to maximum }

* sw 255 <cr>
*

{ display setting }

* sw <cr>
255
*
```

4.14 T - Trace

T [<address>] [, <count>] [, { O | C }]

Trace <count> instructions starting at <address>.

Starting at <address>, trace up to <count> instructions. If <count> is omitted, default to 1. If <address> is omitted, start at the address in the program counter.

If the "O" or "C" options are omitted, the automatic display is in effect. This implies that before the execution of each instruction, ZBUG will display the address of the next instruction to be executed (in the current address display radix) and the instruction (with any operands in the appropriate current address or data display radix). ZBUG will not go to a new line (the cursor will remain on the same line as the instruction) until the instruction displayed is executed.

After the instruction has been executed, display the contents of the SP, IX, IY, AF, BC, DE, and/or HL registers if they were modified by the instruction, and then display the next instruction.

If the "O" option is used, turn off automatic display.

If the "C" option is used, turn off automatic display only when tracing a subroutine (code entered by a call instruction and

left by a matching return instruction) in a deeper level (up to 128 calls deep). With this option, the <count> refers only to instructions that are displayed.

Before executing each instruction, evaluate each currently set conditional-display expression. If a non zero value is found, display the id (of the expression causing it) and the instruction to be executed. If no expression value is non-zero, display without an id if the automatic display is in effect, otherwise do not display.

If the current instruction has been displayed (by the automatic display and/or a conditional-display), wait for the time specified by the SW command before executing it and going to a new line. The pause before new-line gives the user time to interrupt tracing by the use of a control-E.

Before executing each instruction, save the present value of all machine registers and flags as the next "old" values.

After executing each instruction, set the special value "." ("here" - see Section 5.1) equal to the value in the program counter.

After executing each instruction, evaluate each currently set trap expression. If any non-zero values are found, display the related id's and expressions, the next instruction, and stop tracing.

If, during tracing, the user types a control-E, or a halt (HLT) instruction or invalid instruction is encountered, halt tracing at the current location.

If, during tracing, the user types a control-T, display the current instuction being traced.

Note that the <count> specified by the user in this command is treated by ZBUG to be a positive 16-bit value, with a default of 1. If it is desired that ZBUG trace indefinitely, a count of Ø will result in an infinite trace.

Debug I does not provide the "O" or "C" options, traps, conditional display, or the control-T features available in Debug II.

{ trace the next instruction }

```
* t <cr>
        Ø238:   INR     A       - AF (1501)
*
```

{ assuming the trap and conditional-display settings of the S command examples of Section 4.13, begin an "infinite" trace to see what happens }

```
* t 100, Ø <cr>
        Ø100:   ORA     A       - AF (15ØØ)
        Ø101:   JNZ     12ØØ
```

```
    (Ø)     1200:   MVI     B,21      - BC (2100)
    (Ø)     1202:   LXI     H,0000    - HL (0000)
    (Ø)     1205:   CCIR              - AF (0012) - BC (0000) - HL
(2100)
    (Ø)     1207:   JNZ     1450
            1450:   PUSH    D         - SP (050C)
            1451:   MOV     A,B       - AF (0012)
    *** TRAP (Ø) --> ^H(<A ?NE <!A)
    1452:   ORA     A
    *

    { ... and going on from there... }

    * t, Ø <cr>
            1452:   ORA     A         - AF (0044)
            1454:   JRZ     146A
    *** TRAP (3) --> ^H( !. ?EQ [POP D])
    146A:   POP     D
    *

    { now, do it all again, but this time, turn off the automatic
display off }

    * t 100, Ø, o <cr>
    (Ø)     1200:   MVI     B,21    - BC (2100)
    (Ø)     1202:   LXI     H,0000  - HL (0000)
    (Ø)     1205:   CCIR             - AF (0012) - BC (0000) - HL
(2100)
    (Ø)     1207:   JNZ     1450
    *** TRAP (Ø) --> ^H(<A ?NE <!A)
    1452:   ORA     A
    *

    { start a  trace, assuming  trap Ø  is set  to fire  when the
program counter is between 145 and 14E }

    * t 100, Ø
            0100:   MVI     A,01      - AF (0100)
            0102:   LXI     D,0000    - DE (0000)
            0105:   CALL    1230      - SP (05F0)
            1230:   PUSH    H         - SP (05EE)
            1231:   PUSH    B         - SP (05EC)
            1232:   MOV     B,A       - BC (010E)
            1233:   XCHG              - DE (1257) - HL (0000)
            1234:   MVI     M,Ø
            1236:   DJNZ    1234      - BC (000E)
            1238:   POP     B         - SP (05EE) - BC (1537)
            1239:   POP     H         - SP (05F0) - HL (0012)
            123A:   RET               - SP (05F2)
            0108:   JMP     0140
            0140:   LXI     H,0000    - HL (0000)
            0143:   LXI     B,FFFF    - BC (FFFF)
    *** TRAP (Ø) --> ^H((<PC ?GE 145) & (<PC ?LE 14E))
    0146:   XRA     A
    *

    { now, do the same, but with the C option }
```

```
* t 100, 0, c
        0100:   MVI     A,01      - AF (0100)
        0102:   LXI     D,0000    - DE (0000)
        0105:   CALL    1230      - SP (05F2)
        0108:   JMP     0140
        0140:   LXI     H,0000    - HL (0000)
        0143:   LXI     B,FFFF    - BC (FFFF)
*** TRAP (0) --> ^H((<PC ?GE 145) & (<PC ?LE 14E))
0146:   XRA     A
*
```

4.15 X - Examine Register/Flag

X [[<] <register-name> ¦ > <flag-name>]

where <register-name> is:

```
[ ! ] [ ' ]   { AF ¦ BC ¦ DE ¦ HL ¦
                SP ¦ PC ¦ IX ¦ IY ¦
                IR ¦ RD ¦ WR ¦ RR ¦
                DR ¦
                A ¦ F ¦ B ¦ C ¦ D ¦
                E ¦ H ¦ L ¦ I ¦ R ¦
                M }
```

and <flag-name> is:

```
[ ! ] [ ' ] { C ¦ H ¦ N ¦ P ¦ S ¦ V ¦ Z }
```

Open register/flag for examination and modification.

Open the specified register or flag, displaying its contents in the current data display radix (or a 0 or 1 if display a flag). Accept from the user an optional replacement value followed by a valid closing character.

After the replacement value has been typed, or instead of it, the user must close the register or flag, by doing one of the following:

type a <CR> to close the register or flag and exit the X command;

type a semicolon (";") followed optionally by "Rn" (see Section 4.12), followed by a <CR>, to optionally change the current data display radix temporarily (see * Note) and reopen the register (note that this does not work if examining a flag).

* Note: The change of data display radix as indicated above is in effect only until a) it is changed again in the same manner as above or b) the register is closed with a <CR>, exiting the X command and restoring the previous mode.

If no register or flag name is specified, display the contents of all the machine registers, the pseudo registers, the top four word values on the stack, and the instruction pointed to by the program counter.

The register names defined above are further described below:

The 16-bit registers:

AF - The Z80 register pair commonly known as PSW
BC - The Z80 register pair B&C
DE - The Z80 register pair D&E
HL - The Z80 register pair H&L
SP - The Z80 stack pointer
PC - The Z80 program counter
IX - The Z80 index register X
IY - The Z80 index register Y
IR - The Z80 register "pair" made of the combining
- of the interrupt register and the refresh
- register
RD - The ZBUG pseudo register containing the
- address of the last traced memory read
- access
WR - The ZBUG pseudo register containing the
- address of the last traced memory write
- access
RR - The ZBUG pseudo register containing the
- user defined "code" relocation address
DR - The ZBUG pseudo register containing the
- user defined "data" relocation address

The 8-bit reisters:

A - The Z80 register A (accumulator)
F - The Z80 flag "register"
B - The Z80 general register B
C - The Z80 general register C
D - The Z80 general register D
E - The Z80 general register E
H - The Z80 general register H
L - The Z80 general register L
I - The Z80 interrupt register I
R - The Z80 refresh register R
M - The Z80 "register" M (byte pointed
- to by register pair H&L

The flag names described above are further defined below:

C - The Z80 carry flag
H - The Z80 half-carry flag
N - The Z80 add/subtract flag
P - The Z80 parity/overflow flag
S - The Z80 sign flag
V - The Z80 parity/overflow flag
Z - The Z80 zero flag

For a complete description of the Z80 registers and flags, please refer to the Zilog "Z80-CPU Technical Manual".

The optional "!" is used to access the "old" value of a machine register or flag - the value of the register/flag before the last instruction traced. As ZBUG does not save the "old" values of the pseudo registers (RD, WR, RR, or DR), the "!" will

have no affect if used to to refer to them.

The optional "'" used to refer to a register generally means to consider the Z80 auxiliary register of the same name. This refers to the A, F, B, C, D, E, H, L, M, AF, BC, DE, and HL machine registers, and all of the flags. In the case of the other machine registers, the "'" has no affect.

The "'" does have significance with the three ZBUG pseudo registers, and each are described below.

The ZBUG pseudo registers RD and 'RD are set during tracing. Each instruction that accesses memory for a read sets these registers as follows: The RD register is set to the address of the lowest byte accessed by the instruction, and the 'RD register is set to the highest. An instruction that does not access memory for a read will not disturb the contents of either register (the access by the program counter to get the instruction is not considered a read access by ZBUG). For example, if the instruction being traced is a "MOV A,M", both register RD and 'RD will be set to the address contained in the register pair H&L. If the instruction is a "RET", the RD register will be set to equal the address contained in the stack pointer, and the 'RD will be set to that address plus one. If the instruction is an "LDIR", the RD register will be set to the address contained in the register pair H&L, and the 'RD will be set to that address plus the contents of register pair B&C minus one. Finally, if the instruction is an "LHLD 0100", the RD register will be set to 0100, and the 'RD set to 0101.

The ZBUG pseudo registers WR and 'WR act like the RD and 'RD registers, but are set for memory write accesses.

The RD, 'RD, WR, and 'WR registers facilitate the monitoring of memory read and write accesses.

The ZBUG pseudo registers RR, 'RR, DR, and 'DR are registers utilized by the user and ZBUG to facilitate access to memory with "relocatable" addresses. For example, if a user wishes to debug code in a certain sub-set of a program (such as a single module of the program), he sets register RR (or DR) via the X command to the address of the start of the sub-set or module. The 'RR (or 'DR) register is then set to the end of the module. ZBUG will then always display any addresses which lie between these two values as being a relative positive offset from the contents of register RR (or DR). Any addresses lying outside this range will be displayed as absolute. ZBUG signals the difference by following any relocated addresses with a single quote if relative to RR, or a double quote if relative to DR. Absolute address are not flagged in this manner. To any address (or any expression, for that matter) that the user types with a following single or double quote, ZBUG will add the contents of the RR or DR register, resulting in an absolute address.

If the register RR (or DR) is equal to 0 (its default value), ZBUG will not relocate any addresses displayed, and any typed by the user with the "'" signal for relocation will be taken as absolute values (offset plus a base address of 0).

This feature of ZBUG makes debugging a module with a listing showing only relative addresses a simple matter of typing a relocatable address instead of an offset plus a constant. By using the RR pair to refer to the code of a module, and the DR pair for a separate data module, following a listing is easier. For further discussion, refer to Section 5.1.

When ZBUG is first executed, all registers are initialized. The SP, RD, 'RD, WR, and 'WR are set to point to the bottom of ZBUG - the highest memory address that a debugging program may use. The PC is set to 100 hex. The other registers are set to 0.

Debug I does not provide the RR, DR, RD, or WR register pairs, or the re-examine ("Rn") feature found in Debug II.

```
{ examine the A register }

* x a <cr>
00      <cr>
*

{ examine and modify the register pair H&L }

* x hl <cr>
0001    0 <cr>
*

{ examine the B register in various radixes }

* x <b
00       ;ra <cr>
'^@'     ;rd <cr>
  0      ;rr <cr>
   +0    <cr>
*

{ examine and modify the carry flag }


* x >c <cr>
0        (<a + 1) > 8 <cr>
*

{ examine all }

* x <cr>
 AF (000F) BC (0001) DE (1253) HL (0000) FLAGS: .....VNC
 AF'(00F0) BC'(0000) DE'(0000) HL'(0000) FLAGS: SZ.H....
 IX (0000) IY (0000) IP (0000) -   INTERRUPTABLE
 RR (0100) RR'(0000) DR (0000) DR'(0000)
 RD (3900) RD'(3900) WR (3900) WR'(3900)
 SP (0000#) -->          00C3    C370    5734    11F3
 PC (0000') --> MOV      A,M
*
```

4.16 Y - Search

Y [<start>] [, <end>]

Search memory from <start> to <end> for string.

Following this command by up to 32 bytes of data cells in the current mode, separated by semicolons (;) and terminated by a <CR>, search memory within the range <start> to <end>, displaying the addresses (in the current address display radix) of each occurrence. If <end> is omitted, assume 0FFFF hex. If <start> is omitted, assume 0.

Debug I will display the addresses in hexadecimal.

```
{ search for 1,2,3,4 throughout all of memory }

* y <cr>
1; 2; 3; 4 <cr>
100F
3451
A003
FF45
*
```

```
{ assuming instruction mode, look for all calls to location 5
between 100 and 1000 }

* y 100, 1000
call 5 <cr>
0134
0562
*
```

4.17 Z - Zap CP/M fcb's

Z <string>

Set up CP/M input as if <string> were part of command at CP/M command level:

A> <program> <string> <cr>

Set up CP/M's TFCB, TFCB+16, and TBUFF with <string>, as defined by the Digital Research "CP/M Interface Guide". If <string> is omitted, clear TFCB, TFCB+16, and TBUFF as also defined.

Note that ZBUG is sensitive to blanks in parsing filenames from <string> - any encountered are assumed to be terminators.

Debug I does not provide this command.

```
{ set up a filename }

* z file.asm <cr>
*
```

```
    { set up a string }

    * z 3/24/78 10:15:00 <cr>
    *
```

Section 5

Going Beyond the Basics

Although ZBUG can be used after getting to know how to use a few commands, much can be gained by becoming familiar and comfortable with ZBUG's most flexible, and therefore most consequential, feature - the expression.

The majority of this section will be used to discuss the ZBUG expression (5.1). The remainder will be used to dramatize the ZBUG's potential capabilities by giving specific examples of commands utilizing expressions as arguments, along with suggestions and hints to help utilize ZBUG to it fullest.

5.1 The ZBUG Expression

ZBUG expressions follow a relatively small set of recursive rules. To help visualize these rules, the following is the syntax or structure of all expressions in BNF (Backus Naur Form).

```
<exp>    ::=   <sub-exp> [ <c-op> <sub-exp> ]*
<sub-exp> ::=  <term> [ <t-op> <term> ]*
<term>   ::=   <bool> [ <b-op> <bool> ]*
<bool>   ::=   <factor> [ <f-op> <factor> ]*
<factor> ::=   { <con> ¦ <sym> ¦ ( <exp> ) } ['¦"] ¦
               <u-op> <factor>

<con>    ::=   <reg> ¦ <flag> ¦ <num>
<reg>    ::=   < <register-name>
<flag>   ::=   > <flag-name>
<num>    ::=   <string> ¦
               <number> ¦
               <instruction>

<c-op>   ::=
<t-op>   ::=   + ¦ - ¦ ! ¦ ^
<b-op>   ::=   * ¦ / ¦ @ ¦ & ¦ < ¦ >
<f-op>   ::=   ?EQ ¦ ?NE ¦ ?LT ¦ ?LE ¦ ?GT ¦ ?GE
<u-op>   ::=   + ¦ - ¦ # ¦ @ ¦ \ ¦ ^ ¦ !
```

The symbols <sym>, <register-name>, <flag-name>, <string>, <number>, and <instruction> will all be defined in the discussion following.

Debug I provides for a very restricted expression, defined in the following BNF:

```
<exp>    ::=   <factor> [ + ¦ - ] <factor>
<factor> ::=   <con> ¦ <sym> ¦ ( <exp> ) ¦
               [ @ ¦ \ ] <factor>

<con>    ::=   <reg> ¦ <flag> ¦ <num>
<reg>    ::=   < <register-name>
<flag>   ::=   > <flag-name>
<num>    ::=   <string> ¦ <hex>
```

Note that the + and - operators are define in Section 5.1.2, the @ and \ in Section 5.1.5, and <hex> is a hexadecimal number as defined in 5.1.7.1.

5.1.1 <exp> and the "_" Operator

As defined earlier, an <exp> (expression) is:

<sub-exp> [<c-op> <sub-exp>]*

-or-

a subexpression followed by zero or more occurrences of a "_" followed by a subexpression

The "_" operator is defined as the CONCATENATE operator in ZBUG. It is dyadic, meaning that it requires two arguments. It operates by concatenating its two arguments in the following manner: Argument #1 (the left one) is shifted right by the length of argument #2. Argument #2 (the right one) is masked to its length (always an integer number of bytes from 1 to 4), low order bytes being masked first. The two resulting values are then or'ed together, forming one value.

Normally, the default length of an arbitrary argument is 4 bytes, the largest value ZBUG can manipulate. Certain values have an implied length, however - such as a register (one or two bytes), or the result of the @, \, ^, and ! unary operators described later.

The concatenate operator has the lowest precedence of all the operators - unless overridden by parentheses, any concatenate operations will performed last, from left to right.

Note that this operator is not available in Debug I.

Examples....

```
<A _ <BC          ... concatenate the current value of
                  ... the A register with that of the
                  ... B&C register pair
\100 _ @101       ... concatenate the byte at location
                  ... 100 with the word at 101
```

5.1.2 <sub-exp> and the +, -, !, and ^ Operators

As defined earlier, a <sub-exp> (subexpression) is:

<term> [<t-op> <term>]*

-or-

a term followed by zero or more occurrences of a +, -, !, or ^ followed by a term

The +, -, !, and ^ operators are defined as the ADD, SUBTRACT, INCLUSIVE OR, and EXCULSIVE OR operators, respectively. They are each dyadic, requiring two arguments. The operations they perform are two's complement add and subtract, logical inculsive and exclusive or, respectively. Both arguments are considered to be 4 byte values, with no overflow or underflow

indication - the sign bit interpretation is left to the user.

The +, -, !, and ^ operators have the 2nd lowest precedence, and are executed from left to right before any concatenate operators (unless overridden by the use of parentheses, of course).

Note that the ! and ^ operators are not available in Debug I.

Examples....

```
1 + 2           ... add 1 to 2
35 - 28         ... subtract 28 from 35
63 ! 128        ... inclusive or 63 and 128
63 ^ 128        ... exculsive or 63 and 128
```

5.1.3 <term> and the *, /, @, &, <, and > Operators

As previously defined, a <term> is:

<bool> [<b-op> <bool>]*

-or-

a boolean expression followed by zero or more occurrences of a *, /, @, &, <, or > and a boolean expression

The *, /, @, &, <, and > operators are defined as MULTIPLY, DIVIDE, MOD, LOGICAL AND, LEFT SHIFT, and RIGHT SHIFT, respectively. They perform an integer multiply, integer divide, integer modulo, logical and, left logical shift, and right logical shift, respectively. They are dyadic, requiring two arguments, each considered to be 4 byte values, except for the < and > operators, which use only the low order 5 bits of the 2nd argument to determine the number of bits to shift the 1st argument.

The *, /, @, &, <, and > operators have 3rd precedence, being performed from left to right before the +, -, !, or ^ operations (unless overridden by parentheses).

Note that the *, /, @, &, <, and > operators are not available in Debug I.

Examples....

```
4 * 2           ... multiply 4 and 2
4 / 2           ... divide 4 by 2
4 @ 3           ... modulo 4 by 3
1 & 5           ... and 1 and 5
5 < 1           ... shift 5 by 1 bit
6 > 1           ... shift 6 by 1 bit
```

5.1.4 <bool> and the ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE Operators

Earlier, the <bool> was defined as:

<factor> [<f-op> <factor>]*

-or-

a factor followed by zero or more occurrences of ?EQ, ?NE, ?LT, ?LE, ?GT, or ?GE followed by a factor

The ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE operators are the EQUAL, NOT EQUAL, LESS THAN, LESS THAN OR EQUAL, GREATER THEN, and GREATER THAN OR EQUAL operators, respectively. They are dyadic, requiring two arguments, each considered to be a 4 byte value. Each performs an arithmetic compare of the two arguments (signs are significant), and return a zero if the condition is false and a -1 if true.

These operators have 4th precedence - the highest of the dyadic operators, meaning that unless overridden by parentheses, these operations will be the first dyadic ones performed.

Note that the ?EQ, ?NE, ?LT, ?LE, ?GT, and ?GE operators are not available in Debug I.

Examples...

```
<A ?EQ 1          ... does register A equal 1 ?
2 ?LT 3           ... is 2 less than 3 ?
-3 ?GE 78         ... is -3 greater than or equal to 78 ?
```

5.1.5 <factor> and the +, -, #, @, \, ^, and ! Operators

Previously, the <factor> was described as:

{ <con> ¦ <sym> ¦ (<exp>) } [' ¦ "] ¦
<u-op> <factor>

-or-

a constant or a symbol or an expression in parentheses (optionally followed by a ' or "), or a +, -, #, @, \, ^, or ! followed by a factor

This is a recursive definition - describing a factor in terms of an expression or a factor. Later we will discuss constants and symbols and the optional "'", but now to the unary operators, etc.

The +, -, #, @, \, ^, and ! operators are defined as PLUS, MINUS, NOT, WORD INDIRECT, BYTE INDIRECT, EXPLICIT LENGTH or RADIX CHANGE, and INSTRUCTION INDIRECT, respectively. These operators are monadic, requiring only one argument.

The +, -, and # operators consider their arguments to be 4

bytes values, performing a arithmetic plus, arithmetic minus (two's complement), and a logical not (one's complement), respectively.

The @, \, and ! operators use only the low order 2 bytes of their argument, treating the 16-bit value as an address and returning the word, byte, and instruction value pointed to by the address. These three operators also return an implied length of their results (for use by the concatenate operator) - 2 bytes for @, 1 byte for \, and 1-4 bytes for ! depending on the instruction located by the argument.

The ^ operator is further refined by following it with a modifier.

If the ^ is followed by a "1", "2", "3", or "4", it is an explicit length operator, which means that its argument will be considered to be 1, 2, 3, or 4 bytes long for the concatenate operator (the only operator which uses length).

If the ^ is followed by a "B", "D", "H", "O", "R", or "S", it is a radix change operator, which means that its argument will use binary, decimal, hex, octal, relative decimal, or split octal as the default type-in radix instead of that defined by the use of the RT command of ZBUG.

The unary operators have the highest precedence or all the operators (unless overridden by parentheses), and are performed first.

Note that the +, -, #, ^, and ! operators, and the ' and " modifiers are not available in Debug I.

Examples....

```
+1              ... return a plus one (no real
                ... effect)
-1              ... return a negative 1
#1              ... return the one's complement
                ... or "not" of 1
@100            ... get the word pointed to by
                ... 100 (the word at location 100)
\<HL            ... get the byte pointed to by
                ... register pair H&L (actually
                ... another way of saying <M,
                ... or contents of register M)
!<PC            ... return the instruction
                ... pointed to by the program
                ... counter
^1(<HL + <A)    ... calculate the value of
                ... register pair H&L added to
                ... register A and specify a
                ... length of one byte
^D(123 + 32)    ... calculate 123 decimal plus
                ... 32 decimal
--1             ... return a minus minus 1 (or 1)
```

5.1.6 "Symbols"

Although ZBUG does not recognize symbols per se, several permanent symbols are defined by ZBUG to facilitate instruction operand encoding by the user. These "symbols" are register names commonly used in Z80 assembler language, and are as follows:

A	=	7
B	=	0
C	=	1
D	=	2
E	=	3
H	=	4
L	=	5
M	=	6
SP	=	6
PSW	=	6
X	=	4
Y	=	4

Although these "symbols" may be used as a constant value in any ZBUG expression, their usefulness is probably confined to use in instruction operands typed by the user.

The special symbol "." refers normally to the current location. When using the E command, . contains the address of the currently open cell. When a breakpoint is encountered from the use of the G command, . conntains the address of the break. When tracing via the T command, . contains the adress of the next instruction (identical to the <PC).

5.1.7 "Constants"

ZBUG provides for a wide variety of constants, including numbers (in the conventional sense), strings, register and flag contents, and even instructions. Each is described below:

5.1.7.1 Numbers

Numbers in ZBUG are one or more digits, followed optionally by a radix modifier ("B", "D", "H", "O", "R", or "S" representing binary, decimal, hex, octal, relative decimal, and split octal, respectively). If the modifier is not present, the number is assumed to be in the current default type-in radix (unless overridden by the use of the radix change operator "^"). In certain cases, a radix modifier typed by the user might be interpreted by ZBUG to be a digit. For example, if the current default type-in radix is hex, and the user types "10B" intending the number 10 binary (2 decimal), ZBUG will read it as 10B hex. The same is true for the "D" modifier. In order to overcome this misinterpretation, the use of the radix change operator will suffice - i.e., ^B10. As an added aid, ZBUG provides the "." as an additional decimal radix modifier.

If digits found in a number do not correspond to the radix assumed, an error will result. Valid digits are:

```
Binary: Ø,1
Decimal:        Ø,1,2,3,4,5,6,7,8,9
Hex:    Ø,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
        (number must start with Ø-9)
Octal:  Ø,1,2,3,4,5,6,7
Relative Decimal:       same as decimal
Split Octal:    same as octal (however, the number is
        checked to conform to 3 digits per byte,
        with the 3rd being Ø,1,2, or 3)
```

Note that only hexadecimal numbers are acceptable to Debug I.

Examples - assuming default type-in radix of hex....

```
1Ø              ... 1Ø hex
1Ø.             ... ØA hex (1Ø decimal)
345S            ... ØE5 hex (345 split octal)
3450            ... ØE5 hex (345 octal)
```

Improper Numbers....

```
^D12A           ... "A" is not a decimal digit
3456S           ... the "4" is not proper in the
                ... 3rd place of a group of 3
                ... (i.e., 456 is not 1 byte value)
AØ1             ... does not start with Ø-9
```

5.1.7.2 Strings

ZBUG provides the capability of using ASCII character strings as constants. These are a string of characters (not including a <CR>) bracketed by a pair of matched single or double quotes. Strings may be of arbitrary length, but the last four characters are the only ones used.

Examples....

```
"this is a string"      ... only got "ring"
'A'                     ... got a 41 hex value
'AB'                    ... and a 4142 hex
```

Improper Strings....

```
"oops'          ... quotes are not matched
"bad one        ... missing quote
```

5.1.7.3 Registers and Flags

Since the Z8Ø machine registers and flags, and the ZBUG pseudo registers contain values, it stands to reason that the user should be able to access their values in expressions easily. ZBUG provides this useful capability.

The general form to access a register value is:

```
< [ ! ] [ ' ] <register-name>
```

where <register-name> is as described in Section 4.15

The general form to access a flag value is:

> [!] ['] <flag-name>

where <flag-name> is as described in Section 4.15

The optional "!" specifies to ZBUG to return the value of the register or flag before the last instruction trace. ZBUG saves all Z80 machine registers and flags before tracing an instruction to provide access to the previous and current values. This facilitates checking to see if a register/flag value changed during the last instruction traced. For example, the expression ">!C ?NE >C" would return a -1 if the carry flag had changed. The "!" option has no effect if used to access a ZBUG pseudo register, as these registers are not saved during tracing.

The "'" option specifies the auxiliary register set of the Z80, as described in Section 4.15.

The register values have an implicit length associated with them - one or two bytes, depending on the register specified.

Examples....

```
<A              ... return the byte value of
                ... register A
<AF             ... return the word value of
                ... register pair AF (also
                ... known as the PSW)
>Z              ... return the bit value of
                ... the zero flag
```

5.1.7.4 Instructions

As an instruction certainly has a numeric value, ZBUG provides the means to use an arbitrary instruction as a numeric constant.

The form is a "[" followed by the instruction and any operands (separated from the mnemonic by a space (" ") or tab (control-I)) and terminated by a "]". If the "]" is not immediately following the instruction, or if the instruction is not properly formed, an error will result. See the CDL Macro I/II Assembler User's Manual for the correct formats of instructions.

Note that instructions as numeric values are not available in Debug I.

Examples....

```
[MOV A,B]       ... return a 78 hex
[CALL 5]        ... return a 0005CD hex
[MVI B,1]       ... return a 0106 hex
```

Improper Instructions....

```
[MOV A,B)        ... need that "]" !
[INR Z]          ... what's a Z ?
```

5.2 Advanced Ideas

Learning to use ZBUG should be a growing process. After learning and using a few commands, such as those demonstrated in the example of Section 3.3, more advanced features of ZBUG can be incorporated in the user's repetoire of well-used capabilities.

The following is an informal discussion, intended to give the user an idea of how to get the most out of ZBUG, After becoming familiar with ZBUG's sophistication, less time will be spent "fighting with the debugger than the debuggee". Please note that these suggestions are no more than quidelines....

When first entering ZBUG, use the "M" (mode) and/or "R" (radix) command to set up the mode/radix environment you're most comfortable in or will be working in most. For debugging a program with a listing, instruction mode and hex radix are usual. Then, if it becomes convienient later to look at some value in a different mode/radix, the temporary settings available with the "E" (examine) and "X" (examine register) commands will usually be sufficient.

With the "E" command, it is possible to change the mode and radix together.

```
{ this is a demonstration of the mode/radix changing within
the "E" command }

* e 100 <cr>
0100:   01      ;mi rd <cr>
0100:   LXI     B,    0 ;m4 rb <cr>
0100:   00101111000000000000000000000001        ;ro <cr>
0100:   05700000001     ;m2 <cr>
0100:   000001  ;rr <cr>
0100:       +1  <cr>
*

{ radix changing within the "X" command }

* x hl <cr>
1234    ;ra <cr>
'^P4'   ;rs <cr>
022:064 ;rb <cr>
0001001000110100        <cr>
*
```

The more comfortable you are with expressions, and what they're capable of, the less you'll have to type to get the job done, and the more you'll be able to accomplish. For example, remembering that "<BC" means "the value in register pair B&C", whenever you want to see the value pointed to by the address in B&C, you can use "<BC" as the address for an application of the "E" command. This way you save yourself the trouble of examining the contents of the B&C register, and then typing that value. This of course applies for any address, data, count, etc. value expected by any command, for any register or flag value, and for instructions and byte or word memory values. In a sense then, these special values may be thought of - and used instead of -

numbers.

```
    { examine the instruction pointed to by the program counter }

    * e <pc <cr>
    0105:  ANI    0F      <cr>

    { using the "L" command, list the ASCII string, whose
starting address is in the word pointed to by register H&L, and
whose lenth is in the byte following the address word }

    * l @ <hl, \ (<hl + 1) <cr>
    4DA2:   100 PRINT X
    *
```

The "S" command expects a boolean expression in order to set a trap or conditional display. This may actually be ANY expression - ZBUG evaluates it and traps or displays if the result is non-zero. Due to the flexibility of the expressions recognized, the trap/conditional display capability is extremely powerful and versatile.

```
    { set a trap to fire when the next instruction to be executed
is a PCHL and the address in register H&L is 0 }

    * s0,(!. ?eq [pchl]) & (<hl ?eq 0) <cr>
    *

    { set a trap to fire if the last instruction wrote in the
area of 0 to 0FF - note that we'll only use the low byte register
of the write access pair, and we can't be certain that an LDIR or
LDDR instruction wrote here due to those instructions' "wildness"
}

    * s0, (<wr ?ge 0) & (<wr ?le 0FF) <cr>
    *

    { set a trap to fire whenever the carry flag is set (not 0) }

    * s0, >c <cr>
    *

    { set a trap to fire whenever the zero flag changes }

    * s0, >!z ?ne >z <cr>
    *
```

If you're going to be debugging a single module of a large program, for which you've got a listing with only relative addresses, the relocation registers can save you a lot of time and trouble. All that is necessary is to know the absolute address of the start of the module (from a linkage editor load map), and set the RR register (or the DR) to that address via the "X" command. If the ending address (or start address plus length) of the module is known, set the 'RR (or 'DR) to it. From then on, if ZBUG displays an address that is in the module, it will be displayed as a relative offset from the start, followed by a ' (or "). To access an address within the module, you only

have to enter the relative offset, followed by a ' (or "). This alleviates a common frustration of trying to figure out the absolute address. In addition, any addresses outside the module will be displayed as absolute.

Since it is possible to have more than one relocation base (refer to CDL "Macro I/II Assembler User's Manual") for a module, such as a code base and a separate data base, the RR-'RR pair can be used for one (code ?) and the DR-'DR pair for another (data ?).

{ knowing the absolute address, examine a routine to see it with absolute addresses }

```
* e 1fed <cr>
1FED:   PUSH    PSW     <lf>
1FEE:   LXI     B,1     <lf>
1FF1:   RAR             <lf>
1FF2:   JRNC    1FF8    <lf>
1FF4:   LBCD    0455    <lf>
1FF8:   LXI     H,0     <lf>
1FFB:   RAR             <lf>
1FFC:   JRNC    2001    <lf>
1FFE:   LHLD    0459    <lf>
2001:   POP     PSW     <lf>
2002:   RET             <cr>
*
```

{ okay, now set up RR and 'RR }

```
* xrr <cr>
0000    1fed <cr>
* x'rr <cr>
0000    2002+1 <cr>
*
```

{ now look at the routine, remembering that addresses refered to inside the routine will be relative offsets to register RR, and those refered to outside will be absolute }

```
* e 0' <cr>
0000':  PUSH    PSW     <lf>
0001':  LXI     B,1     <lf>
0004':  RAR             <lf>
0005':  JRNC    000B'   <lf>
0007':  LBCD    0455    <lf>
000B':  LXI     H,0     <lf>
000E':  RAR             <lf>
000F':  JRNC    0014'   <lf>
0011':  LHLD    0459    <lf>
0014':  POP     PSW     <lf>
0015':  RET             <cr>
*
```

Remember the current type-in radix that you're operating under - it is easy sometimes to enter a number thinking it is one value, while ZBUG calculates another. For example, the sequence "10B" alway means 10B hex if the current type-in radix is hex,

and not 10 binary with a "B" modifier. The same goes for "10D" - it's not 10 decimal.

To alleviate this confusion, use the radix-change operator.... ^B10 for 10 binary, ^D10 (or 10.) for 10 decimal, etc.. Since the radix-change operator is a unary operator, you can follow it with an expression in parentheses to include more than one number.

ZBUG uses this, in fact, to enclose each trap and conditional display expression entered. This insures the user that the expressions will be evaluated using the type-in radix in effect when the expression was originally entered, in case the user decides to change the defualt later.

Although the use of carriage return and line feed are fairly standard for examining memory (such as Digital Equipment Corporation's DDT), additional flexibility may be gained with the escape ("call") and form feed ("return") methods of closing a currently open cell and moving on. They are especially useful when examining a section of code with a jump or call instruction - facilitating examining the code at the jump or call address with one keystroke, and returning to the instruction following the jump or call with another.

Appendix A

A Quick Reference to the Commands

The following is a quick reference to the ZBUG command set and special characters.

C <expression>	- "Calculate"
D <address>,<count>	- "Display"
E <address>	- "Examine"
F <address>,<count>,<value>	- "Fill"
G <address> [,<break>]*	- "Goto"
I <instruction>	- "Execute"
L <address>,<count>	- "List"
M <mode>	- "Mode"
O <name>,<bias>,<rel>	- "Open"
P <address>	- "Put string"
Q	- "Quit"
R <type><radix>	- "Set radix"
S <id>,<expression>	- "Set trap"
S* <id>	- "Clear trap"
SD <id>,<expression>	- "Set cond. disp."
SD* <id>	- "Clear cond. disp."
SW <time>	- "Set wait"
T <address>,<count>,<op>	- "Trace"
X <reg/flag>	- "Examine reg/flag"
Y <address>,<address>	- "Search"
Z <string>	- "Zap CP/M fcb's"

control-D	- "end ASCII string"
control-E	- "halt trace"
control-T	- "show current trace location"
<CR>	- command terminator, close cell
	- and close reg/flag
<LF>	- open sequential cell
<BS>	- open last sequential cell
<ESC>	- open "called" cell
<FF>	- open "returned" cell
;	- command separator, re-open current cell

Note that the "I", "R", "S", "S*", "SD", "SD*", and "Z" commands, and the control-T feature are not available in Debug I.

Appendix B

Error Messages

The following is a list of various ZBUG error messages.

All errors encountered are flagged by ZBUG with "*** ERROR :" followed by a two digit error code. An explaination of each error code is follows:

00: "Unknown Command Name" - An unidentified command character was encountered

02: "Can't use byte addr" - While in E command and currently in byte mode, an <ESC> was used to close the open cell. ZBUG will not permit this operation.

03: "Bad mode" - While in E command and using ';' to close the cell, an inproperly formed 'Mn' was found.

04: "Missing arg" - ZBUG expected an argument for the command that was omitted.

05: "Bad cmnd mod" - An improper modifier for the command was encountered.

06: "Missing ',', ';', or <CR>" - ZBUG expected to find a separator or command terminator.

07: "Expr too long" - The specified expression for the S or SD command exceeds 59 characters.

09: "Bad reg" - Unknown register name in X command.

10: "Bad radix" - While in E or X commands using ';Rn' to close, an improperly formed 'Rn' was found.

11: "Bad flag" - Unknown flag name in X command.

12: "Bad mnemonic delimiter" - An improper character was found delimiting an instruction mnemonic.

13: "Bad mnemonic" - An unknown instruction mnemonic was found.

14: "Bad 'hlt'" - The sequence "mov m,m" and related indexed instructions are not premitted.

15: "Can't use 2 index regs" - An instruction may not use more than one index register.

16: "Bad rel addr" - The relative branch is out of reach.

17: "Bad reg / disp (index)" - The register or displacement(index) operand is improperly formed.

18: "Too many args" - Too many arguments were encountered.

19: "Bad file name/ext" - The file name/extension specified in the O command is improperly formed.

20: "Can't open file" - The file specified in the O command doesn't exist.

21: "Read error" - A read error has occurred while reading the file specified in the O command.

22: "Record error" - An improperly formed .HEX or .REL record was found while loading the file specified in the O command. Try re-assembling the program.

23: "Program too large" - The program being loaded with the O command has attempted to load within ZBUG's bounds.

24: "Improper Relational Operator" - ZBUG found a relational operator ('?XX') that was improperly formed.

25: "Missing ')'" - ZBUG found a '(' that was not matched by a ')'.

26: "Improper or Missing Unary Operator" - ZBUG found an improperly formed radix change/explicit length operator, or expected a unary operator and didn't find it.

27: "Improper Register Name" - ZBUG found a register reference with an unknown register name.

28: "Improper Flag Name" - ZBUG found a flag reference with an unknown flag name.

29: "Missing String Terminator" - ZBUG could not find a matching string terminator before finding a <CR>.

30: "Improperly Formed Number" - ZBUG found an improper digit for the assumed radix.

31: "Unidentified Symbol" - General catch all for unrecognizable constructions.

32: "Missing ']'" - ZBUG couldn't find a ']' while processing an instruction "number".

98: "Operand Stack Underflow" - ZBUG system error. If this error happens repeatedly, please gather any pertinent information (command executing, user entered information, ZBUG generated results, etc. resulting in the error) and notify Technical Design Labs, Inc., attention Manager of Technical Services.

99: "Operand Stack Overflow" - The expression ZBUG is evaluating is too complicated - parentheses are nested too deeply or too many operations are pending.