RIO SYMBOLIC DEBUGGER

Zilog

Reference Manual

November 1978

Zilog

03-3034-01, Revision B

November 1978

RIO SYMBOLIC DEBUGGER

Reference Manual

November 1978

TABLE OF CONTENTS

PREFACE

This document describes the Zilog symbolic debuggers for use with Z-80 assembly language programs. ZBUG, the name of the debugger, is supplied in relocatable form on the RIO system disk. ZBUG contains many features making it a much more powerful programming tool than the debugger supplied in the MCZ PROM. NBUG, a newer version of ZBUG, is also supplied in relocatable form. NBUG contains several extensions to ZBUG (most notably assembly and disassembly of Z-80 instructions) which make it the more desirable of the two. It does, however, occupy more memory and, because of this, ZBUG is still supplied.

The manual is divided into three parts. A tutorial introducing the user to ZBUG is presented first. It goes through several examples in detail and a careful reading of this section is strongly recommended. The second part is a reference manual describing ZBUG in detail but giving few examples and explaining little about the use of ZBUG-the tutorial is intended for this. Some features of ZBUG are not described in the tutorial, however, so the reference manual is an important source of detail. The third part describes NBUG primarily by noting its differences from ZBUG as the two debuggers are very similar.

The appendix gives a quick reference summary of ZBUG and NBUG commands. Posting the two page summary by the terminal is recommended.

PART 1: ZBUG TUTORIAL

I. What is ZBUG?

ZBUG is an interactive debugger designed to ease the task of debugging Z80 assembly language programs. Several features of ZBUG facilitate this process. Memory can be displayed in several formats. Up to eight breakpoints can be placed in the user's program. A trip-count is associated with each breakpoint to facilitate dealing with loops. Control can be transferred to the user program for a specific number of instructions and then returned to ZBUG. Register contents can be displayed or modified. Facilities are also provided to deal with relocatable modules making manual arithmetic unnecessary and to interface to the assembler symbol table making user program symbols available. ZBUG is highly interactive - all commands are a single character and a carriage return is not required to invoke them.

What do you need to use it?

ZBUG runs only on Zilog MCZ systems. ZBUG itself is slightly over 4K (decimal) in length. Space for the user symbol table (not required but often highly useful) takes roughly 1K per 30 pages of source code. ZBUG is not particularly well suited for debugging interrupt driven programs but can be of some use (see the ZBUG reference manual for details).

What do you need to know?

This tutorial is written for the reasonably experienced assembly language programmer. It assumes knowledge of the Z-80 architecture, the Zilog RIO relocating assembler and linker, and the Zilog RIO operating system. Further details of those packages can be found in their respective manuals.

What the tutorial will and won't tell you.

This section is a tutorial. It, through several examples, illustrates the use of most of the features of ZBUG. Other, hopefully handy, techniques are illustrated. A full explanation of all ZBUG commands and features, however, is not the purpose of this tutorial. Further information can be obtained from part 2, the ZBUG reference manual.

Conventions Used in This Document.

There are several special characters used with ZBUG. In the examples that follow, they are represented as follows:

character	representation
carriage return	CR
line feed	LF
any control character	↑character (e.g., ↑A is control-A)
escape	$
$	$ (with a note that this is the real $)

To make it clear who printed what, user input is indicated in **bold face type** like that. Output printed by ZBUG is in normal type.

II. Generating a version of ZBUG

ZBUG is supplied in relocatable form. This allows it to be linked as an executable version generated at any address or to be linked directly with the user program. Of course, it also requires that you do it. Here's how:

%LINK $=7000 ZBUG (NOM P N=ZBUG70 ST=0)

This RIO command creates a procedure file named ZBUG70. It is a suggested convention that procedure files for ZBUG be suffixed with the first two digits of its load address as a reminder. Thus, a file ZBUGC8 would indicate a version of ZBUG that runs at C800. (Substitutions for the "$=7000" and "N=ZBUG70" can be made to produce versions of ZBUG that run at any desired address. Also, in LINK commands, the "$" is always the real $, not the ESC key.)

The manual entry point to ZBUG is at its first byte address.

Try the above command. Subsequent examples will assume that a version of ZBUG called ZBUG70 exists and is linked to run at 7000. (Most numbers in this document are hexadecimal.)

III. Looking Around

Opening and Closing

Imagine that each memory location and CPU register is in a box. To examine or modify the contents, the box must first be opened. This concept is central to ZBUG. To open a register, ZBUG must be ready to accept a command (i.e., have just typed its "*" prompt). Then, some name that identifies the register is typed, followed by a command that causes the register to be opened. Most such commands are single characters that not only cause the register to be opened, but also specify the format in which it will be displayed. (Note: the term 'register' is here used synonymously with 'CPU register', 'memory location', and 'ZBUG register.')

Once open, the contents of the register can be replaced. This process is described below. Next, the register must be closed and, perhaps, another one opened. A carriage return is the typical signal to close the currently open register; the prompt character "*" then appears indicating that ZBUG is waiting for the next command. A line-feed closes the current location and opens the next; an "↑" closes the current location and opens the previous (lower memory addresses).

There are several output formats available. They are often referred to as modes. ZBUG maintains a "current" mode and displays numbers in that form until another mode is selected. The modes discussed in this section are 8-bit hex, 16-bit hex, and ASCII. The names used for these are HEX8, HEX16, and ASCII.

We will use as an example the first sample program from the RIO operating system manual. The program prints a message on the console and is shown in figure 1. It is recommended that you type in and assemble the program. Then link it and load it with ZBUG70 as follows:

```
        %EDIT EXAMPLE1.MCZ.S

                type type type

        %ASM EXAMPLE1.MCZ (S)
                .
                .
                .
        %LINK $=4400 EXAMPLE1.MCZ (SY)
                .
                .
```

The S and SY options on the ASM and LINK commands are explained below.

```
                                 EXAMPLE1.MCZ                          PAGE   1
 LOC    OBJ CODE M STMT SOURCE STATEMENT                                ASM 5.3

0000    FD210800 R    1           LD       IY,AVEC
0004    CD0314        2           CALL     SYSTEM
0007    C9            3           RET
                      4
                      5  AVEC:
0008    02            6  AVLN:    DEFB     CONOUT
0009    10            7  AVREQ:   DEFB     WRTLIN
000A    1300     R    8  AVDTA:   DEFW     MSG
                      9
000C    2400         10  AVDL:    DEFW     LMSG
000E    0000         11  AVCRA:   DEFW     0
                     12
0010    0000         13  AVERA:   DEFW     0
0012    00           14  AVCC:    DEFB     0
                     15
                     16  SYSTEM   EQU      1403H
                     17  CONOUT   EQU      2
                     18  WRTLIN   EQU      10H
                     19
0013    454E4F52     20  MSG:     DEFM     'ENORMOUS CHANGES AT THE LAST MINU
0036    0D           21           DEFB     0DH
                     22  LMSG     EQU      $-MSG
                     23
                     24           END
```

Figure 1.

```
%EXAMPLE1.MCZ,ZBUG70        Load EXAMPLE1.MCZ and ZBUG70.
                            Execute ZBUG70
*                           ZBUG types its prompt character.
```

Let's look around in memory. There are several output formats to choose from:

```
*4400. FD LF
 4401 21 LF     (Recall that CR and LF
 4402 08 CR      are the carriage return and
                 line-feed keys, respectively.)
*
```

Here, we are examining the first few bytes of the program. The "." command "opens" a location (4400 in this case) and types out the contents as a hexadecimal number. Once open, ZBUG waits for input. The command **LF** (the linefeed key) closes any open location and opens the next one. The command **CR** (carriage return) closes any open location and retypes the ZBUG prompt character, "*".

```
*440A: 4413 LF              Similar to ".", ":" opens a
 440C  0024 CR              memory location but displays
                            it as a 16-bit number.  LF
                            advances the address by 2.
*440A. 13   LF              Note that ":" reverses the
 440B  44   CR              bytes, consistent with the
*                           Z80 architecture.
```

Output can also be displayed in ASCII:

```
*4413( 'E LF                The "(" displays output in
 4414  'N LF                ASCII form.  The form is
 4415  'O LF                'character or <hex number>
 4416  'R CR                if the character is non-
*4408( <02> CR                      printing.
*LF                         LF as a ZBUG command still
 4409 <10> . 10 LF          opens the next location in
                            the last format selected.
                            Once open, a location can be
                            redisplayed in another format
                            by typing the display char-
                            acter as a command.
 440A <13> : 4413 CR        Here, 440A, originally
                            open in ASCII format, is re-
                            displayed as a 16-bit Hex
                            number.
```

"↑" has an effect similar to **LF** except that the previous location, not the next, is reopened:

```
*4416( 'R ↑            Each ↑ closes the current
 4415  'O ↑            location and opens the pre-
 4414  'N ↑            vious one.
 4413  'E . 45 CR
*
```

Note the following:

1) **LF** and ↑ can be used as commands opening the next and last locations based on the most recently examined location. The output mode is whatever the most recent one was.

2) Once open, a register can be redisplayed in another format by typing the appropriate command character. Redisplaying a register does not change the "current" mode.

CPU Registers

CPU registers can be opened, displayed, modified, and closed. There is, however, no notion of a 'next' or 'last' register, as there is with memory. The **↑R** command is used to display or open registers. When given with no arguments, **↑R** displays a standard set of registers. The **↑R** command is also used to open registers. First, the register name is typed, followed by **↑R.** All output is in HEX8 or HEX16 format based on the size of the register. Consider the following examples:

```
*↑R
 PC  A  B  C  D  E  H  L  F   IX   IY   SP  A' B' C' D' E' H' L' F'
7000 70 1C D3 17 E4 70 00 20 0000 15FA 1958 00 00 00 00 B9 26 E9 90
*
```

Recall that **↑R** represents a control R. The register names are:

$A	$A'	$IX
$F	$F'	$IY
$B	$B'	$SP
$C	$C'	$PC
$D	$D'	$I
$E	$E'	
$H	$H'	
$L	$L'	

where $PC is the program counter and $I is the interrupt vector register. Recall that the character '$' represents ESC unless otherwise noted.

***$A↑R** 70 **CR**
***$IX↑R** 0000 **CR**

The CPU registers are saved each time ZBUG is entered and restored each time it returns to a user program. Thus, any change to a register would affect what is seen by the user program when it is executed.

ZBUG Registers

ZBUG itself has several registers used in controlling its operation. These are opened, displayed, modified, and closed in manners similar to the above. Different commands are used, however, and these will be discussed later.

IV. Changing Things

Memory, CPU registers, and ZBUG registers can be modified. The technique is fairly simple - first, open the register. Then, type an expression representing the desired new contents. Finally, close the register in one of the ways described above.

Occasionally, a series of numbers is to be entered in memory. An output mode, called QUIET, is provided so that it is possible to open locations without displaying their contents each time. The command character "!" opens a location in this mode.

```
*9000!  1  LF        Open with no display.  Put in
9001    2  LF        the value 1, then open the next
9002    3  CR        location, put in 2, and so on.
*
```

Note that QUIET mode behaves as HEX8 (except for display), changing the location counter one byte at a time. Only the low 8 bits of any expression input are significant.

Let's revisit example 1 and change the message.

```
*4413('E  'W  LF
4414  'N  'R  LF
4415  'O  'O  LF
4416  'R  'N  LF
4417  'M  'G  LF        ' followed by a character
4418  'O  '   LF        has the ASCII value of the
4419  'U  '   LF        character.
441A  'S  '   CR
*4413('W    LF
4414  'R    LF
4415  'O    LF
4416  'N    LF
4417  'G    LF
4418  '     LF
4419  '     LF
441A  '     LF
441B  '     LF
441C  'C    CR
*
```

Also, expressions can be used anywhere:

```
*4402:  4408  4400+34-4 CR        Expressions are evaluated.
*4400+2:  4430  4408 CR           Change things back.
```

Registers are similarly altered:

```
*$PC↑R  7000  4400 CR             Change PC to 4400
*
```

Better Ways to Specify Locations

Clearly, debugging relocatable modules using only absolute addresses is difficult at best. It is necessary to have a load map from the linker and manually compute absolute addresses by adding the module load address (from the load map) to the offset of the desired location (found in the assembler listing). ZBUG allows all input to be an expression, eliminating the need for manual computation. However, a better way is still desirable. ZBUG provides a 'displacement' register which can be set to any value. If it is set to a module address then relative addresses can be entered. The format of a relative address is a number followed by the character "'" (a single quote mark). Such a number is added to the value in the D register of ZBUG and then the result is used in place of the original number. (Don't be confused with the Z80 D register. Here we are talking about a 16 bit register in ZBUG.)

```
*↑D   0000   4400 CR
```

↑D opens the displacement register. The old value is displayed and a new one entered.

```
*8'.  02 LF
0009'   10 LF
000A'   13 CR
*
```

The relative address is followed by the character to open the location in the desired mode. Note that the addresses are output in relative form.

To deal with the output of relative addresses, a displaced output format is provided. This mode is called DHEX16 and locations can be opened in this mode by using the command character "[". If a value is less than the D register, it is displayed in HEX16 format to avoid negative displacements.

```
*2'[  0008'  : 4408  CR
*
```

In the above example, location 4402 is first opened in DHEX16 mode, then redisplayed in HEX16 mode, and then closed.

```
*↑Q
%
```

↑Q (for Quit) leaves ZBUG and returns to RIO.

User Symbols

Still, it would be nice to access the labels used in the source program. Conveniently, ZBUG can do this. All global symbols and module names are accessible but local labels are available for only one module at a time.

There are several steps necessary to use this feature:

1) Assemble and link your program with the S and SYm options. This causes the assembler to include the local symbols in the object file and causes the linker to produce a file with the same name as the procedure file but with a suffix of ".SYM". The example at the beginning of section III illustrates this.

2) Prior to beginning a debugging session, allocate memory immediately following ZBUG by using the RIO command ALLOCATE to reserve space for the symbol table. ZBUG will (on command) load the user symbol table immediately following itself in memory. It also does not interact with the RIO memory manager while doing this - hence the need to do it manually. In fact, it is not always necessary to do this allocation, but if your program causes or performs any memory management calls, it will, most likely, be necessary.

3) Load your program and ZBUG with control going to ZBUG.

4) The ↑**E** and ↑**L** commands in ZBUG are used to load the symbol table. ↑**E** is used to specify the name of the procedure file and to load the symbol table in memory with global symbols and module names. If the program has a module with the same name as the procedure file, the locals for this module are also loaded. (If no such module module exists, then the message "??" is issued, but everything is otherwise okay.) The ↑**L** command specifies a module name whose local symbols are loaded, replacing in memory the locals of the last module to be there. (Recall that locals from only one module at a time are available.)

Once loaded, any symbol can be substituted in an expression for any number. Symbols must be prefixed with ESC (which is printed as a '$').

In the following example, we will assume the assembly has been done already as shown above.

```
%A 8400 9400 1000                 Reserve 4K for symbol table.
%EXAMPLE1.MCZ,ZBUG70

*↑E    EXAMPLE1.MCZ CR
*
```

Since there is only a single module here, both globals and locals are loaded. (There aren't any globals in this example, anyway.) Let's look around again:

```
*↑D  0000   4400 CR
*$AVEC.   02   LF                 Open location with label AVEC.
AVREQ     10   LF
AVDTA     13   [  0013'   LF      Redisplay AVDTA in displaced number
000B'     44   LF                 format.
AVDL      24   CR
*
*$LMSG=   0024                    Evaluate an expression.
*$MSG+$LMSG-1(   <0D>   ↑         Look at last character in message.
0035'    'E   CR
*
```

Needless to say, the symbols come in very handy.

V. Running Programs

The process of debugging supported by ZBUG is based on watching the execution of the user program including control flow and changes in data structures as execution proceeds. Thus, there are provisions for executing part of the program and then having control return to ZBUG. Then, memory and registers can be examined and modified and control can be returned to the user program. Through a series of steps such as these, the point in the program at which "things go wrong" can be isolated and, finally, bugs identified and obliterated.

Control can be transferred to the user program at any address. It is possible to execute one or any number of instructions and then have control return to ZBUG. This process is referred to as 'stepping'. Up to eight 'breakpoints' can be placed in the user program. A breakpoint is a connection between ZBUG and the user program and is placed at a specific location in the user program. When control comes to that location, the normal flow of control is halted and control comes to ZBUG. Then, memory and registers can be examined as usual and control can be returned to the program at the breakpoint, continuing execution as though nothing had happened. In a sense, ZBUG has been 'inserted' between two locations in the user program.

In the following several examples, it is assumed that the sample program used above is loaded and that the symbol table has been allocated and loaded.

First, let's just run the program. The **↑G** command transfers control to the address represented by the expression given immediately before it.

```
*4400↑G                                 Start it up.
ENORMOUS CHANGES AT THE LAST MINUTE
%
```

Control ends up at RIO. Let's get back to ZBUG

```
%X 7000                        RIO goes back to ZBUG
*↑Q                            Quit from ZBUG.  Back to RIO
%
```

Why the jumping back and forth at the end? Control is first transferred to the sample program and then to RIO. ZBUG still thinks the 'user' program is running as it started the user program and never heard anything to say that it was done. So ZBUG is sitting waiting for, for example, the NMI ('BREAK') button to be pressed (which transfers control to ZBUG). However, once back at RIO, we are effectively through with the run and the memory space allocated for ZBUG and the example program has been deallocated. It is a good idea to tell ZBUG that its through, too.

Next, let's backup and load the original program again:

```
%A 8400 9400 1000                 This is not necessary if it
                                  was done already in the last example.
                                  The allocation is reset only by
                                  issuing an appropriate DEALLOCATE
                                  command or re-bootstrapping.
%EXAMPLE1.MCZ,ZBUG70

*↑E   EXAMPLE1.MCZ CR
*
```

Let's put a breakpoint at 4' (just before the call to SYSTEM).

```
*↑D   0000  4400 CR
*4'↑B
*
```

The ↑B command sets a breakpoint at the address specified by its argument.
With no arguments, it lists all active breakpoints and their numbers:

```
*↑B
0B  4404   1B      2B      3B      4B      5B      6B      7B
*
```

Thus, breakpoint number zero is set at location 4404. Run the program:

```
*4400↑G

B0 0004'
*↑R                               Control came back to ZBUG; list
                                  registers.
 PC   A  B  C  D  E  H  L  F  IX   IY   SP   A' B' C' D' E' H' L' F'
4404 70 1C CE 17 F1 70 00 20 0000 4408 44FB 00 00 00 00 B9 2B E3 90
*
```

IY has been loaded. Now execute the next instruction:

```
*↑S                               Single step

S  SYSTEM
*                                 The call to SYSTEM has been made.
*$SP↑R  44F9 CR                   Look at stack pointer.
*%:  4407  CR                     Look in stack.
*%↑B                              Place breakpoint there.
*
```

First, the CALL instruction was executed (by single stepping). Control returned to ZBUG with the message "S SYSTEM" informing the user that control came to it at the conclusion of a step operation and the next instruction to be executed is at address SYSTEM. Then, the stack pointer register is opened, displayed, and closed. The symbol "%" has a special meaning to ZBUG. In an expression, it has the value of the last register opened (or memory location opened). Thus, the command **%:** opens the location whose address was just printed - in this case, the top of the stack. Since the instruction just executed was a CALL, the value on top of the stack is the return address, 0007'. In the next line, a breakpoint is placed at that address, again using the symbol % to stand for the last value ZBUG typed out. The above sample sequence is frequently used when stepping through code: A subroutine call is encountered and one wishes not to step through the subroutine, but to continue stepping when it returns.

***↑P**	Continue execution. The **↑P** command is used to proceed from a breakpoint.
ENORMOUS CHANGES AT THE LAST MINUTE	This is the program output.
B1 0007' *	The breakpoint is encountere and control returns to ZBUG.

The breakpoint was encountered when the system returned after printing the message on the console. ZBUG is running again. It is possible to start the program at the beginning again by transferring control to location 4400.

It is desirable sometimes to have ZBUG list the registers after each step and at each breakpoint. There is a location in ZBUG called $RSWITCH that controls this.

***$RSWITCH.** 00 **1 CR**	Put a 1 in.
***4400↑G**	Start running at 4400.

```
B0  0004'
  PC   A  B  C  D  E  H  L  F  IX   IY   SP   A' B' C' D' E' H' L' F
4404 80 00 00 1F 4B 00 24 54 28C2 4408 44FB 24 00 00 00 B9 2B E3 2
```

***↑P** Proceed from last breakpoint.

ENORMOUS CHANGES AT THE LAST MINUTE

```
B1  0007'
  PC   A  B  C  D  E  H  L  F  IX   IY   SP   A' B' C' D' E' H' L' F
4407 80 00 00 1F 4B 00 24 54 28C2 4408 44FB 24 00 00 00 B9 2B E3 2
*
```

This time the registers were automatically listed when the breakpoint was encountered.

***↑Q**	Quit, return to RIO.

VI. A Second Example

Now, let's go through another, more complex, example. This example is a program to sort numbers using the bubble sort algorithm. Although the bubble sort is one of the least efficient sorting algorithms, it serves well to illustrate the use of ZBUG. The program is (somewhat unnecessarily) broken into three modules to illustrate techniques used when dealing with multiple modules.

The first module is the main loop of the program. It first generates the numbers, then goes into a loop that makes a pass over the numbers, exchanging any two that are out of order, and finally prints the resulting array of numbers.

The second module, called EX2.2, contains the routine that makes a pass over the array, exchanging any two consecutive numbers not in ascending order. If an exchange is made, a flag is set.

The third module contains the array generating procedure and the output conversion and RIO interface code.

The listing of the source code follows. It is recommended that you refer to it continually while following the subsequent discussion.

```
;        EXAMPLE 2 - BUBBLE SORT IN SEVERAL MODULES
;
;        THE MODULES ARE:          1)  READ NUMBERS, MAIN SORT LOOP,
;                                      PRINT NUMBERS
;                                  2)  INNER SORT LOOP (MAKES ONE PASS)
;                                  3)  INPUT AND OUTPUT.
;
;
GLOBAL  SWAP               ; Element swapped flag
        GLOBAL  ARAY               ; Holds the actual numbers
        GLOBAL  LOW,HIGH           ; Point to first and last locations
                                   ;  of area to be sorted.

EXTERNAL        LOAD,PRINT         ; Routines to read and print the
                                   ;  numbers
EXTERNAL        PASS               ; Make one pass over the array.



BEGIN:  LD      HL,ARAY            ; -> Beginning of array of numbers
        CALL    LOAD               ; Read the numbers and return:
        LD      (LOW),HL           ;      index of first element, and
        LD      (HIGH),DE          ;       index of last element.

; Loop here for each pass over the array.  Each pass moves the
; largest number to the end of the array.  If, after a pass, the
; swap flag is still zero, then no numbers were exchanged and the
; array is sorted.

NPASS:  XOR     A                  ; =0
        LD      (SWAP),A           ; Clear swap flag.

        LD      HL,(LOW)           ; HL = index of first element
        LD      DE,(HIGH)          ; DE = index of last element
        CALL    PASS               ; Make a pass over the array

        LD      A,(SWAP)           ; See if any exchanges were made
        OR      A
        JR      NZ,NPASS           ; Yes, pass over the array again.

; All done, print results
        LD      BC,ARAY            ; BC -> array
        LD      HL,(LOW)           ; HL = index of first element
        LD      DE,(HIGH)          ; DE = index of last element
        CALL    PRINT

; Back home to RIO
        RET

LOW:    DEFS    2                  ; Index of first element
HIGH:   DEFS    2                  ; Index of last element in array
SWAP:   DEFS    1                  ; Swapped elements flag

ARAY:   DEFS    100                ; Space for the array of numbers
        END
```

MODULE 2 OF EXAMPLE 2 - PASS OVER THE ARRAY EXCHANGING ELEMENTS OUT OF ORDER

```
        GLOBAL  PASS
        EXTERNAL        ARAY,SWAP

;;PASS
; PASS - Make a pass over the array.

;   A single pass is made over a specified area of memory.  Any
;   adjacent numbers out of order are exchanged.  If any exchanges
;   are made, the SWAP flag is set.

;   HL = index of first element in ARAY
;   DE = index of last element
;   SWAP flag is zero
;   CALL    PASS
;   <RETURN>        SWAP flag set if any exchanges are made.


   PASS:   LD      (last),HL         ; Save indices
           LD      (current),DE

; Loop here for each element.  See if done.
   NCHK:   LD      HL,(current)
           INC     HL                ; Move to next
           LD      (current),HL
           DEC     HL                ; HL = index if next elt to look at.

           LD      DE,(last)         ; DE = index of last elt to look at.
           OR      A
           SBC     HL,DE             ; = Curr - last
           RET     NC                ; Current >= last, all done with
                                     ;  this pass.
           ADD     HL,DE             ; Restore HL

           LD      DE,ARAY
           ADD     HL,DE             ; HL -> element
           LD      A,(HL)
           INC     HL
           CP      (HL)              ; Compare ARAY[current] and
                                     ;       ARAY[currnet+1]
           JR      C,NCHK            ; No exchange necessary, move to next
                                     ;  element
           LD      B,(HL)
           LD      (HL),A            ; Exchange elements
           DEC     HL
           LD      (HL),B

           LD      A,1
           LD      (SWAP),A          ; Set swap flag
           JR      NCHK

   current:DEFS    2                 ; Index of current element to look at
   last:   DEFS    2                 ; Index of last element to look at
           END                       ; of module 2
```

```
;
;
;
;       EXAMPLE 2 MODULE 3 - Generation and output of numbers


        GLOBAL  LOAD,PRINT


;;LOAD
; LOAD - Generate some random numbers to be sorted.

;       HL -> array area.  This area must be at least 30 bytes long.
;       CALL    LOAD
;       <RETURN>        HL = index of first number
;                       DE = index of last number


LOAD:   LD      A,13
        LD      B,30

; Loop here for each number
NUMB:   LD      (HL),A
        ADD     A,157   ; Next number = current + 157 MOD 256
        DJNZ    NUMB

        LD      HL,0    ; Low index
        LD      DE,29   ; High index
        RET


;;PRINT
; PRINT - Print a series of 8 bit numbers

;       Unsigned numbers are output to the console, converted from
;       a specified area of memory.

;       BC = Array base address
;       HL = index of first number to output
;       DE = index of last number to output
;       CALL    PRINT
;       <RETURN>                all done


PRINT:  ADD     HL,BC           ; -> First number
        EX      DE,HL
        ADD     HL,BC           ; -> Last number
        EX      DE,HL
        INC     DE              ; -> Last+1
```

```
; Loop here for next number
PRTNXT: PUSH    HL
        OR      A
        SBC     HL,DE
        POP     HL
        RET     NC                  ; Current > last, done

        LD      A,(HL)              ; = Number to print
        PUSH    HL
        PUSH    DE                  ; Save DE,HL
        CALL    OUT8                ; Print A
        POP     DE
        POP     HL
        INC     HL
        JR      PRTNXT              ; Loop for next


;;OUT8
; OUT8 - Print an 8 bit number in A


OUT8:   PUSH    AF
        RRA
        RRA
        RRA
        RRA                         ; Left digit first
        CALL    OUT4
        POP     AF
        CALL    OUT4                ; Then right digit

        LD      A,CR
        CALL    OUTCH               ; Print a CR
        RET                         ; Wasn't that easy?


;;OUT4
; OUT4 - Output a digit in the right 4 bits of A

OUT4:   AND     0FH
        CP      10
        JR      C,OUT4B             ; Hex is so nasty!
        ADD     A,'A'-('9'+1)       ; Convert to ABCDEF
OUT4B:  ADD     A,'0'               ; Make a digit
        CALL    OUTCH               ; Output a character
        RET
```

```
;;OUTCH
; OUTCH - Output a character to the console.


OUTCH:  LD      (chr),A             ; Set the data area

; Prepare the vector and call RIO
        LD      HL,1
        LD      (DL),HL
        LD      IY,VECTOR
        CALL    SYSTEM

        LD      A,(ccode)           ; Check completion code
        CP      80H                 ; OK?
        RET     Z                   ; Yes

; Panic - RIO error
PANIC:  JR      PANIC               ; Hang up here
                                    ; (this will never happen)


VECTOR:
        DEFB    CONOUT              ; Console
        DEFB    WRTLIN              ; Write
        DEFW    chr                 ; -> Buffer
DL:     DEFS    2                   ; Length
        DEFW    0                   ; Completion return address
        DEFW    0                   ; Error return address
ccode:  DEFS    1                   ; Completion code

chr:    DEFS    1                   ; A short buffer


SYSTEM  EQU     1403H               ; RIO entry point for MCZ
CONOUT  EQU     22                  ; Console logical unit number
WRTLIN  EQU     10H                 ; Write code
CR      EQU     ' '                 ; Carriage return

        END
```

The assemblies are performed as follows:

%ASM EX2.1 (S); ASM EX2.2 (S); ASM EX2.3 (S)

%LINK $=4400 EX2.1 EX2.2 EX2.3 (SY)

Next, we want to try the program on the chance that it will work the first time. Either we can resist the temptation or just try it (it goes into an infinite loop). With that out of the way, it's time to find the bug(s). (Note the optimism in writing "bug(s)", implying that there might be just one.)

We will use ZBUG with the symbol table of the program. Because the program was assembled and linked with the SYM option, the file EX2.1.SYM exists and contains the symbol table for use by ZBUG. Space for the symbol table should be allocated immediately above ZBUG in memory. To find the low address to use, type:

%EXTRACT ZBUG70

(We are still using ZBUG70 here.) The first address to allocate is the high address of ZBUG rounded up to the next 80H byte boundary, in this example 8400H. This program doesn't need much space but since there is a lot available, we'll allocate 1000H bytes. (Again, this is not necessary if already done.)

%A 8400 9400 1000

The space in memory above ZBUG is now reserved so that RIO won't allocate any space in that area. Next, load the example program and ZBUG:

%**EX2.1,ZBUG70**	and off we go.
***↑E EX2.1 CR**	Load the global symbols
***↑D** 0000 **$BEGIN CR**	Set displacement register to beginning of first module.

Next we must execute the program slowly and in parts, checking the results of each part. By doing this, we can look for a part of the program that is not functioning properly and also verify that other parts are functioning properly. The subroutine LOAD is the first one called, let's break there.

***$LOAD↑B**	Set breakpoint at entry to LOAD
***BEGIN↑G**	Begin execution at label BEGIN.
B0 EX2.3 *	The breakpoint was encountered and control comes back to ZBUG.

Notice that the location indicated was EX2.3 rather than LOAD. An examination of the listing reveals that LOAD is the first address in the module EX2.3. When listing addresses, ZBUG prints the first symbol it finds that matches the address so that module names tend to appear instead of other labels on the first location of a module.

Let's proceed to the return of the LOAD subroutine. It should fill the array ARAY with random numbers and return the indices of the first and last elements of the array in registers.

```
*$SP↑R  4679  CR
*%:     4406  CR          This is the return address in
*%↑B                      the stack. Place a breakpoint
*                         there. Recall that '%' has the
                          value of the last number
                          printed.

*↑P                       Proceed from current breakpoint.

B1  0006'                 The second breakpoint is
                          encountered.
*↑R                       Look at the registers.

 PC  A  B  C  D  E  H  L  F  IX   IY   SP   A' B' C' D' E' H' L' F'
4406 73 00 CE 00 1D 00 00 35 0000 15FA 467B 00 00 00 00 B9 2B E3 90
   *
```

HL has zero, a reasonable number for the index of the first array element. DE has 1D, again a reasonable value for the index of the last element. Let's look are the array itself:

```
*$ARAY.  D6  LF
0035'    FF  LF
0036'    FF  LF
0037'    FF  LF           These numbers are not reasonable
0038'    FF  CR           at all!
*
```

Something must be wrong with the LOAD routine - it generates very poor random numbers. We can restart the program from the beginning and this time go through LOAD in more detail.

```
*$BEGIN↑G                      The breakpoints are still in,
B0  EX2.3                      and we arrive at LOAD.
*↑L   EX2.1  EX2.3 CR          Change the local environment
                               to the third module, as that is
                               where LOAD is.
*$NUMB↑B                       Break at NUMB, in the loop.
*↑P                            Go 2 instructions forward to
                               there.

B2  NUMB
*↑R                            Check the registers.

 PC  A  B  C  D  E  H  L  F   IX   IY   SP  A' B' C' D' E' H' L' F'
44CC 0D 1E CE 00 1D 44 34 35 0000 15FA 4679 00 00 00 00 B9 2B E3 90
*$ARAY=4434
*
```

Control is now at the label NUMB, the beginning of the loop to generate and store random numbers in ARAY. HL should point to the first element. Printing the registers reveals that HL=4434. In the next line, we ask ZBUG to evaluate the expression $ARAY. It has the value 4434, verifying that HL has the correct value. Register B has the count of numbers to be generated, 1E. Let's go through the loop a few times and see what changes (or fails to).

```
*10↑P

B2  NUMB
*
```

The command **10↑P** tells ZBUG to proceed from the last breakpoint and also not to report the occurrence of the breakpoint (breakpoint number 2, in this case) until it has been encountered 10H times. Thus, the registers should now look as though the loop had been executed 10H times.

```
*↑R
 PC  A  B  C  D  E  H  L  F   IX   IY   SP  A' B' C' D' E' H' L' F'
44CC DD 0E CE 00 1D 44 34 88 0000 15FA 4679 00 00 00 00 B9 2B E3 90
```

Note that register B has been decremented 10H times as expected, however, HL seem unchanged. A glance at the code evokes an "ah-ha" experience as we see that an

```
INC HL
```

instruction is missing from the loop. While we have a moment, let's look at another way to use breakpoints with loops. A command like **10↑P** proceeds through the next 10 occurrences of the breakpoint at the location to which the proceed command sends control. Such a count can be established as a default by setting the N register (a ZBUG register unrelated to the Z80) to a certain number. For example, suppose we wish to step through the loop 3 iterations at a time:

```
*2↑N  0001  3 CR                 Set N register for breakpoint
                                  2 to 3.
*↑P                               Then proceed.

B2  NUMB
*$B↑R  0B CR                      B has been decremented by 3
*↑P

B2  NUMB                          Three more times
*$B↑R  08 CR                      B is decremented by 3 again.

*$H↑R  44 CR
*$L↑R  34 CR                      HL hasn't changed.

*2↑N   0003  1 CR
*2↑K   0003  1 CR                 Reset N and K registers to 1.
```

The K register is the trip-count used to control each breakpoint. Each time a breakpoint is encountered, its trip-count is decremented. When the count reached zero, the break is reported on the console. Otherwise, execution of the user program continues. A command of the form **n↑P** sets the K register of the breakpoint at the current location to n. If control is resumed from a location that has no breakpoint, the n is ignored. When the trip-count reaches zero and the break is reported, the K register is automatically reset to the value in the corresponding N register. Thus, setting the N register for a particular breakpoint establishes a default trip-count.

Now that the trip-count for breakpoint 2 is back to 1, let's check that the loop is producing the proper random numbers. Each one should be 157 (9D hex) MOD 256 from the last.

```
*$A↑R  8B CR
*↑P                               Go through the loop again.

B2  NUMB
*$A↑R  28 CR                      This is the new value in A.

*8B+9D=0128
*
```

Thus, the A register is advanced properly each time through the loop.

```
*↑Q
%                                 Back to RIO to EDIT in the fix.

%EDIT EX2.2.S
```

Make a change so that LOAD reads:

```
LOAD:   LD      A,13
        LD      B,30

; Loop here for each number
NUMB:   LD      (HL),A
        INC     HL
        ADD     A,157   ; Next number = current + 157 MOD 256
        DJNZ    NUMB
```

Assemble and link:

```
    %ASM EX2.3 (S);LINK $=4400 EX2.1 EX2.2 EX2.3 (SY)
```

We didn't deallocate memory so the space for the symbol table is still protected. Start the debugging process again.

```
    %EX2.1,ZBUG70

    *↑E    EX2.1 CR
    *↑D  0000   $BEGIN CR
```

Now confident that LOAD works, we will break initially at NPASS, the loop point in the main module.

```
    *$NPASS↑B
    *$BEGIN↑G

    B0  NPASS
    *$ARAY.  0D LF
    0035'      AA LF
    0036'      47 LF              These are more reasonable
    0037'      E4 LF              numbers to be sorted.
    0038'      81 CR
    *↑R
 PC  A  B  C  D  E  H  L  F   IX   IY   SP  A' B' C' D' E' H' L' F'
440D 73 00 CE 00 1D 00 00 35 0000 15FA 467B 00 00 00 00 B9 2B E3 90
```

Thus, coming back from LOAD we see that ARAY has reasonable numbers in it. HL has the low index, zero, and DE has the high index, 001D. Let's check the variables LOW and HIGH:

```
    *$LOW:   0000 LF              LF opens next word
    HIGH     001D CR
    *
```

They are ok. We have a breakpoint at NPASS, the loop point of the main loop. If we proceed, control should come to NPASS (causing a break) for each pass over the array. Eventually, control will go out of the loop, eventually arriving at the routine PRINT. We will place a breakpoint there to catch control when it gets out of the loop.

```
*$PRINT↑B
*↑P                                  Go ahead and make a pass
                                     over the array.  Control
                                     should return to ZBUG when
                                     the pass is over and control
                                     loops back to NPASS.

B1 PRINT
*
```

Why did control go to PRINT? Could the array already be sorted?

```
*$ARAY.   0D LF
0035'     AA LF
0036'     47 CR
```

No, these numbers are not in order. The code says that control comes out of the loop only if SWAP (a flag) is zero.

```
*$SWAP.   00 CR
*
```

It is zero and this implies that the problem is in the second module in the subroutine PASS. Let's look at that.

```
*↑L  EX2.1  EX2.2 CR                 Change the local environment.
```

We want to begin again and watch as PASS is executed. Where are the breakpoints now?

```
*↑B0 440D B1 44D9 B2      B3      B4      B5      B6      B7
```

The command ↑B with no arguments lists the breakpoint numbers and their addresses. Only two breakpoints are active at the moment. Unfortunately, we have to figure out where those addresses are. We can guess and check:

```
*$PRINT=44D9                         PRINT is one
*$NPASS=  UND ??                     The symbol NPASS is in
                                     the first module and we
                                     no longer have access to its
                                     locals.
*$PASS=4498                          PASS isn't one.
*$LOAD=44C8                          LOAD isn't one, either.
```

Well, at this point we can guess that NPASS is breakpoint 0 and go on. Let's break on entry to PASS and check its execution carefully.

```
*$PASS↑B                             Set the break, then start
*$BEGIN↑G  UND??                     over.  Oops.
*$EX2.  UND??
*4400↑G
```

Don't forget that BEGIN is local to module EX2.1. Locals are accessible from only the current module; in this case we are in EX2.2. Module names and globals are always available and our program begins at the first word of the module EX2.1 so we could use that name to start running. Unfortunately, symbols that include the character '.' can only be used in the **↑E** or **↑L** commands so we must resort to the old reliable form - absolute hex addresses!

```
B0 000D'                     The first break is at NPASS.
*↑D  EX2.1 $PASS CR          Set the displacement register
                             for the module we're in.
*↑P

B2 EX2.2                     This break is at PASS.
*
```

Let's look at the interaction above and note some details. The displacement (↑D) register is not changed by changing the local environment (with ↑L). Thus, the displacement register has been EX2.1 the whole time. When breakpoint 0 was encountered the addresses was printed as 000D' because:

1) There was no symbol available that matched the address (NPASS is local to EX2.1), and
2) The displacement register was less than the address NPASS so that the displacement would be positive.

When the address printed in that form, we realized that we hadn't changed the D register to the beginning of the module currently being investigated, EX2.2. The symbol used to set it was PASS which fortunately has the same value as EX2.2 since EX2.2 contains the character '.' and can't be used to set a register.

Next, let's move forward a few instructions to the first place that something interesting happens: the test for pass complete.

```
*$NCHK↑B
*↑P

B3 NCHK
*4↑S                         Step 4 instructions.

S 000F'                      Control returns to ZBUG.
*S ??                        Oops, forgot to hold down
                             the 'control' key.
*↑S                          Step one instruction.

S 0013'                      Now, DE=last, HL=current.
*↑R
```

```
 PC  A  B  C  D  E  H  L  F   IX   IY   SP  A' B' C' D' E' H' L' F'
44AB 00 00 34 00 00 00 1D 40 0000 15FA 4677 00 00 00 00 B9 2B E3 90
```

At this point it appears that DE, the index of the last element, is zero, and HL, the index of the first element, is 001D. This is clearly wrong, reversed, in fact. A look at the code at the beginning of the subroutine reveals a conflict between the code and the comments above about the calling sequence. If we believe the comments to be correct, the code reverses DE and HL at the entry to PASS. If we step a few more to see what happens:

```
*2↑S

S 0016'
*↑S

S 441B                        Control returned to the
*                             caller, in NPASS.
```

We could, at this point, edit, reassemble, and relink but instead let's fix the code (since it's easy) and go on debugging. Be sure to note in a log that the bug is found and to edit in the fix.

If we reverse the addresses in the two store instructions at the beginning of PASS we can go on looking for more bugs.

```
*$PASS+1[  002E'  $curren CR
*$PASS+3+2[  002C'  $last CR
*
```

Here, the address in the first instruction is displayed as a displacement and the new value (the address of 'current') is typed to replace the value at that location. Then the address in the next instruction is displayed (the first instruction is three bytes long, there are two bytes before the address in the second instruction) and replaced with the proper address. Note that the symbol 'current' must be entered as 6 characters as the assembler recognizes only that many. PLZ identifiers and module names, however, may be longer.

```
*4400↑G                       Start over again.

B0 440D                       Break at NPASS.
*↑P

B2 EX2.2                      Break at PASS.
*↑P

B3 NCHK                       Break at NCHK in PASS
*
```

Let's check that 'last' and 'current' have the proper values:

```
    *$last:  001D ↑              last is ok,
    curren   0000 CR             current is ok, also.
    *↑P

    B3 NCHK                      We've gone through the PASS
                                 loop once.
    *↑P                          Do it again.
    B3 NCHK
    *3↑X                         Delete breakpoint #3.
```

The ↑X command with no arguments deletes all breakpoints; with a single argument it deletes the specified breakpoint. The argument is the breakpoint number, not the address.

```
    *↑P
    B0 440D                      This break is at NPASS,
                                 main loop of EX2.1.
    *$SWAP.  01 CR               SWAP is set, as expected.
    *↑P

    B2 EX2.2                     Break at PASS.
```

It looks like PASS may work; let's not break there any more but rather look at the array as it changes.

```
    *2↑X
    *↑P

    B0 440D
    *↑P
    B0 440D                      We've been through the main
                                 loop a few times, let's look
                                 at it again.
    *↑L  EX2.2 EX2.1 CR
    *$HIGH:  001D CR
    *$ARAY,$ARAY+%.
4434 0D 47 1E 81 58 AA 92 2F BB 69 06 A3 40 CC 7A 17   .G..X*./;i.#@Lz.
4444 B4 51 DD 8B 28 C5 62 E4 9C 39 D6 EE F5 FF         4Q].CEbd.9Vnu.
    *
```

In the above, first the local environment is changed back to EX2.1. Then the variable HIGH is displayed. Finally, a block of memory is dumped in hex and ASCII. The format of the dump command is

    low,high.

where low and high are the lower and upper addresses to be dumped. (The '.' is the command character.) In the above, the low address is ARAY, the first location of the array. The special character '%' has the value "the last number typed out"; in this example it has the value 001D. Thus, the high address is ARAY+001D.

```
    *3↑P                          Loop 3 more times.

    B0 NPASS
    *$ARAY,$ARAY+1D.
4434 0D 1E 47 58 2F 81 69 06 92 40 A3 7A 17 AA 51 B4   ..GX/.i..@#z.X(
4444 8B 28 BB 62 C5 9C 39 CC D6 DD E4 EE F5 FF         .(;bE.9LV]dnu.
    *
```

Here we can see that the larger numbers are moving to the end of the array. Let's let the program go and finish sorting.

```
    *0↑X                                Delete breakpoint #0.
    *↑B0      B1 44D9 B2       B3       B4       B5       B6       B7
    *↑P

    B1 PRINT                      Break at PRINT routine.
    *
```

The sorting loop has finished. Control is now at the PRINT routine. Let's look at the array:

```
    *$ARAY,$ARAY+1D.
4434 06 0D 17 1E 28 2F 39 40 47 51 58 62 69 7A 81 8B   ....(/9@GQXbiz.
4444 92 9C A3 AA B4 BB C5 CC D6 DD E4 EE F5 FF         ..#*4;ELV]dnu.
    *
```

Terrific, the numbers are sorted! Next, we should trace through the output code some.

```
    *↑L  EX2.1  EX2.3 CR          Change modules.
    *$OUT8↑B                      Break at number output code.
    *↑P

    B0 OUT8
    *$A↑R  06  CR                 A should have the number to
                                  print (the first one).
    *↑P                           Onward.
```

When we broke at OUT8, the number printing routine, register
had the first number to print. We then let control proceed
expecting to break again when the second number was to be ou
Unfortunately, after waiting several seconds, nothing has
happened. Control seems to have gone into never-never land (or
some infinite loop, at least). We can cause control to return
to ZBUG by initiating a non-maskable interrupt. This is done
by pressing the 'BREAK' or 'NMI' button on the MCZ panel (the
button is next to the reset button). (The button might also
be labelled 'MON'.) Our program has gone away so it's time
to press it.

```
    <press BREAK button>
    ??B PANIC
    *
```

When ZBUG is entered in this way it behaves as though it had
encountered a breakpoint but prints ??B instead of the number.
PANIC is the address at which the break occurred. Checking the
listing, we see that control goes to PANIC, an intentional
infinite loop, if RIO returns an error when trying to print
a character on the console. Register A has the error number.

```
    *↑R
 PC  A  B  C  D  E  H  L  F   IX   IY   SP  A' B' C' D' E' H' L' F'
4425 42 44 34 44 52 00 00 87 0000 4527 466B 00 00 00 00 B9 2B E3 90
    *
```

Code 42 is 'Invalid Unit', not something expected when printing
on the console. What unit did RIO receive in the parameter
vector?

```
    *$VECTOR. 16  CR            Not the right number.
```

What is CONOUT, then?

```
    *$CONOUT=16
    *
```

With that hint, we notice that CONOUT is equated to 22, not 2.
With that error discovered (and another one remaining to edit
out) we end the debugging session.

```
    *↑Q
```

In the course of this example several ZBUG commands, features,
and general techniques have been demonstrated. Here is a
brief summary.

Breakpoints

n↑B	Set a breakpoint at location n
↑B	List the beakpoints
n↑X	Delete breakpoint number n
↑X	Delete all breakpoints
n↑N	Open the breakpoint count register for breakpoint #n
n↑K	Open the trip-count register for breakpoint #n
↑P	Proceed from breakpoint
n↑P	Proceed from breakpoint and set trip-count for this breakpoint to n.

When a breakpoint is encountered, control comes to ZBUG. The trip-count for the breakpoint is decremented and, if zero, is reset to the value in the N register for the particular breakpoint. The breakpoint number and address are then reported to the user. If an RST 38 instruction or 'BREAK' interrupt is encountered, control goes to ZBUG which prints the message '??B' and the address at which the break occurred.

Stepping

↑S	Step one instruction
n↑S	Step n instructions

After a stepping operation is completed, control returns to ZBUG which prints the message 'S' and the address of the next instruction to be executed.

Environment

↑L	Open the local environment register. Input is a module name (no leading ESC)
↑E	Open the environment register. Input is a program name (no leading ESC)
↑D	Open the displacement register

When the ZBUG D register is nonzero, addresses are printed in symbolic form if an appropriate symbol is accessible, in relative form if the displacement is non-negative, and in hexadecimal otherwise.

Starting a Program

n↑G	Begin execution at location n

Symbols

Symbol names may be used freely in expressions. The symbol must be either global or local in the module specified by the L register to be accessible. Symbols from the assembler must be no more than six characters. The character '.' may not be used in a symbol except in a program name (E command) or module name used in a response to the ↑L command.

Using the Last Value Printed by ZBUG

The character '%' has the value of the last number output by ZBUG.

Dumping Blocks of Memory

first,last.	will dump locations first to last in HEX8 and ASCII modes.

The ZBUG Reference Manual gives a complete but terse description of all the ZBUG commands. The quick reference sheet, one of the appendices of the reference manual, lists all the commands and other information useful to have beside you when debugging at the terminal.

PART 2: ZBUG REFERENCE MANUAL

I. CONVENTIONS

↑<character>	means control (CRTL) <character>
$	means the escape key (ESC) unless otherwise noted
CR	means the "return" key
LF	means the "line feed" key
ESC	means the escape (ESC) key
DEL	means the DEL or RUBOUT key
*	ZBUG's prompt character (precedes most examples)

II. ZBUG GENERATION, ENTRY AND EXIT

Unlike the PROM debugger, ZBUG must be loaded into memory explicitly in order to be used. This may be done either by linking ZBUG directly with your program or by generating a procedure file in a specific area in memory and loading it with your program at the RIO command level. The relocatable version of the ZBUGger is called ZBUG.OBJ and is referenced in the LINK command as ZBUG. To produce a procedure file version, a command such as

```
%LINK  $=7000  ZBUG  (N=ZBUG70 NOM  ST=0)
```

can be given. (The "$" is the real $.) This example produces a file called ZBUG70, containing the ZBUGger which can be loaded with your program by

```
%your.prog,ZBUG70  <optional parameter list
                    for your program>
```

Control goes to the ZBUGger following this load. Note the convention of including the address of the debugger in the file name (i.e., ZBUG70 implies starting address 7000).

The ZBUGger can be manually started at its first word address (7000 above). Once a user program in which breakpoints have been placed has been started, control comes to ZBUG if a breakpoint is encountered. Control will then also come to ZBUG if the NMI (BREAK) button on the console is pressed.

Exit

Control can be returned to the RIO command interpreter by issuing the ↑Q command:

```
*↑Q
%       (control has returned to RIO)
```

All breakpoints are removed.

User Symbols

In order to have ZBUG know about the labels in your assembler program, the options to produce a binary symbol table file must be selected at assembly and link time. An example illustrates:

```
%ASM MOD1 (S)
%ASM MOD2 (S)
%LINK $=4400 MOD1 MOD2 (SY)
```

The S options on the ASM commands cause the assembler to append the symbols to the binary file so that the linker can combine them into a binary symbol table file. The SY option on the LINK command causes said file to be created (with extension .SYM). This file name can then be input to ZBUG in the E (Environment) register (See IV).

III. ERRORS AND DELETING COMMANDS

The error messages are:

??	something is wrong
OVF??	a number was out of range (generally too big for context)
DISK ERROR xx	the specified error occurred while trying something with symbol files
UND??	a symbol given is undefined.

Correcting Errors

THERE IS NO BACKSPACE CHARACTER

Mistakes made while typing numbers can sometimes be corrected. Only the rightmost four digits are accepted, so typing several zeros and retyping the number may work. Also, if an incorrect number is typed in an expression it can sometimes be later subtracted and the correct number added.

Pressing DEL generally gets you out of anything without modifying register contents or taking other actions.

IV. EXPRESSIONS, SYMBOLS AND DISPLACEMENTS

Many inputs to ZBUG are expressions. Any expression may consist of the elements described below. Several different modes of input are accepted as elements in expressions. These may be combined using one of several operators.

Elements in Expressions

Each element has a 16 bit numeric value. Whether the value is treated as 16 bit or not is dependent on context. In computing expressions, however, 16 bit arithmetic is used.

The legal elements are:

<hex number>	The rightmost four digits of the number typed are used. Upper or lower case characters for A-F are accepted.
<hex number>'	The rightmost four digits of the number typed are added to the contents of the D register, and this value is used. This form is useful for specifying addresses in relocatable modules by setting the D register (see below) to the module origin and then inputting the addresses on the listing with the "'" sign to form the correct absolute address.
$<symbol>	The <symbol> is looked up in the ZBUG symbol table and the corresponding value is used in the symbol's place. See below for a description of how to gain access to your program's symbols.
'<character>	The value of the ASCII code for <character> is used.
$ (real $)	The location of the last memory location opened is used.
%	The value of the last register opened or the last expression evaluated with the "=" command is used.

Elements may be preceded by a unary "+" or "-" sign and combined with the operators "+", "-", "*" (multiply), and "/" (divide). Expressions are evaluated left to right with no operator precedence.

Evaluating an Expression

The "=" command can be used to output the value of an expression.

```
*n=
```

where n is an expression whose 16 bit hex value is output.

Loading the Symbol Table

Assuming that a binary symbol table has been produced as described in Section II, the ↑E and ↑L commands can be used to load the ZBUG environment.

The symbol table is loaded immediately following ZBUG in memory. Hence, IT IS A BAD IDEA TO HAVE CODE OR DATA FOLLOWING ZBUG IF YOU PLAN TO USE THE SYMBOL TABLE COMMANDS. Also, no check is made to prevent the symbol table from running past the end of physical memory. The ↑W command reports on the bounds of ZBUG and the current symbol table.

ZBUG does not interact with the RIO (Rev. F and later) memory manager. Manual allocation of space for the symbol table is advised, (and usually necessary).

The binary symbol file is made known to ZBUG by issuing the ↑E (ENVIRONMENT) command. ↑E types the name of the current symbol file and accepts the name of a new one, if desired. The name must be entered WITHOUT the .SYM extension.

```
*↑E   BASIC
*↑E   BASIC  NEWPROG
*
```

In this example, first the symbol file BASIC.SYM is specified, then the file NEWPROG.SYM is selected. The global symbols and module names are loaded into the ZBUG symbol table upon specification of this command. The local symbol portion is initialized to the symbols from the module of the same name as the symbol file, if any. If no such module exists, then a question mark is generated. The globals and module names, however, are always loaded.

Local symbols in a module can be loaded by specifying the module name in response to the ↑L (LOCAL) command's prompt:

```
*↑L  INFORM
*↑L  INFORM  SCANNER
```

Here the module INFORM has its locals loaded first and then the module SCANNER has its locals loaded, overwriting the previous set of locals. You can thus have locals of only one module active at a time.

NOTE: ZBUG uses RIO unit 20 to load the symbol table. Therefore, user programs should avoid that unit.

The bounds on the symbol table and ZBUG are reported by the ↑W command.

```
*↑W  7000  8323  85F3
```

Here ZBUG occupies locations 7000 to 8323 and the current symbol table (including any locals loaded) occupies locations 8323 to 85F3. Great care should be taken to prevent the symbol table from overwriting anything or running past the end of memory.

Reserving Symbol Table Space

Space for a symbol table immediately following ZBUG can be reserved as follows:

1. Link a version of ZBUG as described above. (Here, we will assume it was called ZBUG70.)

2. Use the RIO command EXTRACT to find the highest address used. Round this number up to the next 80H byte boundary.

3. The size of the symbol table can be guessed very roughly at 1K for each 20 pages of source. Allocate sufficient memory starting at the address calculated above. Unless memory space is very scarce, it doesn't hurt to overestimate; once the symbol table is loaded, the ↑W command can be used check that sufficient space was allocated.

For example:

```
%EXTRACT ZBUG70
...........
 LOW ADDRESS = 7000  HIGH ADDRESS = 83A2
...........
%ALLOCATE 8400 A400 2000
```

This reserves 8K of space.

The Displacement Register

The D register is used for two purposes:

(1) To supply a basis for numbers entered as relative (i.e., with the "'" suffix), and

(2) to supply an origin from which addresses output by ZBUG will be offset.

When an address is output and the D register is nonzero, a symbol table search is performed to find a symbol with the value of the address to be output. If found, the symbol name is output; otherwise the address is output as a 16 bit hex number if it is less than the value in the D register, and in "displaced" mode if it is not. Thus, relative addresses are never output as negative numbers and setting the D register to -1 will force the symbol table search but never output in relative hex mode.

To set the D register, the command ↑D is used:

```
*↑D  0000
*↑D  0000   $MODB
```

Here the D register is opened by typing ↑D but not modified (CR is pressed). Next, it is again opened and the value of the symbol MODB (probably a module name) is placed in it. A more complete discussion of opening registers is given in Sections V and VI.

V. MEMORY COMMANDS

Memory and registers can be displayed in one of several output modes. They are:

HEX8	8 bit hex
HEX16	16 bit hex
DHEX16	16 bit hex displaced from the D register as described in the previous section
ASCII	as a 7 bit ASCII character
QUIET	no output

ZBUG maintains a "current" output mode which is set as the most recently specified of one of the above. The output mode may be explicitly specified by issuing one of the following commands:

*.	HEX8 mode
*:	HEX8 mode
*[	DHEX16 mode
*(	ASCII mode
*!	QUIET mode (no output)

These characters are also used in conjunction with one or two parameters: to open a specified location or to dump a range of locations, respectively.

Opening a register is analogous to opening a box: you can examine and/or modify the contents when the box is open, and you cannot when the box is closed. When a memory location is opened, the contents are displayed in the mode selected by the command that opened it; or in the current mode, if the command that caused the location to be opened selected no mode. Then an expression to replace the contents of the location can be optionally input followed by one of:

CR	to close the location (replacing the contents if new contents were input)
LF	to close the location as in CR but then open the next location
↑	to close the location as in CR but then open the previous location
DEL	to close the location immediately with no alteration
.	to redisplay the location in HEX8 mode
:	to redisplay the location in HEX16 mode
(	to redisplay the location in ASCII mode
[	to redisplay the location in DHEX16 mode

The contents of the location are never changed if an error (message ??) occurs.

Memory locations are opened in one of the above modes with the command:

```
*nc
```

where c is one of ., :, (, !, or [.

LF and ↑ issued as single commands open the next or last location in the ZBUG current mode (next or last from whatever location was last open).

Dumping Memory

A range of locations can be dumped by issuing one of the following commands:

```
*n,m.
*n,m(
*n,m:
```

n,m. and n,m(produce dumps in HEX8 and ASCII modes (combined) of locations n through m. n,m: produces a dump of locations n to m output in HEX16 mode. The dump can be interrupted by pressing any key.

VI. BREAKPOINTS, CPU REGISTERS, AND STEPPING

The general strategy of ZBUG is to insert itself between two instructions so that, between these instructions, registers and memory can be examined and/or modified and an evaluation made of whether or not the program is executing properly.

ZBUG allows this kind of debugging by providing features allowing the placement of up to 8 breakpoints in the user program, by making it possible to step through the program one or several instructions at a time, and by saving and restoring the machine state on entry and exit from ZBUG.

Registers

Any time ZBUG is entered it saves the contents of all registers. These values are available for inspection and modification. The registers can be displayed or opened in a manner similar to memory locations.

The ↑R command causes most registers to be displayed:

```
*↑R
```

Individual registers can be opened by specifying the register name followed by the ↑R command:

```
*$B↑R   04
```

Here, register B is opened and the value displayed (04) in HEX8 mode. Once a register is open, an expression to replace the value can be optionally entered followed by CR to close the register.

Only the low 8 bits of a value input to an 8 bit register are used. The register names are:

```
$A  $B  $C  $D  $E  $H  $L  $F  $A'  $B'

$C'  $D'  $E'  $H'  $L'  $F'  $SP  $IX  $IY  $PC  $I
```

where $SP is the stack pointer register and $PC is the program counter.

The interrupt vector can be similarly examined and modified but is not displayed by the ↑R command with no arguments. It's name is $I.

Stepping

One or more instructions (beginning at PC) can be executed (with control then returning to ZBUG) by issuing the ↑S command (STEP).

There are 2 forms:

```
*↑S      single step
*n↑S     step through the next n instructions
         (n an expression)
```

After the specified number of instructions have been executed, control returns to ZBUG. The contents of the registers are optionally displayed (see below).

Breakpoints

Breakpoints are placed on the first byte of an instruction which meets the restrictions listed below. When this instruction is executed, control goes to ZBUG which may return control to the user program after reporting the break and address.

To provide flexibility when using breakpoints in loop-like structures, there is a trip count associated with each breakpoint. Each time the breakpoint is encountered, the trip count is decremented and if the value is nonzero, control returns to the user program. When the count reaches zero, ZBUG reports the breakpoint.

The addresses of the breakpoints are kept in the B registers, the trip counter value in the N registers, and the trip countdown (the value that gets modified) in the K registers.

Each entire group of these registers can be displayed with:

```
*↑B      display breakpoint address registers
*↑N      display trip count registers
*↑K      display trip countdown registers
```

All are output in HEX16 mode.

To set a breakpoint at location n, the command

```
*n↑B
```

is issued. An error results if this would be the 9th concurrent breakpoint. Each breakpoint is assigned a number (as displayed by the ↑B command) and this number is used by the debugger to report the occurrence of a breakpoint and by the user to delete a particular breakpoint.

*↑X	deletes all breakpoints
*n↑X	deletes breakpoint n (n = 0 - 7)

The breakpoint count and countdown registers may be opened (and hence altered) by giving the commands:

*n↑N	open N register for breakpoint number n
*n↑K	open K register for breakpoint number n

Controlling Execution

The user program can be started (or continued) in one of several ways:

(1) Starting at a particular address

*n↑G GO (execution begins at address n)

(2) Starting at the current value of PC

*↑G GO (execution begins at the current PC value)

(3) Proceeds from a breakpoint

*↑P PROCEED

(4) Proceed from a breakpoint and set the trip countdown (all at once---i.e., the K register)

*n↑P PROCEED, set trip countdown for last BP

Controlling Register Display

Normally, when control returns to ZBUG following a step or breakpoint, only the address of the next instruction to be executed is displayed. It is sometimes desirable to display the CPU registers at this time. A one-byte register, $RSWITCH, controls this display option. The value one causes registers to be displayed and zero suppresses the display.

```
*$RSWITCH. 00 1
```

Here, the display is enabled.

Restrictions

Breakpoints may not be placed on:

1) any but the first byte of an instruction;

2) any instruction that is modified;

3) any instruction that is also used as data;

4) any instruction within ZBUG;

5) any location in non-modifiable memory (PROM, ROM, etc.);

6) any location that follows a non-modifiable location in memory (ROM, PROM, etc.);

7) at location FFFF; or,

8) any instruction that fails to satisfy the step restrictions below as the instruction at the location of the breakpoint must be stepped through.

In addition, anomolous results will be obtained if the instruction on which the breakpoint is placed references the immediately preceding location in memory. This is because the instruction preceding the breakpoint is altered when the instruction at the breakpointed location is executed and restored after that instruction is executed.

The stepping operations cannot be used if:

1) The location preceding the instruction to be stepped is in non-modifiable memory (ROM, PROM, nonexistent memory, etc.)

2) The instruction to be stepped through references the preceding location as data

3) The instruction to be stepped through is an:

```
IM      0
IM      1
LD      I,A      with A≠13H
```

(The idea here is that an interrupt is going to occur at the end of the instruction, and if the interrupt environment is faulty, the state of ZBUG will be likewise).

Also, if a

DI

instruction is stepped through, control will not return to ZBUG until one instruction after an EI is executed.

If an

EI

is stepped through, the instruction following it will also be executed before control returns to ZBUG.

VII. SEARCHING AND FILLING MEMORY

Searching

ZBUG provides a facility to search for particular bit patterns in memory; up to a four byte value may be searched for. The search proceeds as follows:

Each location in the specified range is tested by loading the four bytes beginning at the current location. These bytes are 'and'ed with the four byte Mask register and then compared to the four byte Word register. If there is a match, the location and contents are output in the current output mode. Then the process is repeated for the four bytes beginning at the next location.

One, two, three or four byte instructions may be searched for using this feature, as can a two byte address, for example.

To set the Mask and Word registers, locations accessed using the symbols $MASK and $WORD are opened. Input is as any other memory location but take care not to modify any but the four bytes beginning at these symbols as the locations are within ZBUG.

The search is initiated with the command:

```
*n,m↑S
```

which causes locations n to m to be searched as described. For example, suppose we wish to search for a HALT (76) instruction followed by a 1.

```
*$MASK!   -1  LF
xxxx      -1  LF
xxxx       0  LF
xxxx       0  CR
*$WORD!   76  LF
xxxx       1  LF
xxxx       0  LF
xxxx       0  CR
*4000,5000↑S            does said search on locations
                        4000 to 5000
```

Filling Memory

A series of memory locations can be set to a value as follows:

```
*n,m↑Z          sets locations n to m to zero
*n,m,k↑F        fills locations n to m with k
```

ZBUG cannot be overwritten with these commands.

Example:

```
*4000,5000,'X↑F    fills locations 4000 to 5000 with
                   the ASCII character 'X'
```

VIII. INTERRUPTS

Due to the complications noted below, interrupt code debugging is complex and somewhat ill-advised. The following commands are provided to monitor and control the interrupt system.

Interrupt Flip-Flop

The ↑I command opens the Interrupt flip-flop (IFF) register. The values zero and one indicate interrupts disabled and enabled, respectively.

When ZBUG receives control (through a breakpoint or by entry to its first word address) the IFF register is saved. When control returns to the user program (by use of the ↑P or ↑G commands) the hardware IFF is set to the value in the ↑I register.

It should be carefully noted, however, that ZBUG itself enables interrupts while it is executing and disables and re-enables them during step operations (↑S command). Also, ZDOS requires that interrupts be enabled in order to access the disk, (which happens while loading the symbol file).

Interrupt Vector Register

The interrupt vector register can be examined and modified as the other hardware register by being opened, modified, and closed. It has the name $I (see example below). Once again, note that for proper ZDOS operation and for ZBUG stepping the I register must have value 13H. Upon entry to ZBUG the hardware I register is saved and then set to 13H. When control returns to the user program (↑P or ↑G commands) the hardware I register is restored to its value on entry (or the value explicitly set).

Interrupt Mode

The interrupt mode can be set by

```
*n↑I
```

where n=0,1, or 2. The mode change takes effect immediately. ZBUG, however, changes the mode to 2 to do stepping. Thus, setting the mode would be cancelled if a subsequent ↑S command is issued.

IX. Rough Spots and Their Conquest

There are some idosyncrosies associated with the use of ZBUG. Some could certainly be considered "bugs", but, in any case, here is a list of them and, when possible, how to overcome them.

1. Use of module names which include the character "." causes problems in that "." is a ZBUG command. Thus, such names cannot be used in any context other than a response to the ↑E (environment filename) or ↑L (local environment module name) commands.

   solution: 1. Don't use "." in module names.
   2. Have another global symbol in such a module with value equal to the first byte address in the module and use it instead of the module name.

2. A symbol table search must match all characters typed to succeed. Recall that the assembler truncates names to 6 characters. Thus, a symbol in a program:

   SEARCHTBL: .....

   must be referenced as $SEARCH. Module names, however, are not truncated.

3. The response to an ↑E or ↑L command must not be preceeded by an escape ($).

4. The area following ZBUG where the symbol table is loaded is not allocated using RIO memory management and, hence, must be manually allocated if ZBUG is to be used with a program that allocates memory through RIO. Also, ZBUG does not check if the memory required for symbol tables is already allocated.

   Solution: 1. Manually allocated space for the symbol table prior to loading ZBUG and the program to be debugged. (Use the RIO ALLOCATE command).
   2. Instead of linking ZBUG with ST=0, link with ST=n where n is large enough for both the symbol table and stack of the program being debugged. Be sure that this stack gets allocated immediately above ZBUG. Example:

   LINK $=9000 ZBUG (NOM ST=1400 N=NBUG90)

   TO.BE.DEBUGGED.PROGRAM,ZBUG90

   3. Be very careful.

Part 3: NBUG

NBUG is an extension of ZBUG that incorporates an assembler/disassembler allowing display and entry of Z80 instruction mnemonics. It is approximately 2.5K bigger than ZBUG. Except as noted below, it functions identically to ZBUG.

Instruction Mnemonic Mode

The instruction output format is selected by the ; command. The ; can be used to open a memory location or redisplay a value the same as ., :, (, and [. If the value to be displayed is not legal instruction, it is displayed in HEX8 mode. LF advances the location counter past the instruction displayed. ↑ decrements the location by 1 byte (regardless of the intruction size).

Once a memory location is open, a Z80 instruction can be entered. The number of bytes written upon as well as the number of bytes the location counter is advanced when the LF command is issued depends on the length of the instruction. Instructions can be entered regardless of the output format used to display a location.

Notes on Instruction Assembly

Several differences between NBUG's assembler and the RIO assembler exist. They are listed below.

1) Blanks, as well as commas, are accepted as field separaters.

2) All numbers are assumed to be hexadecimal.

3) Numbers do not have to begin with a digit, however they will be interpreted as a resigter name if such an interpretation is possible. For example,

```
    LD  B A            load register B with register A
    LD  B 0A           load register B with A (hex)
```

4) IM0, IM1, IM2 must be entered without spaces.

5) Any user symbols used must be prefixed with ESC ($), consistent with ZBUG symbol use.

Backspace Is Here

The backspace (control-H) and DEL (RUBOUT) keys function as they do under RIO. A command character still terminates input, so errors cannot be corrected once the command is issued. The backslash ('\') serves the 'abort' function formerly served by DEL (RUBOUT).

Breakpoint Register List

Breakpoint addresses listed by the ↑B command are displayed symbolically rather than as absolute HEX addresses.

Linking Instructions

A command file called NBUG.LINK.CMD is provided. It accepts one or two parameters:

DO NBUG.LINK.CMD addr [stack_size]

where addr is the high two digits of the address for NBUG to run (the low order digits are zero) and stack_size is an optional stack to be allocated when NBUG is loaded. NBUG does not use this stack; it is for user programs if needed.

Example: To make a version of NBUG that runs at 7000 (hex) enter

%DO NBUG.LINK.CMD 70

This will produce a procedure file called NBUG70.

Sorry About This

Life is, of course, not a bed of roses.

1) Some illegal instructions are assembled and disassembled without complaint. The set roughly includes:

   ### Assembly

   Usage of IX/IY in 2 fields (e.g., LD (IX) (IX))
   Usage of IX/IY with some instructions for which such usage is not legal.

   ### Disassembly

   Instructions that begin as IX/IY instructions but don't use IX or IY (extra DD or FD prefix)

APPENDIX - ZBUG Quick Reference Sheet

Conventions:	↑<chr>	means a control character
	n m k	are expressions (see below)
	$	is ESC unless otherwise noted

Errors: OVF?? means a number is too big for context. UND?? means an undefined symbol. ?? means "I can't do that."

RUBOUT or DEL ('\' in NBUG) gets you out of most everything without modifying anything.

Zero Argument Commands

↑B	List breakpoints
↑D	Open displacement register
↑E	Open Symbol File name register
↑G	Go at address in PC (exits debugger to user program)
↑I	Open interrupt flag register
↑K	List breakpoint countdowns
↑L	Open the local symbols module name register
↑N	List breakpoint count registers
↑P	Proceed from breakpoint
↑Q	Quit. Return to RIO.
↑R	List registers
↑S	Step. Execute one instruction.
↑W	Where? List debugger beginning, end and symbol table end addresses.
↑X	Delete all breakpoints
↑Z	Same as ↑P for people with small hands.

!	Set QUIET output mode
.	Set HEX8 output mode
:	Set HEX16 output mode (16 bit hex)
(	Set ASCII output mode
[	Set Displaced HEX16 mode (16 bit hex offset from D register)
;	Set INSTRUCTION output format (NBUG only)

LF	Open next memory location (after whatever the last one was)
↑	Open the previous memory location

One Argument Commands

n↑B	Set breakpoint at location n
n↑G	Begin execution at location n
n↑I	Set interrupt mode n (n=0,1,2)
n↑K	Open breakpoint countdown register n (n=0-7)
n↑N	Open breakpoint count register n (n=0-7)
n↑P	Proceed from breakpoint and set BP countdown register to n
n↑R	Open register n (n=0-20. or $A $B ... $A' $F' $IX $PC $SP $I)
n↑S	Execute the next n instructions
n↑X	Delete breakpoint number n (n=0-7)

n!	Open memory location n with no output
n.	Open memory location n in HEX8 mode (8 bit hex)
n:	Open memory location n in HEX16 mode
n(	Open memory location n in ASCII output mode
n[	Open memory location n in displaced HEX16 mode
n;	Open memory location n in INSTUCTION mode (NBUG only)
n=	Type n in HEX16 mode (evaluates expression)

Two Argument Commands

n,m↑S	Search memory from n to m
n,m↑Z	Zero memory from n to m inclusive
n,m.	Dump memory in HEX8 mode and ASCII from n to m
n,m(	Same as n,m.
n,m:	Dump memory in HEX16 mode from n to m

Three Argument Commands

n,m,k↑F Fill memory from n to m with k

Expressions for Input

Expressions are evaluated from left to right and may include the operators +, -, *, and /. There may be a leading + or - sign on each element. The elements in the expression may be:

<hex number>	The last 4 digits are significant
<hex number>'	The number offset by the D register
'<character>	The value is the specified 7 bit ASCII character
$<symbol>	The value of the symbol table entry for the symbol is used ($ is ESC).
$	The value is the address of last location examined ($ is $, not ESC).
%	The value is the contents of the last location or register opened or the last expression evaluated by =

Once a Memory Location is Open

An expression replacing the contents of the location may be typed. Following the expression (if any), a CR closes the location, LF closes the location and opens the next, ↑ closes the location and opens the last.

If no expression is typed, the value may be redisplayed in another output mode by typing a zero-argument command listed above.

Hardware and ZBUG registers

Once opened, an expression may be typed which replaces the previous contents followed by CR or the register may be closed unmodified by typing only CR.

$MASK is the origin of the 4 byte mask register. $WORD is the origin of the 4 byte word register. $RSWITCH is the one byte register whose value determines whether the registers will be displayed after steps and breakpoints.