# ZBUG

ZBUG enables you to trace through any Z80 machine language program in a controlled manner. You can initiate the trace in two ways:-

1) Branch to ZBUG, then branch to the program start address using the ZBUG JUMP command. If ZBUG is loaded at address A, and your program starts at address abcd (in hexadecimal), then you type

```
SYSTEM (enter)
*?  / A (enter)     (A is in decimal)
ZBUG J abcd (enter)   (abcd is in hexadecimal)
```

2) Call ZBUG from your running program at the point at which you want tracing to start. ZBUG will reset your call stack and commence tracing from the instruction following the call to ZBUG.

## Instruction Trace

ZBUG is set up by default to trace every instruction. For each Z80 instruction it displays a line of information on the screen and then pauses. If you hit any key other than 'Z' it will continue to the next instruction and will display the next line automatically.

The information displayed at each trace point is:

```
LL iiiiii oooo-> nnnn ssss hhll ddee bbcc aaff xxxx yyyy
```

where all values, except LL, are in hex.

LL is an optional mnemonic character pair, a'label', which identifies the trace point. (See the STOP command).

| | |
|---|---|
| iiiiii | is the last executed instruction, (2 to 8 hex digits). |
| oooo | is the old program counter, before execution of the instruction. |
| nnnn | is the new program counter, after execution of the instruction. |
| ssss | is the value of the stack pointer, SP. |
| hhll | is the value of HL. |
| ddee | is the value of DE. |
| bbcc | is the value of BC. |
| aaff | is the value of AF (A is the accumulator, F is the flags). |
| xxxx | is the value of index register IX. |
| yyyy | is the value of index register IY. |

## Selective Trace

ZBUG contains a print selector at byte A + 2, where A is the address at which ZBUG is loaded. The bits in the print selector are interpreted as follows:

| | |
|---|---|
| X '80' | Trace every instruction. |
| X '01' | Trace successful CALL s , RETURN s and RESTART s. |
| X '02' | Trace computed branches (E9, DDE9, FDE9) |
| X '04' | Trace successful branch-to-address instructions. |
| X '08' | Trace successful relative branches. |
| X '10' | Trace each execution of the important storage access instructions , 22, 2A, 32 and 3A. |

Any combination of bits is permissable, but X '80' subsumes all the others.

## Mnemonic Address Stop

Using the stop command (see later) you can build up a table of addresses at which you want execution to pause. If you set the print selector to X '00' then these will be the only trace points displayed. The trace is controlled by a table which starts at A+ 047C (hex). Each table entry is 4 bytes long, being the 2-byte address at which you want execution to pause, and a 2-byte character-pair label, which is displayed at the left of the trace line, to identify the point. The table is at the end of ZBUG and can be as long as you like, provided you leave the necessary free memory. The table end is recognised as a 'zero' address. The trace point must be on an instruction boundary, and the trace display
shows the machine state after execution of that instruction.

## Trace Suspension

The preceding options have described how you can control the amount of trace information displayed. You can also suspend tracing altogether, across well-behaved CALLS. The called routine will then run at full speed, without interference from ZBUG, until it returns. The return point will be in ZBUG itself, so the called routine must not use the stack return point improperly.

a) Byte A + 3 (where A is the starting address of ZBUG) is the invocation depth limit , initially set to 128.
If you set this to n , then ZBUG will only trace to depth n , i.e. the top n+1 levels of your program. All the time the depth of nesting of calls excedes n, tracing will be suspended. The untraced call is marked on the display with '*'.

b) Bytes A + 4 and A + 5 contain the interpretation address limit, initially set to 4200 hex. ZBUG suspends tracing on any calls below this address. If you dont modify this address, then its effect is to inhibit tracing of any calls your program may make to TRS ROM routines. This is normally what you want. If you reduce the value of this address, or set it to zero , then ZBUG will trace your calls to ROM, and you can, if you wish, single-step through ROM to see how the BASIC routines work. (However if you trace the input/output routines at 0033, 002B, or 0361, then your traced program will not work, since ZBUG itself uses these routines).

## ZBUG Commands

Program flow pauses at each trace point, and it will continue automatically to the next trace point if you hit any key other than 'Z'. Hitting 'Z' takes you into ZBUG command mode. The word 'ZBUG' is displayed and ZBUG waits for your input. This is also the default mode, when you first invoke ZBUG. You can exit from command mode and continue tracing by hitting the ENTER key. The commands are as follows, where abcd is taken to represent a hexadecimal address. If you key in an error, ZBUG displays a ? and gives you a second attempt.

**H abcd** Hexadecimal display of memory. 256 bytes of memory are displayed on the screen, starting at address abcd.
The presentation is 16 'Lines where each line is
1. Hex address on the left.
2. 16 bytes of memory displayed as 4 blocks of 8 hexadecimal digits.
3. The same 16 bytes of memory displayed as 16 characters. (Characters are folded to the range 0-128, and then non-displayable characters are shown as decimal points).

The display fills the screen. You can now type:
C (Continue). Display the next 256 bytes of memory
B (Back). Display the preceding 256 bytes of memory
Enter key. Return to ZBUG command mode.

**M abcd** Memory display / modify. Memory is displayed, in hex, a byte at a time. You can type in :-
1. Enter key. Proceed to next memory byte without modification.
2. Valid hex digit pair. This byte replaces the displayed byte in memory.
3. X. This cancels memory display/modify and returns to ZBUG command mode.

You can use this command to modify the control bytes (print selector, etc.) within ZBUG, or you can modify your program's current registers by changing their saved values in the ZBUG stack.

**J abcd** Jump to address abcd. This changes tree flow of the interpreted program, and causes interpretation to continue from abcd.

**R** Return from ZBUG. Interpretation will cease altogether. Your interpreted-program will continue from its current point without interference. Valid only if ZBUG initiated by CALL entry.

**S LL abcd** Stop at address abcd. This adds another entry to the address-stop table which starts at location A + 047C (hex). If subsequent program flow executes the instruction at abcd, then a trace line will be displayed, with mnemonic label LL at the left, and program flow will pause.

Note that each use of the STOP command adds 4 bytes to the end of ZBUG. So you must leave sufficient space at the back of ZBUG when you load it, for whatever table size you need.

## Layout of ZBUG

The first few bytes of ZBUG contain control values, as described earlier. The addresses down the left-hand side are in hex and are relative to A, the starting address of ZBUG.

| Addr. | length | | |
|---|---|---|---|
| 0000 | 2 | X '1842' | Branch to real start point |
| O002 | 1 | X '80' | Print Selector |
| 0003 | 1 | X '80' | Invocation Depth Limit |
| 0004 | 2 | X '0042' | Interpretation Address Limit ( = 4200 ) |
| 0006 | 2 | X '7C04' + A | Address of address-stop table |
| 0008 | 34 | | ZBUG stack |
| 002A | 2 | AF | Saved register. (can be modified) |
| 002C | 2 | BC | |
| 002E | 2 | DE | |
| 0030 | 2 | HL | |
| 0032 | 2 | SP | |

## How to Load and Relocate a Southern Software Machine-Language Program.

You choose the location of the program in memory, to suit your machine size. This MUST be in protected memory, or the program will not run. So, taking account of your machine size, allow enough space for the program itself, plus any other machine-language subroutines you may need, either above or below the program you are loading.

As an example, suppose you are loading Southern Software DLOAD (size 160 bytes). You have already loaded, or are going to load, TRS KBFIX at the top of memory, and Southern Software TSAVE below DLOAD. Plan your memory use as follows, working out the values (T) and (A) for your situation:

| | PROG SIZE (bytes) | MACHINE SIZE 4K | 16K | 32K | 48K |
|---|---|---|---|---|---|
| Memory limit | | 20480 | 32768 | 49152 | 65536 |
| Space for KBFIX | 56 | 20424 | 32712 | 49096 | 65480 |
| Space for DLOAD | 160 | 20264 | 32552 | 48936 | 65320 (T) |
| Space for TSAVE | 512 | 19752 | 32040 | 48429 | 64808 (A) |

1)Turn on the computer. If you have a DISK system, enter Level2, not DISK BASIC.
2)answer the MEMORY SIZE question with your value of (A). (On Video Genie, this value is used after READY?).
3)prepare the cassette player to load the self-relocating program.

| | TRS-80 | You type: |
|---|---|---|
| 4) | > | SYSTEM (enter) |
| 5) | *? | DLOAD (enter) or your program name |
| 6) After tape has loaded | *? | / (enter) |
| 7) | TARGET ADDR? | Your value of (T) |
| 8) | READY | |

Notes:
1)At step 5 the tape will load and a pair of asterisks will blink on the display. If there are no asterisks, or two unblinking asterisks, or C*, then there has been a loading error. Stop the recorder, reset, and retry with a new volume setting.

2)At step 7, the program will relocate itself to address T. If instead of typing a value you just hit enter, then the program will relate itself to A, the answer to the MEMORY SIZE question.

3)Under Level2, after relocation, the program is ready to be invoked with a USR(n) call, since the USR address is automatically primed. However this does not work under DISK BASIC (or Level3), and you must additionally set DEFUSRn to inform the system of this routine's address.

4)Once a program has been loaded and relocated, it can be dumped to a new tape using Southern Software TSAVE, or TRS TBUG. Then it will load directly to its final location. Use of TSAVE has the advantage that several programs can be dumped on a single file, which can also preprime the USR address.

5)During step 5, the program is temporarily load into locations 18944 and up. This means that
a)You must perform all necessary relocating loads before loading a BASIC program, or entering DISK BASIC.
b)The final location, T, of the self-relocating program can never be lower than 18960. (Hex 4A10).

6)If you run under DISK BASIC, then perform the initial self-relocating load under Level2, as described. Then reenter TRSDOS (or NEWDOS, etc) and use the DUMP command to save the core image directly from its relocated position. Subsequently you can LOAD the core image directly, under TRSDOS. But when you enter DISK BASIC, remember to set MEMORY SIZE to leave this area of core protected, and remember that the top 64 bytes of memory are corrupted by the DISK BASIC loader, and should not be used for programs.

## Hints on Tape Loading.

1) Listen to the tape to establish exactly where the data starts. Note this on the tape label.
2) Turn the volume down to zero, and "attempt" a tape load, very slowly increasing the volume until you get asterisks on the screen. Stop the tape (not the computer), note the volume level, Reboot.
3) Turn the volume up to maximum, and "attempt" a tape load, very slowly decreasing the volume until you get asterisks on the screen. Again, stop the tape, and note the vole,
4) Set the volume to slightly above the mid-point of the two extremes of volume, and attempt a real load.

## Possible Tape or Recorder Faults.

1) Kink or fold in the tape. Even a minor fold may render the tape unloadable, (Southern Software tapes carry a second copy of the file, in case the first gets damaged).
2) Noise caused by RESET when tape is running, Always stop the tape before hitting RESET.
3) Being small, all the plugs are prone to intermittent error and should be protected against movement.
4) Inconsistent tracking of the tape over the head.

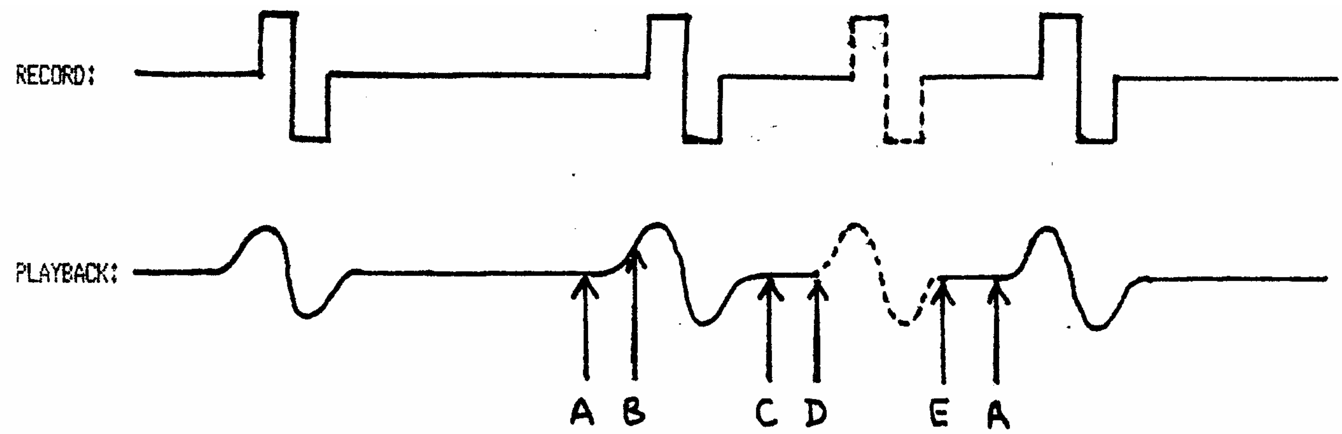
This list does not include poor tape quality, since it is very unlikely to be a problem, at the frequency bits are recorded. However, you may have found that one make of cassette seems much better than another. This is probably due to the construction of the cassette, rather than the tape. Generally more expensive cassettes run more smoothly, and therefore reduce the chance of poor tracking of tape over the head.

## How DATA is Recorded and Read.

The computer contains hardware to generate an "above-and-below-zero" pulse, as shown below. This is fired by direct program control. The output routine produces one such clock pulse every 500th of a second (by looping). Data ones and zeroes are recorded as pulses halfway between these clock signals, A zero is the absence of a pulse, a one is the presence of a pulse.

The playback logic is analogous to a keyboard "debounce" routine, To read a single bit, start somewhere near (A). Loop, until the hardware recognises a signal, at (B). This is a clock pulse. Now loop until that signal is bound to have died away, and reset the hardware latch, at (C). Now wait an exact length of time, till (D), and listen for another signal, YES, then it's a one, NO, then it's a zero. In either case reset the latch after the sampling time, at (E), and loop again until the next time (A).

As you can see, the TIMER must not be running during either record or playback, since exact looping times are vital. Nor does the logic take time off to test the keyboard for the BREAK key. However tape speed is not ultra-critical, since there is a resynchronisation wires at (A) on every bit.




southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ