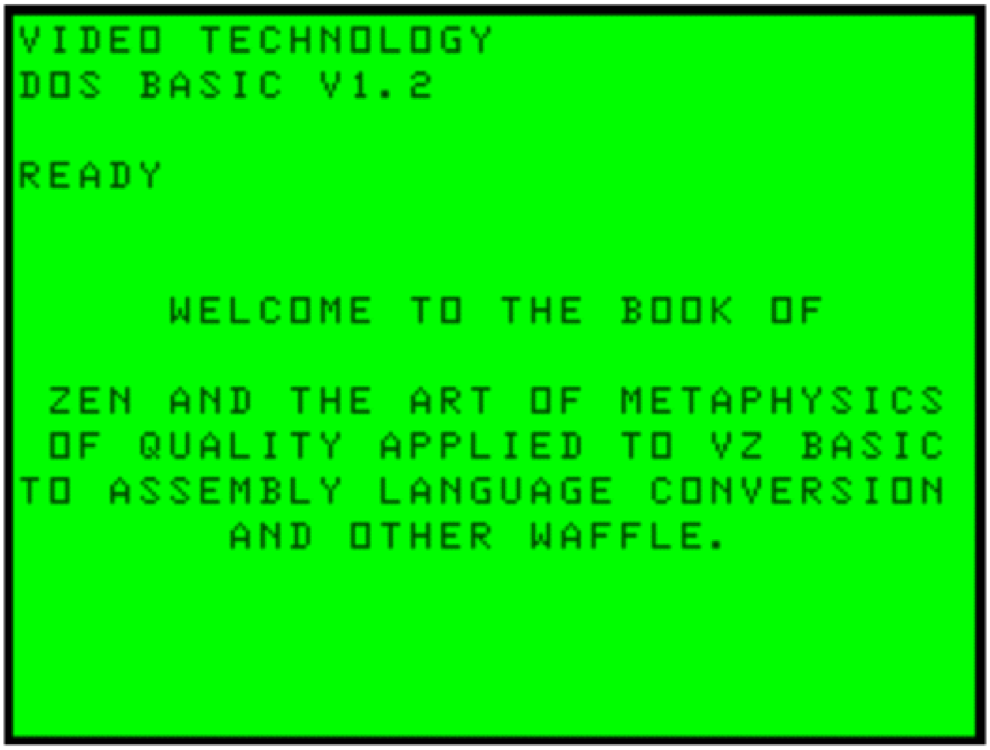
Silly /Dave presents : **2nd edition**



Your first goto book for of a lot of useless jibber jabber chit-chat with no real down to earth goal.

"What utter garbage" - Craig of Craigslist, Sep 1993.
"…the author could not construct a sentence if he had to." – Bishop, Aliens, 1986.
"Filth. Absolute filth" – Sally, last Tuesday evening on the second picnic table over in the RSL park

INDEX

BASIC INTRO

This book has no end goal. Therefore there is little point in reading any of this. There isn't much of an intro, and there isn't much of an ending. "Assembly!" The final frontier. No, wait, that was Star Wars or Jedi Trek the 13th something. Learning and being confident in assembly I always thought was not ever possible. Guess what, it isn't. I'm getting there though. Writing cool little routines in BASIC and then being able to convert them into Z80 assembler is a horrible experience for the wife, but most enjoyable for someone with utterly no life and whom has great delight in seeing a slow moving SET(X,Y) pixel in BASIC, move so quickly in ASM that you can't see it. That really has to be the highlight – seeing slow routines sped up hundreds of times within ASM. And then having the proud ego of loudly saying to no-one in particular "I wrote that". Moving on; This booklet has not an ounce of seriousness to it anywhere. It was written with love by a Circus lover. Hopefully you, the reader may even be here by now if you haven't already given up reading. All of these examples are written for the PASMO assembler which I have a slight preference to now over good old TASM. PASMO and SJASM / SJASMPLUS are the choice of assembler of the cool people these days. You need to become one of them even if it is just for one day. Obviously different assemblers require the overall structure to be a little bit different than the next. Eg TASM is fairly strict in its layout, PASMO seems a bit more relaxed in the layout. And SJASM couldn't care less what the structure is – it will just assemble whatever it is given.

TASM needs hex written out in form of $FF. SJASM and PASMO are happy with #FF or $FF.
TASM needs an END at the end of the listing. SJASM and PASMO couldn't care less.
TASM wants directives as .ORG .END .EQU .DEFB
PASMO prefers ORG, END, EQU, DEFB
SDJASMplus couldn't care less. It will accept anything and everything.

Study the commented listings and try them all out if you can be bothered. They were all assembled with pasmo, and should work on first assemble for you , dependant on typo'd.
With all assemblers, you either need to (1) add the .CVZ or .VZ header directly to within the asm listing. I personally do not know the .CVZ (cassette) file format, and therefore can not provide this info. .VZ file header will be shown elsewhere in this documentation. Or , (2) assemble without the header to an object file, and then use RBINARY.EXE to add the .VZ snapshot header to the object file, thusly creating a final .VZ snapshot file. Another alternative is to use Gavin's VZ Assembler8 GUI IDE to assemble directly to a .CVZ file output.

<u>The Quicky Summary :</u>
Z80 asm has a bunch of 1 byte and 2 byte combined registers. 8 bit (1 byte) registers are : A, B, C, D, E, F, H, L, I, PC, R, IXH, IXL, IYH, IYL. And then there are a bunch of these registers in duplicate form that can only be accessed at a certain time. They are A', B', C', D', E', F', H', L', IXH', IXL', IYH', IYL'.
All of these are good for storing numbers from 0 to 255. They are used for your everyday typical add, sub, mul and division stuff. A is your everyday common register, B is typically for looping, C for counting and summing, E another general all rounder, and F is a flags only, used after looping, comparing, for jumping and the likes.

16 bit (2 byte) registers : AF, BC, DE, HL, IX, IY. BC for 16 bit loops, IX and IY for indexing, DE typically as a destination and HL for a source – all dependant on the opcode commands of course. There is way more to it all than this, but this is just the quick basics after all, and I am probably the worst person on this planet to attempt to clearly explain it all.
No doubt the majority of people over the earlier 1980's and 1990's years have gone from learning Z80 assembly and then moved on to Intel 80x86 assembly. Having done the reverse, learning up to '386 assembly and studying and playing with it for ten years, then moving to Z80, it was certainly a smooth and very easy transition, and I somewhat recommend it as it is fully choice.

Example 1 : First example of BASIC TO ASM
10 CLS
20 PRINT "HELLO WORLD"

Within the ROM, there are tens and tens, if not a hundred plus routines all sitting there, used everyday via the normal BASIC tokens that are first interpreted and then these routines-in-rom are called. All of them are accessible direct from asm. Be aware though that even though they were written a hundred years ago by people, whom may have done on it a Friday afternoon. Meaning there may be a faster routine possible than what is embedded in silicon. We use the ROM routines coz they are there, and typically work well. CLS is at offset $01C9 in the ROM. Calling this, and you will clear the screen.
The Print String function is at offset $28A7. Point HL register to your string, call $28A7, and you have written your string.

To convert the above program, we'd start by using an ASM listing template or from scratch if you know the structure from off the top of your head.

```
               ORG     $8000          ; Start the program at the memory address of $8000.

               CALL    $01C9          ; CLEAR THE SCREEN, then return back here.
               LD      HL,  message   ; Make HL point to where the real message actual is.
                                      ; It's making HL point to where the message lives.
               CALL    $28A7          ; CALL the print string routine and return.
jp_forever:    jp      jp_forever     ; Jp forever back to our lp_forever label.


message        defb    'HELLO WORLD' , $0D , $0A , 00
```

If you were to type this in, assemble it with PASMO, then RBINARY it, you would have a FILE.VZ
Running in on a real VZ or an emulator will do exactly that : Print "HELLO WORLD' on to the screen and then loop for ever. If we removed the loop-forever, the program would continue to execute the instructions that form the string H E L L O W O R L D etc.(which does very little), and then continue into the unknown contents of memory after this. Trying this now simply shows that the program continuously shows Hello World, crashes, clears the screen, displays HELLO WORLD again, on a vicious continuous loop. What happens all depends on what the processor attempts to execute in memory. Or, of course, with the correct byte sequence in place, it would run "CIRCUS 2, The Penguins Revenge!" game.

Example 2: variable movements.

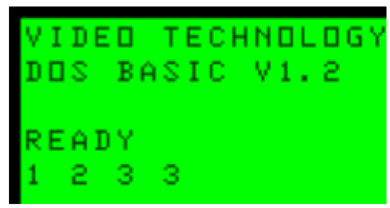
10 LET A=1 : LET B=2 : LET C = 3 : LET D = A + B

Each of the following examples are valid

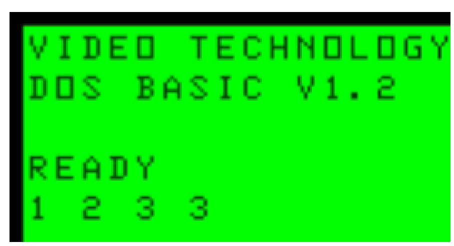
```
 (#) we will assume that every register is zero at the start.

 ...                ; --> A=0:B=0:C=0:D=0
 LD      A, 1       ; --> A=1:B=0:C=0:D=0   "LET A=1"
 LD      B, 2       ; --> A=1:B=2:C=0:D=0   "LET B=1"
 LD      C, 3       ; --> A=1:B=2:C=3:D=0   "LET C=1"
 ADD     A, B       ; --> A=3:B=2:C=3:D=0   "LET A=A+B"
 LD      D, A       ; --> A=3:B=2:C=3:D=3   "LET D=A"
 LD      A, 1       ; --> A=1:B=2:C=3:D=3   "LET A=1"
-------------------
 ...                ; --> A=0:B=0:C=0:D=0
 LD      A, 1       ; --> A=1:B=0:C=0:D=0   "LET A=1"
 LD      B, 2       ; --> A=1:B=2:C=0:D=0   "LET B=1"
 LD      C, 3       ; --> A=1:B=2:C=3:D=0   "LET C=1"
 PUSH    AF         ; --> A=1:B=2:C=3:D=0   Push AF register on to the STACK. (value 1)
 ADD     A, B       ; --> A=3:B=2:C=3:D=0   "LET A=A+B"
 LD      D, A       ; --> A=3:B=2:C=3:D=3   "LET D=A"
 POP     AF         ; --> A=1:B=2:C=3:D=3   Restoring A from the stack. So "LET A=1"
-------------------
 ...                ; --> A=0:B=0:C=0:D=0
 LD      A, 1       ; --> A=1:B=0:C=0:D=0   "LET A=1"
 LD      B, 2       ; --> A=1:B=2:C=0:D=0   "LET B=2"
 LD      C, 3       ; --> A=1:B=2:C=3:D=0   "LET C=3"
 LD      D, B       ; --> A=1:B=2:C=3:D=2   "LET D=B"
 ADD     A, D       ; --> A=3:B=2:C=3:D=2   "LET A=A+D" was performed.
 LD      D, A       ; --> A=3:B=2:C=3:D=3   "LET D=A" was performed.
 LD      A, 1       ; --> A=1:B=2:C=3:D=3   "LET A=1"
-------------------
 ...                ; --> A=0:B=0:C=0:D=0
 LD      A, 1       ; --> A=1:B=0:C=0:D=0   "LET A=1"
 LD      B, 2       ; --> A=1:B=2:C=0:D=0   "LET B=2"
 LD      C, 3       ; --> A=1:B=2:C=3:D=0   "LET C=3"
 LD      D, B       ; --> A=1:B=2:C=3:D=2   "LET D=B"
 PUSH    AF         ; --> A=1:B=2:C=3:D=2   Push AF register on to the STACK. (value 1)
 ADD     A, D       ; --> A=3:B=2:C=3:D=2   "LET A=A+D" was performed.
 LD      D, A       ; --> A=3:B=2:C=3:D=3   "LET D=A" was performed.
 POP     AF         ; --> A=1:B=2:C=3:D=3   Restoring A from the stack. So "LET A=1"
-------------------
```

Example 2.1: variable movements (more of)
10 LET A=1 : LET B=2 : LET C = 3 : LET D = A + B
20 PRINT A;B;C;D;
30 END

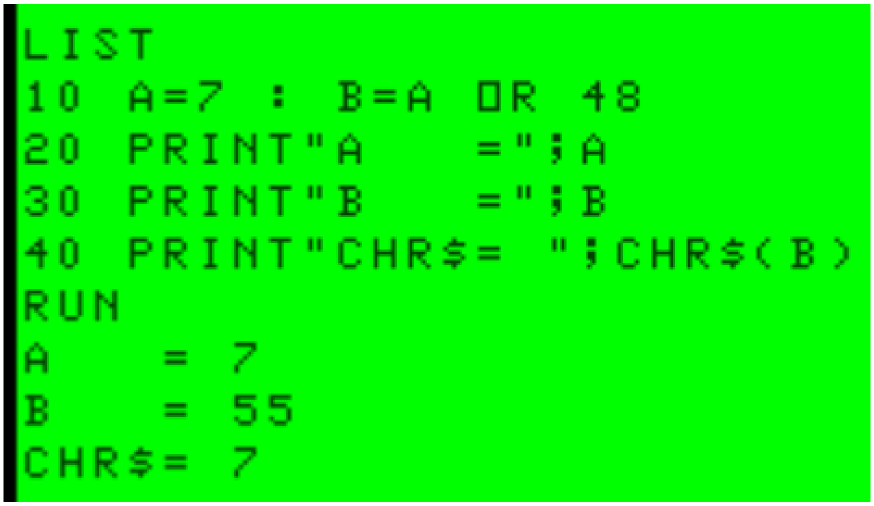
```
;10 LET A=1 : LET B=2 : LET C = 3 : LET D = A + B
;20 PRINT A;B;C;D;
;30 END

        org     $8000

        ld      a, 1            ; A=1           A=1,B=0,C=0,D=0
        ld      b, 2            ; B=2           A=1,B=2,C=0,D=0
        ld      c, 3            ; C=3           A=1,B=2,C=3,D=0
        add     a, b            ; A=A + B       A=3,B=2,C=3,D=0       ; need to use Reg A; it will be destroyed
        ld      d, a            ; D=A           A=3,B=2,C=3,D=3
        ld      a, 1            ; A=1           A=1,B=2,C=3,D=3       ; Fix up original Reg A value.
; ---------------------------------------------------------------
; Do value in Register A.
        or      $30             ; Dirty old trick : Simply by OR'ing 30hex to Reg A we change numerical
                                ; values into the numerical character that can be printed.
                                ; So we change value A=1 to become A=$31. Char(31) is character '1' in ASCII table.
        call    $033A           ; call print single character ROM routine. Whatever is in reg A it will print.
        ld      a, $20          ; we will print a space here.
        call    $033A           ; call print single character.
; ---------------------------------------------------------------
; Do value in Register B.
        ld      a, b            ; A = 2
        or      $30             ; A becomes $32
        call    $033A           ; call print single character.
        ld      a, $20          ; we will print a space here.
        call    $033A           ; call print single character.
; ---------------------------------------------------------------
; Do value in Register C.
        ld      a, c            ; A = 2
        or      $30             ; A becomes $32
        call    $033A           ; call print single character.
        ld      a, $20          ; we will print a space here.
        call    $033A           ; call print single character.
; ---------------------------------------------------------------
; Do value in Register D.
        ld      a, d            ; A = 2
        or      $30             ; A becomes $32
        call    $033A           ; call print single character.
        ld      a, $20          ; we will print a space here.
        call    $033A           ; call print single character.

loop:   jp      loop
```




```
; this can lead to calling a nice little subroutine to ease up on our code a little.

        org     $8000

        ld      a, 1            ; A=1           A=1,B=0,C=0,D=0
        ld      b, 2            ; B=2           A=1,B=2,C=0,D=0
        ld      c, 3            ; C=3           A=1,B=2,C=3,D=0
        add     a, b            ; A=A + B       A=3,B=2,C=3,D=1
        ld      d, a            ; D=A           A=3,B=2,C=3,D=3
        ld      a, 1            ; A=1           A=1,B=2,C=3,D=3
; ---------------------------------------------------------------
                                ; Do value in Reg A.
        call    print_char
        ld      a, b            ; Do value in Reg B.
        call    print_char
        ld      a, c            ; Do value in Reg B.
        call    print_char
        ld      a, d            ; Do value in Reg B.
        call    print_char
loop:   jp      loop            ; Loop forever to show the screen.

print_char:
        or      $30             ; Dirty old trick to change numerical to character.
        call    $033A           ; call print single character ROM routine.
        ld      a, $20          ; we will print a space here.
        call    $033A           ; call print single character.
        ret
END
```



Here we are using a simple old trick to change the value in a register to a printable numeric character. By OR'ing a single value with 30 hex (48 decimal), we change the value of, say, 7 to the character $37 (55 decimal) which is the alpha-numeric character '7'. This makes it nice and simple for things like additions or something like a game score when we go to print the value to screen. Imagine if you will a game score of value of 7. When this is printed to the screen, ie, PRINT CHR$(7), on the old ASCII table, it will attempt to display the Audible BELL character (beep) on most computers other than the VZ. The VZ doesn't do anything. PRINT CHR$(7); CHR$(7); on a Microbee or an Apple][will go beep beep. This is not ideal, therefore we do the OR, change the value into a printable character, and then display it.



<u>Example 3: Looping.</u> 10 FOR B = 1 TO 100 : NEXT I

Looping is fairly straight forward, and, looping of 8 bit values is incredibly easy to achieve in the world of assembly. The easiest method is to set a value to Register B and to use the DJNZ op-code. This is an automated loop that automatically decrements one from the current value of B. It then does an IF statement whereby if B <> 0, then jump back and loop through again. If B=0, the condition is set, the Zero flag is set, and the jumping at the DJNZ op-code loopy thing falls through, and code continues on.
Yes, the ASM example given below is going backwards from 100 to 1 instead of the wanted BASIC example of going from 1 to 100. The slightly longer but correct 1 to 100 method is shown in example 2.

```
                LD      B, 100      ; SET B register to Loop from 100 to 1
Label1:
                ...                 ; DO STUFF HERE
                DJNZ    Label1      ; Decrement B by one, JUMP if NOT ZERO to Label1
                ...                 ; Continue on...
```

```
        LD      B, 1            ; Set B register to 1
Label1:LD       A, B            ; Set up A reg for a Compare
        CP      101             ; Are we at 101 yet?
        JR      Z, Here         ; If at 101, then we jump out of loop
        INC     B               ; Otherwise we add one to reg B.
        JR      Label1          ; And go back again for another loop.
Continue:
        ...
```

Examaple 4: Looping and printing.

10 FOR I = 1 TO 99 : PRINT I; NEXT I

There are a few ways to demonstrate this method of looping and then performing a value-to-character conversion for printing. I am going to demonstrate a dumb method. Using a register for each size-width of the value. Meaning, we are counting from 0 to 99 , so we use a width of two characters. So , very badly, we are going to use register H for the TENS (left hand value), and register L for the single units. We will loop from 1 to 99, or by using the B / DJNZ counter we will actually set the Loop count to 99 and count back to 1. We'll start counting from 1 to 9 in register L, an then compare the L register value with 10. If L=10, we reset L back to zero, and then add one to the TENS value, being register H. If L is not equal to 10 yet, we ignore the reset & increment-H-register. Then decrement one from B register and jump back to the start of the loop for another round. Very inefficient, but can clearly show how to go about setting up a two width counter.
In the past I have written some very bad code using this exact method. I don't care.

```
        org     $8000

        LD      B, 99                ; Do for 99 times.   This is our : FOR I = 01 TO 99.
        LD      H, 0                 ; First digit of I.  This is the TENS on the left hand side.
        LD      L, 1                 ; second digit.      This is the single values on the RHS.

Label1:LD       A, H                 ; Get ready to display first digit.
        OR      $30                  ; Perform our value-to-CHR$ trick
        CALL    $033A                ; Do PRINT single CHR$

        LD      A, L                 ; Get ready to display second digit.
        OR      $30                  ; value to CHR$() trick.
        CALL    $033A                ; Do PRINT single CHR$

        INC     L                    ; add one to the 'single' left hand side value.
        LD      A, L                 ; Re-load value into A for comparing if > 10 decimal.
        CP      10                   ; Compare A with value of 10.
        JP      nz, Loop             ; Not zero, therefore we are in range of 0 to 9 and JP
                                     ; ELSE we do this...
        LD      L, 0                 ; We zero the second digit.
        INC     H                    ; We add one to the 'TENS' first digit. 9->10, 19->20, 29->30

Loop:   LD      A, 13                ; Display a carrage line
        CALL    $033A                ; Do PRINT single CHR$

Keys:   ld      a, ($68ef)           ; we read the appropiate location for the SPACE bar
        and     $10                  ; Comparing 10 hex which is space bar.
        jr      nz, Keys             ; if not 10 hex, we loop forever until we see $10. (Space Bar)

        DJNZ    Label1               ; Decrement one from our 99 loop. jump back to Label1 if not yet zero.
                                     ; Once B = 0, we then...
Forever:JP      Forever              ; Jump forever to show on the screen.
```

Example 5 – Sound

Rom call $345C is the VZ's sound. Load up HL as the frequency and BC the duration length, call the call, and we have the theme to Star Wars. Darth Vader's entrance music all came from a VZ. True story.

```
LD      HL, note
LD      BC, duration
CALL    $345C
```

	BASIC	Assembly HL value
Low C	SOUND 4,1	526 decimal
Middle C	SOUND 16, 1	259 decimal
High C	SOUND 28,1	127 decimal

Below are two examples, each will play seven individual notes, the user presses <S>, and then a short tune plays. Both examples show how music can be written out in ASM for the VZ. The first example shows how music can be achieved using the sound routine in the VZ's ROM, whilst the second listing shows how we can achieve the same thing by not using the ROM routine, and writing our music data directly to the memory address latch, which is physically linked to the piezo speaker with copper tracks on the motherboard. The memory address latch is located at $6800. It is actually a 2k chuck of your memory addressing space and takes in all memory addresses from $6800 through to $6FFF. Why so large? Rhetoric question here coz I simply don't know the answer. I'm going to guess that there has been way less than 1k of addressing that has ever been used by everyone combined together. A copy of the address latch of $6800 (26624 decimal) also sits in memory at 30779 - which should be rightly used for "last speaker state" when playing 1-bit audio music. 30779 is a POKE number that has been around since forever and would be one of the more famous ones for its use.

```
        ORG     $8000
; ---------------------------------------
; Play indiviudal notes one at a time
; ---------------------------------------
here:   di                              ; Disable interrupts
        ld      a,$10                   ; Individual notes. Set the note to $10.
        call    sound                   ; $10 will be a high note.  Play sound
        ld      a,$20
        call    sound
        ld      a,$30
        call    sound
        ld      a,$40
        call    sound
        ld      a,$50
        call    sound
        ld      a,$60
        call    sound
        ld      a,$70
        call    sound

key1:   ld      a, (0x68fd)             ; Press S to continue. Read latch for S key row.
        and     0x02                    ; test for <S>
        jr      nz, key1                ; If not, loop forever until <S> is pressed.

;------------------------
; Play a short tune.  Like performing a : SOUND 22,1;24,1;22,1;26,1 etc.
;------------------------
        ld      ix, Tune                ; read in and play a bunch of
loop:   ld      a, (ix)                 ; music data from 'Tune' into register A
        cp      0                       ; compare A with 0
        jr      z, forever              ; If A=0 then jump to forever (quit)
        call    sound                   ; play the sound
        inc     ix                      ; increase pointer to point to the next note in "Tune".
        jp      loop                    ; jump back to the start of the loop

forever:jp      forever                 ; End : Loop forever


sound:  ld      h, 0                    ; set H to be 0, don't need it for this example.
        ld      l, a                    ; Reg L becomes our music data tune value for ROM call.
        ld      bc, 75                  ; Set our note duration.
        call    $345C                   ; call ROM sound routine.

        ld      bc, $1200               ; This is a set Sound duration delay.
delaylp:dec     bc                      ; Without this, sounds are way too quick.
        ld      a, b                    ; This is like a BASIC "FOR I = 1 TO 4600 : NEXT I" delay.
        or      c                       ; But quicker because it isnt being interpreted.
        jr      nz, delaylp
        ret


Tune:   db $aa,$0a,$aa,$0a,$32,$28,$ff  ; A hopeless Tune/music in some sort of data format.
        db $aa,$0a,$aa,$0a,$32,$28,$ff
        db $aa,$0a,$aa,$0a,$32,$28,$ff,$00
```

```
        ORG     $8000
; ----------------------------------------
; Play indiviudal notes one at a time
; ----------------------------------------
here:   di                              ; Disable interrupts
        ld      b,$10                   ; Individual notes. Set the note to $10.
        call    sound                   ; $10 will be a high note.  Play sound
        ld      b,$20
        call    sound
        ld      b,$30
        call    sound
        ld      b,$40
        call    sound
        ld      b,$50
        call    sound
        ld      b,$60
        call    sound
        ld      b,$70
        call    sound

key1:   ld      a, (0x68fd)             ; Press S to continue. Read latch for S key row.
        and     0x02                    ; test for <S>
        jr      nz, key1                ; If not, loop forever until <S> is pressed.

;------------------------
; Play a short tune.  Like performing a : SOUND 22,1;24,1;22,1;26,1 etc.
;------------------------
        ld      hl, Tune                ; read in and play a bunch of
loop:   ld      a, (hl)                 ; Place music data from HL pointer of 'Tune' into Reg A.
        cp      0                       ; compare A with 0
        jr      z, forever              ; If A=0 then jump to forever (quit)
        ld      b, a                    ; load into reg B the note
        call    sound                   ; play the sound
        inc     hl                      ; increase pointer to point to the next note in "Tune".
        jp      loop                    ; jump back to the start of the loop

forever:jp      forever                 ; End : Loop forever


sound:  ld      d, b                    ; Copy of our "pitch" to be used again after B is destroyed.
sndlp:  ld      a, 0                    ; send a Positive pulse to speaker. Bits 0 & 5 are zero.
        ld      ($6800), a              ; Latch is at $6800
        ld      b, d                    ; restore reg B. Needed upon 2nd & onwards iteration of loop
        djnz    $                       ; Wait / Loop B times. Part of the pitch as well as duration.
        ld      a, 33                   ; Send a negative pulse to speaker. Bits 0 & 5 are set to 1.
        ld      ($6800), a              ; Latch is at $6800
        ld      b, d                    ; restore our original "pitch" value.
        djnz    $                       ; Wait / Loop B times. Part of the pitch as well as duration.
        dec     c                       ; Initially C = 0, dec C makes this a 256 loop.
        jr      nz, sndlp               ; Jump if not yet zero.

        ld      bc, $1200               ; This is a set Sound duration delay.
delaylp:dec     bc                      ; Without this, sounds are way too quick.
        ld      a, b                    ; This is like a BASIC "FOR I = 1 TO 4600 : NEXT I" delay.
        or      c                       ; But quicker because it isnt being interpreted.
        jr      nz, delaylp
        ret


Tune:   db $aa,$0a,$aa,$0a,$32,$28,$ff  ; A hopeless Tune/music in some sort of data format.
        db $aa,$0a,$aa,$0a,$32,$28,$ff
        db $aa,$0a,$aa,$0a,$32,$28,$ff,$00
```

One last quick thing to note is the speed between the two listings. You can easily hear the difference between the first listing using the direct sound routine and the second listing using the ROM call. The second listing sounds slower - this is due to the extra overheads in calling the ROM routine, and once you are in there, there are further stack commands which slow things down enough that you can actually hear this in the sound pitch and duration,

The first listing is also done a bit dodge-ly. It is essentially using the pitch as a duration loop to set the tone. What is tone anyway? The quickness of the vibration isn't it?. The quicker the vibration the higher the tone, right? We are using the pitch in the first and second DJNZ $ loops to adjust the timing that we are sending of bits 0 and 5 to the $6800 address. We then set the actual duration length of the note further in a separate delay. Performing this routine quick enough and with the right values, yes, yes you can have star wars theme playing from your piezo speaker. But this 1-bit audio is beyond this book.

Example 6: Waiting for the keyboard. - Keyboard input

In C (Z88dk) one would use this kind of generic code for keyboard input.

```
if(inch()=='0') { printf("You pressed <0>" ;}
if(inch()=='1') { printf("You pressed <1>" ;}
if(inch()=='2') { printf("You pressed <2>" ;}
if(inch()=='3') { printf("You pressed <3>" ;}
if(inch()=='4') { printf("You pressed <4>" ;}
if(inch()=='5') { printf("You pressed <5>" ;}
```

We've found though that the VZ's inch() code can be a little buggy some times, and reading directly from the latch is a far better method of getting a near 100% accuracy keyboard read.

```
if((mem[0x68ef] & 0x10) == 0) { printf("You pressed <space>";}
if((mem[0x68fd] & 0x1)  == 0) { printf("You pressed <G>";}
if((mem[0x68fd] & 0x2)  == 0) { printf("You pressed <S>";}
if((mem[0x68df] & 0x10) == 0) { printf("You pressed <0>";}
if((mem[0x68f7] & 0x10) == 0) { printf("You pressed <1>";}
if((mem[0x68f7] & 0x2)  == 0) { printf("You pressed <2>";}
if((mem[0x68f7] & 0x8)  == 0) { printf("You pressed <3>";}
if((mem[0x68f7] & 0x20) == 0) { printf("You pressed <4>";}
if((mem[0x68f7] & 0x00) == 0) { printf("You pressed <5>";}
if((mem[0x68fb] & 0x04) == 0) { printf("You pressed <LEFT SHIFT>";}
```

This is essentially the keyboard table with the appropraite memory locations. Taken straight from the reference manual.

	x20	x10	x8	x4	x2	x1
68FE	R	Q	E		W	T
68FD	F	A	D	ctrl	S	G
68FB	V	Z	C	SHFT	X	B
68F7	4	1	3		2	5
68EF	M	SPC	,		.	N
68DF	7	0	8	-	9	6
68BF	U	P	I	RTN	O	Y
687F	J	L	K	:	L	H

This leads to a very simple read conversion to asm for the VZ.

Read in a memory location, say, $68F7.
Mask register A with the corresponding hex value up the top of the table for the KEY that we are after. As an example, $8 for the <3> key. Depending on the masking, the flag will either be set or not set, and by using this we can then do something dependant on if the key was pressed or not.

```
        LD      A, ($68F7)              ; read in memory location of the 4,1,3,2,5 key row.
        AND     $08                     ; mask and test for the correct value
                                        ; Does Register A = 8 ?
        JR      Z, jump_here            ; flag was set if <3> was pressed. Do the Jump!
        …                               ; else, <3> was not pressed, continue on doing other stuff
Jump_here:                              ; do stuff here coz <3> key was pressed,
```

```
Loop:
        ld      a, ($68fd)          ; Key : S
        and     $02
        jr      z, key_s_pressed    ; Jump if S is pressed
        ld      a, ($68fd)          ; Key : G
        and     $01
        jr      z, key_g_pressed
        ld      a, ($68fb)          ; Key : Z
        and     $10
        jr      n, key_z_pressed
        ld      a, ($68fd)          ; Key : D
        and     $08
        jr      z, key_d_pressed
        ld      a, ($68fd)          ; Key : A
        and     $10
        jr      z, key_g_pressed
        ld      a, ($68fb)          ; Key : Z
        and     $10
        jr      n, key_z_pressed
        jp      Loop                ; Else loop back to Loop.

loop1:  ld      a, ($68fd)          ; Key : S
        and     $02
        jr      nz, loop1           ; Loop forever until <S> is pressed.

loop2:  ld      a, ($68ef)          ; Key : SPACE
        and     $10
        jr      nz, loop2           ; Loop forever until <SPACE> is pressed.

loop3:  ld      a, ($68f7)          ; Key : 1
        and     $10
        jr      nz, loop3           ; Loop forever until <1> is pressed.
```

The VZ's ROM also, of course, has a keyboard scanning routine at $2EF4 which is used upon each and every time you press a key on the VZ's keyboard, be it in BASIC or line entering in ASM. This routine just runs nice and silently in the background. Within BASIC, the key-presses are then sent on to other parts to display , accept a line entry or some value that is inputted.

As shown in the Technical Reference manual, the ROM routine is also reasonably simple to use.

```
key:    CALL    $2EF4           ; call the ROM routine.
        OR      A               ; test reg A with 0. If a key is pressed it will return a value anything but 0
        JR      Z, key          ; If Reg A = 0, then no key was pressed, Jump again for another scan.
                                ; Essentially, we wait until a key is pressed.
        CP      $D              ; 13 decimal. ASCII value of <CR>. Compare register A with 13.
        JR      Z, do_return    ; We jump if it is a match. <ENTER> has been pressed.
        CP      $41             ; 65 decimal. ASCII value of the letter A, also the pressing of key <A>
        JR      Z, do_a         ; Jump if <A> key was pressed.
        CP      $42             ; 66 decimal. ASCII value of the letter B, also the pressing of key <B>
        JP      key             ; We jump again looking for a key pressed, and then if it is <ENTER> or
<A>

do_return:
        ...
do_a:
        ...
```

Example 7 : Assembling directly to .VZ snapshot.

Purist's will skip this part, as the .VZ snapshot file format is a hack and nothing more than a hack. And rightly so. It was created by Brian Murray way back in the early days just to get something to work , and a such, from whichever side of the fence you are on, has stuck and has been pretty much the majority standard for VZ snapshots, be it good or bad.

Unfortunately I have no documentation on the more formal and proper file method being the ".CVZ" cassette file format, of which, MAME (I think?), DSVZ200 and JVZ200 emulators use. So in this section, we will quickly look at how to assemble a listing to the .VZ file snapshot.
There are two methods, either including the 23 bytes of the .VZ header into your own assembly listing, and assembling or compiling the lot into a direct outputted machine code object code that is the .VZ snapshot.

Or, by assembling a generic Z80 listing to object code, then running the Wintel executable file "RBINARY.EXE" (created by Brian or Guy Thomason years ago), which simply amends the .VZ file header to the machine code object file, and spits out the resulting .VZ snapshot file that all known emulators do read.

RBINARY.EXE utility can be found on most good VZ200 It can also be found in the files section of the VZ/Laser Facebook group. You will need to rename it. Worse case scenario, email the author for it.

```
; Code for .VZ snapshot header.

        defb    'VZF0'
        defb    'AGDGAME         '      ; 16 spaces for filename.
        defb    $f1
        defb    $00     ; lb $7B00
        defb    $7B0    ; hb $7B00
        org     $7B00
```

METHOD 1

Listing that shows how to include the .VZ file header into the original source file.

```
nolist
write "program.vz"


header db      "VZF0"                  ; .VZ snapshot header
       db      "PROGRAM FILE    "      ; VZ program Name. 16 chars.
       db      $f1                     ; VZ program type. F1=Binary. F0=BASIC.
       dw      start

       org     $7b00

start: di
       ld      a,8
       ld      ($6800),a               ; Set mode(1)

       ld      a,  255                 ; Set background to red
       ld      hl, $7000
       ld      de, $7001
       ld      bc, 2048
       ld      (hl),a
       ldir


key1:  ld      a, (0x68ef)             ; Wait forever until <SPACE> is pressed
       and     0x010
       jr      nz, key1


       ld      a,  85                  ; Set background to yellow
       ld      hl, $7000
       ld      de, $7001
       ld      bc, 2048
       ld      (hl),a
       ldir

key2:  ld      a, (0x68fd)             ; Wait forever until <S> is pressed
       and     0x02
       jr      nz, key2
```

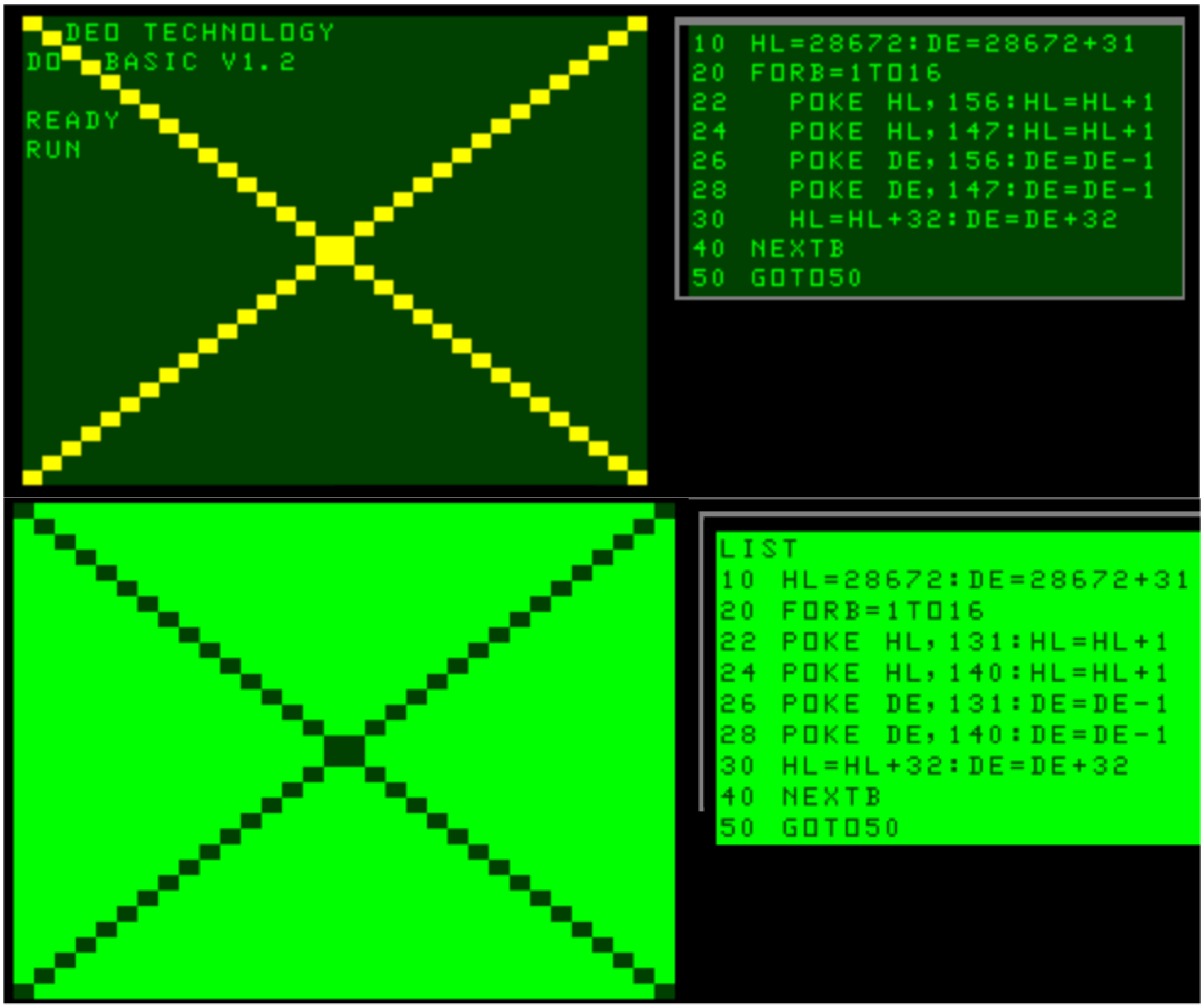
Saving this as TEST.ASM and using SJASMPLUS you'd simply do a :
SJASM TEST.ASM

METHOD 2 – using Rbinary utility

```
       org     $7b00

start: di
       ld      a,8
       ld      ($6800),a               ; Set mode(1)

       ld      a,  255                 ; Set background to red
       ld      hl, $7000
       ld      de, $7001
       ld      bc, 2048
       ld      (hl),a
       ldir


key1:  ld      a, (0x68ef)             ; Wait forever until <SPACE> is pressed
       and     0x010
       jr      nz, key1


       ld      a,  85                  ; Set background to yellow
       ld      hl, $7000
       ld      de, $7001
       ld      bc, 2048
       ld      (hl),a
       ldir

key2:  ld      a, (0x68fd)             ; Wait forever until <S> is pressed
       and     0x02
       jr      nz, key2
```

Again, saving this as test.asm, you would perform the following :
PASMO TEST.ASM TEST.OBJ
RBINARY TEST.OBJ TEST.VZ

This particular example originally came about from the C64 Twitter crowd where a small competition started to display an X on the screen with the smallest code. This particular example is a little different to the other examples, in that, this was written in assembler to begin with, and in order to recreate a basic-to-asm example, the basic code had to be written from the asm listing.



We can't use PRINT@ here since upon trying to print in the lower bottom right hand corner, the VZ always will want to add a <CR> which is normal, but for our purpose it wrecks the final display. There is no way around this as far as I know when using PRINT@; …so we use POKE.

We first need two variables, one for each line. There are 16 lines in height, in which each iteration we need to display a 'top' graphic block, then a 'bottom' block for line 1 (L to R), and the reverse for line 2 (R to L). We then add the width of the screen to both variables to go down to the next line. HL can become line 1 and DE line 2. Can use B for the DJNZ loop. The tricky part in this first asm attempt is that , in the lower section, we really want to do a ADD HL, 32 and ADD DE, 32. Z80 asm doesn't allow for this, so we need to get creative to do our additions. It is a bit of a mess, however the comments are reasonably clear.

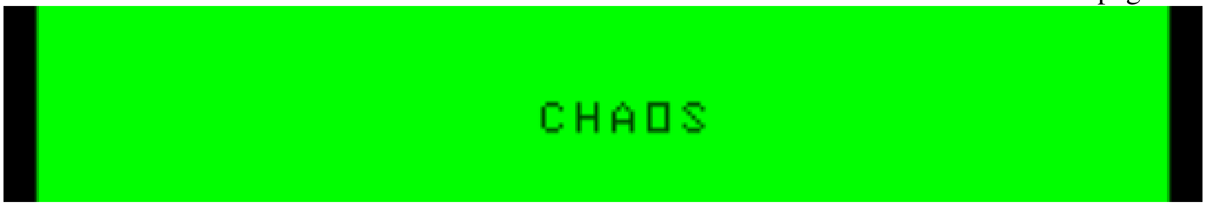
```
; Displays an X on the VZ.  Attempting the smallest sized code.
; VZ has 23 byte loader/overheads. Assemble with PASMO assembler.
;
; Loop 16 times for height of screen
; HL = top left. DE = top right.
; HL will INC whilst DE will DEC.
; Four writes per line.
;     HL will be upper block then lower block. Heads left-to-right.
;     DE will be lower block then upper block. Heads right-to-left.
; INC HL and DE pointer so that they both share same adding of 32 to go to
;   next line, as well as increasing HL to go right, and decreasing DE to go left.
; End Loop
; Jump forever to hold on screen.

; * Could attempt to reuse $1F (31) in e for 'next line' as it seems a waste that its already there.
; * James uses LDD which is a move contents from HL to DE "LD (DE),(HL)" and dec's DE,HL,BC.
; * Could use  LDI which is a move contents from HL to DE "LD (DE),(HL)" and increases DE,HL and decreases BC.
; * Can't do a "ld hl, de" so need to exchange de/hl registers to add 32 to both hl and de.
; * B is for global loop. Have also run out of nice registers for 'add 32' so re-use BC; so need push/pop bc combo.
; * Can use IX,IY,IXL,IXH,IYL,IYH registers, but their larger opcode size is useless.
; * Could use SP for 'add 32' to remove push/pop bc and but messing with stack pointer is not always the best.
        ORG $8000               ; Program to kick start at 32768.
start:  LD      B, 16           ; Loop 16 times for height of screen and then loop2 to show.         FORB=16to1STEP-1
        LD      hl, $7000       ; HL becomes the pointer to top left corner                          HL=0
        LD      de, $701F       ; DE becomes the pointer to top right corner                         DE=31
loop:   ld      a, $83          ; Wanting the "upper block char". a=131, which is like:              A$=CHR$(131)
        ld      (hl), a         ; Send it to top left hand corner                                    PRINT@HL,A$;
        inc     hl              ; Inc pointer to the right. loop1:$7001. Loop3:$7002. Loop5:$7003    HL=HL+1
        ld      (de), a         ; Send it to top right hand corner.                                  PRINT@DE,A$;
        dec     de              ; Dec pointer to the left.  Loop1:$701E. Loop2:$701D. Loop3:$701C    DE=DE-1
        ld      a, $8c          ; Wanting lower block character. a=140, which is like:               A$=CHR$(140)
        ld      (hl), a         ; Send it to left-to-right pointer HL.                               PRINT@HL,A$;
        inc     hl              ; Inc left hand pointer again (2 char blocks for HL). (#1)           HL=HL+1
        ld      (de), a         ; Send it to rght-to-left pointer DE.                                PRINT@DE,A$;
        dec     de              ; Dec right hand pointer again (2 char blocks for DE).(#2)           DE=DE-1
        push    bc              ; Have run out of registers to use to add 32 to HL&DE. So reusing BC.  TMP1=BC
                                ;  In order to re-use BC for adding, need to keep the 16 loop B reg.
        ld      bc, 32          ; 32 into BC for adding to HL and DE - to go to next line.           BC=32
                                ; Have already setup HL and DE so that they can now both use a shared
                                ;  single "ADD 32", rather than a seperate "ADD 33" and "ADD 31"
        ex      de, hl          ; Can't do a "ADD HL, 32" and can't do a "ADD DE, BC" in Z80 so      TMP2=HL. HL=DE
                                ;  need to use HL temporarily to add 32 via BC for both HL and DE regs.
                                ; Temporarily move HL into DE and DE into HL. Adding 32 to DE first.
        add     hl, bc          ; Original DE pointer is now currently in HL. Add 32 (in BC) to HL.  HL=HL+BC
        ex      de, hl          ; Put HL back into DE. Put DE back into HL.  DE here has +32.        DE=HL. HL=TMP2
        add     hl, bc          ; Now that HL pointer is back to itself, add 32 to it.               HL=HL+BC
                                ;  At this point both HL and DE are on the next line, as well as,
                                ;  are setup correctly (offset-wise via #1 and #2 above) that they
                                ;  are pointing to the correct location for the next sending-of-char.
        pop     bc              ; Restore the 16 loop counter which is used by DJNZ looping opcode.  BC=TMP1
        djnz    loop            ; Loop de loop exactly 16 times, back up to loop as long as reg-B    NEXTB (For-to-next)
                                ;  is greater than NON-ZERO, which in our case will keep looping
                                ;  as long as reg-B is greater than 0. When a jump to label 'loop'
                                ;  occurs the opcode djnz automatically decreases register B by one.
                                ;  as it loops back up to loop. Opcode djnz does the actual 16to1step-1.
loop2:  jr      loop2           ; Loop de loop2 just to show on screen. Can remove it to save 2 bytes  220 GOTO 220
                                ;  however the program will continue to execute whatever is in memory.
                                ;  It might syntax error, or crash, or hang, or play Circus.
```

Note that with the For-To-Next loop in the comments is not quite correct. It is showing you essentially the loop, (setting up variable B here), however note that the ending DJNZ LOOP jumps back to the label LOOP. Not back to the LD B, 16. This is important, as the BASIC comments will not work exactly as they are. The FORB=16TO1STEP-1 should be with the LOOP label, since that is where the corresponding DJNZ jumps back to.

Second attempt, we move things into 8 bit registers where we can since 16 bit isn't all that necessary in this example, and does allow for some simpler adding. It can also reduce our overall code size. The below is just included to show that things can be further improved on the code size of things.

```
; Displays an X on the VZ. Attempting the smallest sized code.
; VZ has 24 byte loader/overheads in the ".VZ" header.  52 bytes.

; H = line 1 ( left to right)
; L = line 2 ( right to left)

 ORG $8000                                ; Kick off at location $8000 in memory.
            ld          h, $70            ; HL = $7000
            ld          bc, $8c83         ; B = TOP block, C = LOW block
            xor         a                 ; ZERO A. Do not assume A = 0 at startup.
each_row:   push        af                ; store A and flags for later.
            xor         $1f               ; trick to alternate at every iteration between
                                          ; 0 to 31, 1 to 30, 2 to 29, 3 to 28, 4 to 27 etc.
            ld          l, a              ; 1st iteration:line 2 = starting at 31. (top right)
            ld          (hl), c           ; 1st iteration:top block
            dec         hl                ; 1st iteration:line 2 = offset 30.
            ld          (hl), b           ; 1st iteration:low block.
            pop         af                ; restore A = 0.
            ld          l, a              ; 1st iteration:line 1 = starting at 00. (top left)
            ld          (hl),c            ; 1st iteration:top block.
            inc         hl                ; 1st iteration:line 1 = offset 01.
            ld          (hl),b            ; 1st iteration:low block.
            add         a, 34             ; add 34 to goto correct postion on next video line
            jr          nc, each_row      ; Loops 8 times (255 div 8) before dropping thru
            inc         h                 ; inc line 1 offset position
            inc         l                 ; inc line 2 offset position
            jr          nz, each_row      ; Loops once more (while L <> 0)

loop:       jr          loop
```

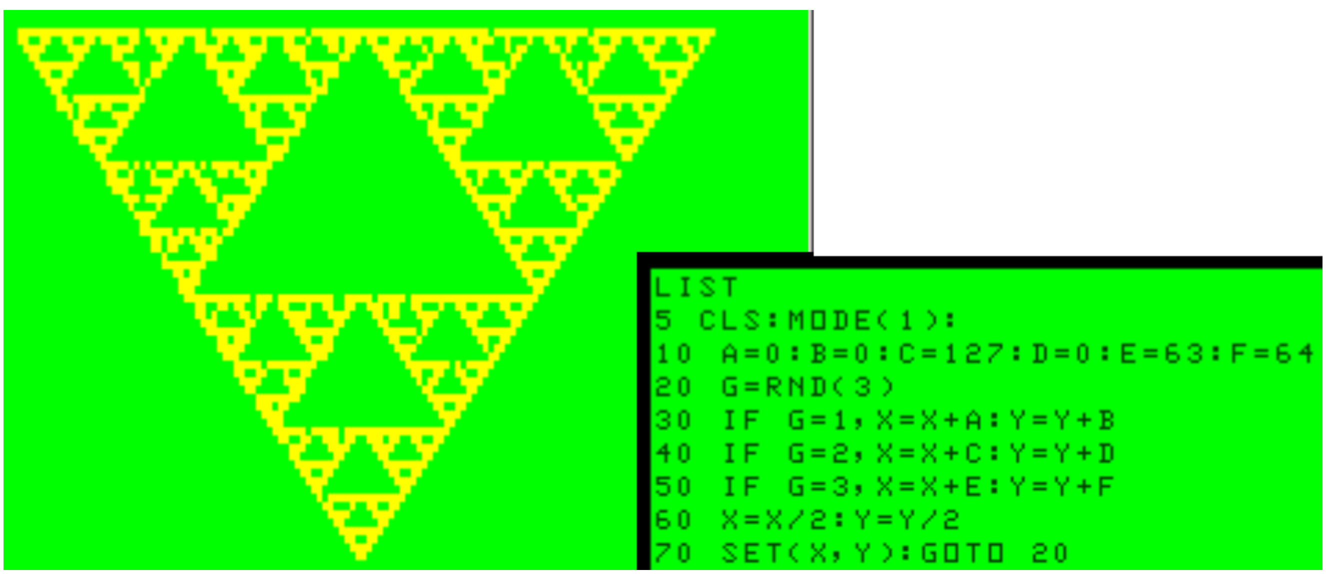
This page intended to leave <u>BANK</u> (….worded just like the VZ DOS Manual)



Chaos is actually Serpinksi's Gasket or Triangle. Chaos is a true love of mine. It was given to me as an Apple][e BASIC listing by my computer teacher in, perhaps, 1988 era in high school. I ran with it!! I had it typed in and running on the Apple by lunch time and I was gob-smacked (after waiting minutes and minutes). Then over on to the VZ it was. Few years later (about six) I had it running nicely in 320x200 VGA on the PC in Turbo Pascal. At the time I was learning assembly on the PC and after a long time, I managed to get it running nicely from an original 2000 bytes of dribble, down to 100 bytes, and after a few more years finally got it running at 62 bytes. It was even entered in as a demo for a 64 byte assembly demo competition. Then along came a very simple 23 byte algorithm that just blew mine out of the water - but that’s another story

I played with this for ages, days / months / years, animating it to fly around the screen, rotating it, and changing shades of colour. Somewhere along the way I played with the randomness, creating a replication of Serpinksi's Carpet, Serprinki's Dragon and Serprinki's Fractal Leaf. Its extraordinary how a very simply random routine can create such beautiful designs - hence from the Chaos of random numbers comes beauty.

Some very cluey folks have over the years created 256 byte VGA demo's that are 3D, flying through Serpinki's cubed carpet in 3D as well as 3D Gasket as a 3D pyramid. These demos , although extremely small in size (256 bytes) use Pentium math co-processor assembly code and self-building math tables that would occupy perhaps megabytes of memory. Well beyond my league.
Years later I brought Chaos back over on to the VZ running in assembly, and thus, bringing my entire story of Chaos back on its self.

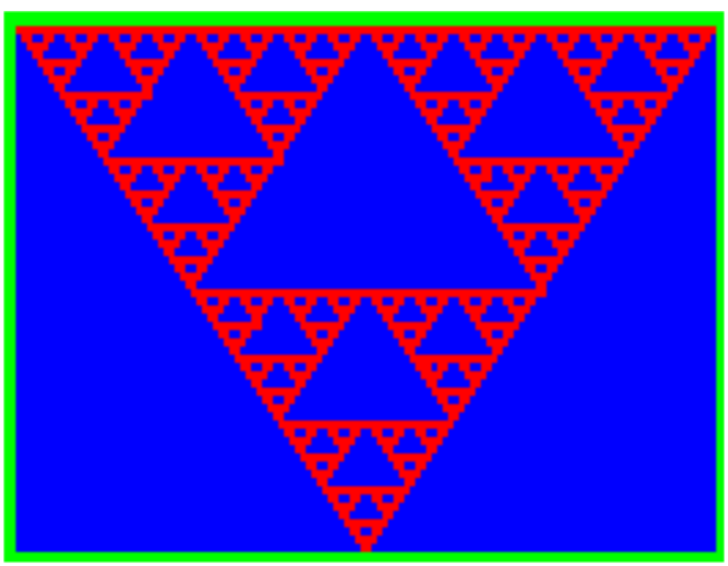


We can produce the same pattern in C by using something similar - although it is a little rough around the edges. It is much quicker of course, being compiled, than being interpreted by BASIC.

```
#include <vz.h>
#include <graphics.h>
#include <stdio.h>
#include <sound.h>
#include <stdlib.h>
#include <ctype.h>
#include <strings.h>
#include <conio.h>
#include <math.h>
int rnd, x, y, z, i, j, k;
int main() {
        vz_mode(1);
        vz_setbase(0x7000);
        vz_color(1);
        vz_bgrd(1);
        z = 1;
        x = 63;
        y = 63;
        i = 63;
        j = 63;
        while (z == 1){
                rnd = rand(255);
                if ((rnd > 10921) && (rnd < 21846)){
                        x = x + 64;
                        y = y + 64;}
                if (rnd > 21845){
                        x = x + 128;
                        y = y + 0;}
                x = x /2;
                y = y /2;
                vz_plot(x,y, 2);
        }
}
```

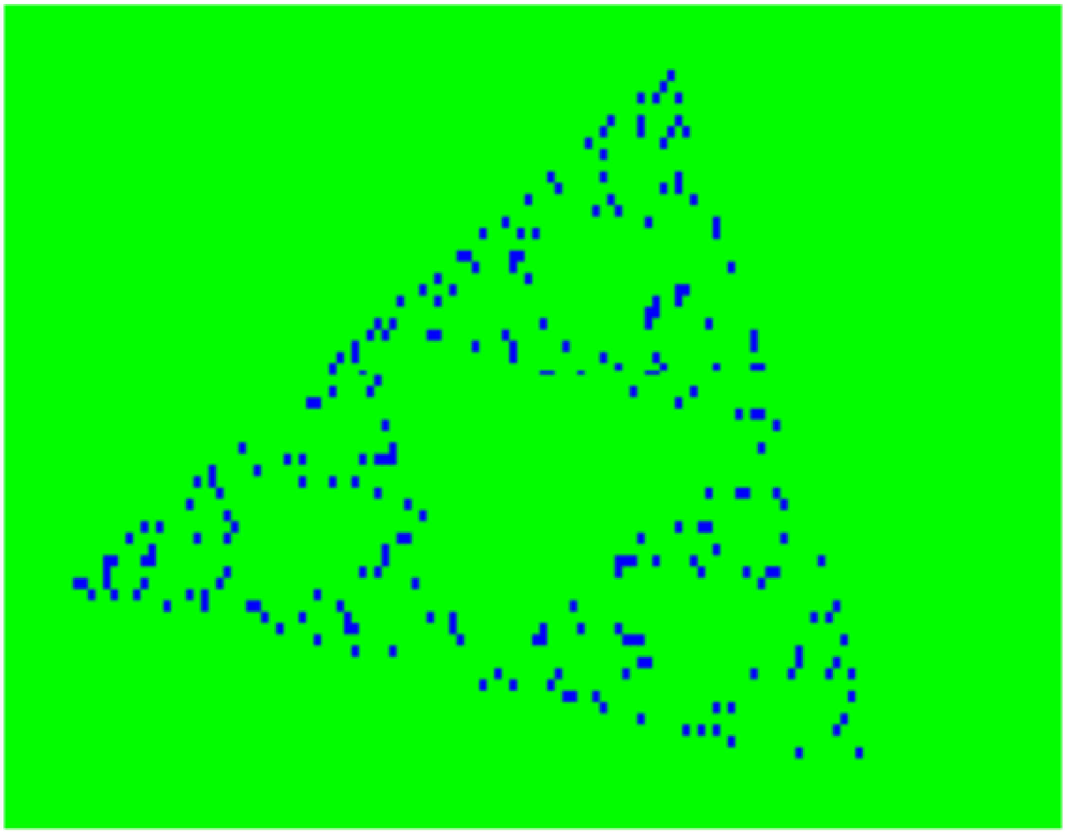
So, to bring this over into Z80 asm we need the following things :

(0) A continuous forever loop,
(1) Mode (1) enable and setting of initial default values.
(2) Simply but effective random routine,
(3) Addition routine,
(4) Subdivision routine
(5) Plotting pixels routine.



```
        ORG $8000
        ld      a,8                     ; mode (1)
        ld      ($6800),a
        ld      ix, 64          ; preset X to be 64.
        ld      iy, 64          ; preset Y to be 64.
        ld      hl, $7000       ; MODE(1) CLS
        ld      de, $7001
        ld      a, 170          ; A=0=green. A=85=yellow. A=170=blue
        ld      (hl), a
        ld      bc, 2048
        ldir
random:                         ; Generate a random number.
rand1   equ $+1                 ; output A= RND(255)
        ld      a,$A6
rand2   equ $+1
        ld      hl,$8243
        inc     l
        dec     h
        add     a,(hl)
        ld      (rand2),hl
        rlca
        rlca
        sub     h
        add     a,l             ; A is:   0<=A<=255.
        ld      (rand1),a

        ld      bc, 64          ; HEIGHT and HALF WIDTH
        cp      85              ; 1/3 of RND(255) = 85.
        jr      c, calc         ; If below 85 then JP and add (0,0)
        add     ix, bc          ; Else add (64, 0)
        cp      170             ; 2/3 of RND(255) = 170.
        jr      nc, next        ; If above 170 then JP and add (0,64)
        add     ix, bc          ; Else add another (64,0) = (128,0)
        jp      calc            ; Jp to division bit.
next:   add     iy, bc          ; Add (0, 63)
calc:   ld      a, iyl          ; Division section
        srl     a               ; Y=Y/2
        ld      iyl, a          ; Y=Y/2
        ld      h, a            ; Y=Y/2
        ld      a, ixl          ; X=X/2
        srl     a               ; X=X/2
        ld      ixl, a          ; X=X/2
        ld      l, a            ; X=X/2
        ld      c, 3            ; Colour! 1=yellow,2=Blue,3=RED.
        sla     l               ; calculate screen offset
        srl     h               ; rotate and shift to get correct
        rr      l               ; X and Y offsets.
        srl     h
        rr      l
        srl     h
        rr      l
        and     $03             ; Mask pixel offset colour
        inc     a
        ld      b, a            ; Set up for rotating nibbles
        ld      a, $fc
pset1:  rrca
        rrca
        rrc     c
        rrc     c
        djnz    pset1
        ld      de, $7000       ; Load video offset.
        add     hl,de           ; HL = exact pixel (X,Y) position
        and     (hl)            ; mask with correct colour
        or      c               ; OR Reg A pixel
        ld      (hl),a          ; Set correct colour pixel.

        jp      random
```

Rotating Serprinki's triangle. Animated graphics.



It isn't flash, and can be code-sized optimised greatly, and I'm getting to the point where I am starting to be over this book and to move on to other things "Oh look, a shiny thing!". So, the following listing has very limited comments – not what I was originally planning. Anyhow, by performing dodgy loops from 0 to 127 on (X,0), and then 0 to 63 on (127,Y), and then 127 to 0 on (X, 63), and then finally 63 to 0 on (0,Y), we cover the entire boundary of the mode(1) screen. If we then plot the three points of the triangle to these outside boundary loops, we come out with a rough rotating real-time calculated triangle. SIN(), COS() and TAN() are routines in ROM , and using these may be quick enough to draw a proper real-time calculated object to rotate nice and clear and awesome looking, but… the author is not quite there yet! Perhaps this might be available in Book 2. But I highly doubt it. Of course, the best method is to use pre-calculated values in a big lookup table, or use a short routine at the start of your program to calculate these tables and to auto-generate the SIN() or COS() table upon initial execution. Then you can have proper circles and nice fancy smooth sine rhythm's.

```
ORG $8000
        ld      a,8             ; mode (1)
        ld      ($6800),a
loop00: ld      hl, $7000       ; MODE(1) CLS
        ld      de, $7001
        ld      a, 0
        ld      (hl), a
        ld      (de), a
        ld      bc, 2048
        ldir
        ld      hl, $9000       ; Screen Buffer
        ld      de, $9001       ; At $9000
        ld      a, 0
        ld      (hl), a
        ld      bc, 1048
        ldir
        ld      bc, 1048
        ldir
go:     ld      ix, 63
        ld      iy, 63
        ld      a, 0            ; part 0
        ld      (x1), a
        ld      (y1), a
        ld      (y2), a
        ld      a, 127
        ld      (x2), a
        ld      a, 63
        ld      (x3), a
        ld      (y3), a
        ld      b, 64
l1:     push    bc
        call    chaos
        ld      a, (x2)
        dec     a
        ld      (x2), a
        ld      a, (y1)
        inc     a
        ld      (y1), a
        ld      a, (x3)
        inc     a
        ld      (x3), a
        pop     bc
        djnz    l1
        ld      b, 64           ; part 1
l2:     push    bc
        call    chaos
        ld      a, (x2)
        dec     a
        ld      (x2), a
        ld      a, (x1)
        inc     a
        ld      (x1), a
        ld      a, (y3)
        dec     a
        ld      (y3), a
        pop     bc
        djnz    l2
        ld      a, 63           ; part 2
        ld      (x1), a
        ld      (y1), a
        ld      a, 0
        ld      (x2), a
        ld      (y2), a
        ld      (y3), a
        ld      a, 127
        ld      (x3), a
        ld      b, 62
l3:     push    bc
        call    chaos
        ld      a, (y2)
        inc     a
        ld      (y2), a
        ld      a, (x1)
        inc     a
        ld      (x1), a
        ld      a, (x3)
        dec     a
        ld      (x3), a
        pop     bc
        djnz    l3
        ld      a, 127          ; part 3
        ld      (x1), a
        ld      a, 63
        ld      (y1), a
        ld      (y2), a
        ld      (x3), a
        ld      a, 0
        ld      (x2), a
        ld      (y3), a
        ld      b, 62
l4:     push    bc
        call    chaos
        ld      a, (x2)
        inc     a
        ld      (x2), a
        ld      a, (y1)
        dec     a
        ld      (y1), a
        ld      a, (x3)
        dec     a
        ld      (x3), a
        pop     bc
        djnz    l4
        ld      a, 127          ; part 4
        ld      (x1), a
        ld      a, 0
        ld      (y1), a
        ld      (x3), a
        ld      (y3), a
        ld      a, 63
        ld      (x2), a
        ld      (y2), a
        ld      b, 62
l5:     push    bc
        call    chaos
        ld      a, (x2)
        inc     a
        ld      (x2), a
        ld      a, (x1)
        dec     a
        ld      (x1), a
        ld      a, (y3)
        inc     a
        ld      (y3), a
        pop     bc
        djnz    l5
        ld      a, 0            ; part 5
        ld      (y1), a
```

```
        ld      (x3), a
        ld      a, 127
        ld      (x2), a
        ld      a, 63
        ld      (x1), a
        ld      (y2), a
        ld      (y3), a
        ld      b, 62
l6:     push    bc
        call    chaos
        ld      a, (y2)
        inc     a
        ld      (y2), a
        ld      a, (x1)
        dec     a
        ld      (x1), a
        ld      a, (x3)
        inc     a
        ld      (x3), a
        pop     bc
        djnz    l6
h3:     jp h3
chaos:  ld      b, 20                   ; Will loop 500+ times
        ld      d, 2
chaos2: push    bc
        push    de
rand1 equ $+1
         ld a,$A6
rand2 equ $+1
         ld hl,$8243
         inc l
         dec h
         add a,(hl)
         ld (rand2),hl
         rlca
         rlca
         sub h
         add a,l
         ld (rand1),a
        cp      85
        jr      c, next2  ; 0,0       -JMP BELOW
        cp      170
        jr      nc, next  ;-JMP ABOVE
        ld      bc, (x2)
        add     ix, bc                  ; 128, 0
        ld      bc, (y2)
        add     iy, bc
        jp      calc
next:   ld      bc, (x3)  ; (64, 63)
        add     ix, bc
        ld      bc, (y3)
        add     iy, bc
        jp      calc
next2:  ld      bc, (x1)  ; (0,0)
        add     ix, bc
        ld      bc, (y1)
        add     iy, bc
calc:   ld      a, iyl                  ; DIV IY /2
        srl     a
        ld      iyl, a
        ld      h, a
        ld      a, ixl                  ; DIV IX /2
        srl     a
        ld      ixl, a
        ld      l, a
        ld      c, 2
        sla     l          ; calculate screen offset
        srl     h
        rr      l
        srl     h
        rr      l
        srl     h
        rr      l
        and     $03           ; pixel offset
        inc     a
        ld      b,a
        ld      a,$fc
pset1:  rrca
        rrca
        rrc     c
        rrc     c
        djnz pset1
        ld      de, $7800
        add     hl,de
        and     (hl)
        or      c
        ld      (hl),a
        pop     de
        pop     bc
        djnz    chaos2
        dec     d
        jp      nz, chaos2
        ld      hl, $7800               ; BLIT FROM $7800
BUffer to screen
        ld      de, $7000
        ld      bc, 2048
        ldir
        ld      hl, $9000               ; MODE(1) CLS
BUFFER at $9000
        ld      de, $7800
        ld      bc, 2048
        ldir
        ret
x1      defw    0
y1      defw    0
x2      defw    127
y2      defw    0
x3      defw    63
y3      defw    63
```

This quickie was done up for the 2020 ten-liner BASIC competition. After submitting it, thought I'd have a go at converting it to assembly. Again, it is too lengthy for this book, but have shoved it in regardless. I had planned to have it fully commented, but again, it isn't going to happen.

```
0 B=29182:DIMA(13):FORI=1TO13:A(I)=28672+RND(14)*32+RND(32):NEXT
1 FORI=RND(3)TORND(3)+4:FORJ=1TORND(8):POKEA(I),RND(63)+64
2 NEXTJ,I
3 FORI=1TO7:IFA(I)<B,POKEA(I),RND(63)+64:A(I)=A(I)+32:NEXT:GOTO5
4 A(I)=28671+RND(32):NEXT
5 FORI=8TO12:IFA(I)<B,POKEA(I),96:A(I)=A(I)+32:NEXT:GOTO1
6 A(I)=28671+RND(32):NEXT:GOTO3
```

Line 0 : Set fall off screen location. Set array, clear screen. Set 13 entrys of array to be random locations on the screen.

Line 1 : For the first random amount of entries, display a random amount of random characters on screen. This is the initial effect when The Matrix characters appear dripping down the screen.

Line 2: Can not fit on Line 1 unfortunately. And can not find enough space for a CLS.. ONLY ONE SINGLE MORE CHARACTER IS NEEDED!. I gave up looking further.

Line 3 : For the first seven entries that are on the screen, pick a random character and display it. Increase the location on the screen by one line down. And do this 7 times for each entry. If the location is on the screen then skip line 4.

Line 4 : This line will only be reached if a single entry's display location has dropped off / fallen off the screen. So select a new screen location.

Line 5 : For the next six array entries if they are still on the screen, blank them out - make them light green space for VZ300. And increase down to the following line. Do this six times, then jump back to line 1.

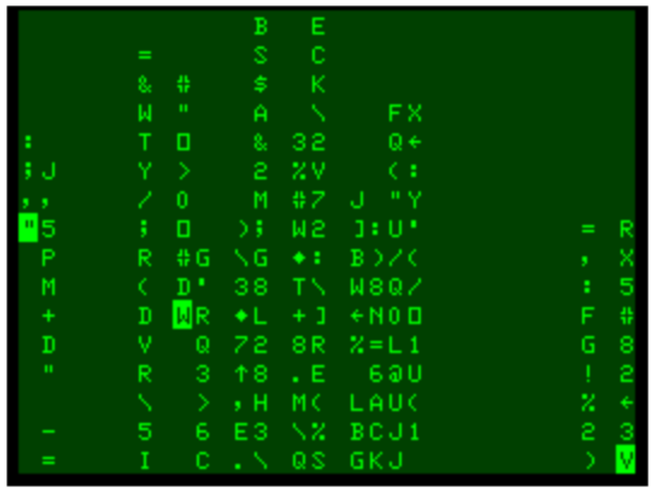
Line 6 : For each array entry that has fallen off the screen, pick a new screen location. Goto 3 since there isn't the need to add the fancy char display and all it does is add a small un-required delay.

```
; MATRIX VZ200
; =============
;
        ORG     $8000

        ld      hl, $7000   ; CLEAR SCREEN
        ld      de, $7001
        ld      (hl), 32
        ld      bc, 2048
        ldir
        di
line0:  call    load_1              ; BASIC Line 0
        ld      (a1), hl
        call    load_1
        ld      (a2), hl
        call    load_1
        ld      (a3), hl
        call    load_1
        ld      (a4), hl
        call    load_1
        ld      (a5), hl
        call    load_1
        ld      (a6), hl
        call    load_1
        ld      (a7), hl
        call    load_1
        ld      (a8), hl
        call    load_1
        ld      (a9), hl
        call    load_1
        ld      (a10), hl
        call    load_1
        ld      (a11), hl
        call    load_1
        ld      (a12), hl
        call    load_1
        ld      (a13), hl
        call    load_1
        ld      (a14), hl
        jp      here2

load_1: ld      hl, $7000
        ld      d, 0
        call    random
        ld      e, a
        add     hl, de
        call    random
        ld      D, 0
        ld      e, a
        add     hl, de
        ret
here2:
line1:  ld      b,  55              ; BASIC Line 1 & 2
loop1b: call    random63_2          ; select random char to display.
        ld      de, (a1)
        ld      (de), a
        djnz    loop1b
        ld      b, 55
loop2b: call    random63_2
        ld      de, (a2)
        ld      (de), a
        djnz    loop2b
        ld      b, 85
loop3b: call    random63_2
        ld      de, (a3)
        ld      (de), a
        djnz    loop3b
        ld      b, 120
loop4b: call    random63_2
        ld      de, (a4)
        ld      (de), a
        djnz    loop4b
        ld      b, 50
loop5b: call    random63_2
        ld      de, (a5)
        ld      (de), a
        djnz    loop5b
        ld      b, 5
loop6b: call    random63_2
        ld      de, (a6)
        ld      (de), a
        djnz    loop6b
        ld      b, 50
loop7b: call    random63_2
        ld      de, (a7)
        ld      (de), a
        djnz    loop7b
line3:  ld      hl, (a1)      ; BASIC Line 3
        ld      ix, a1
        ld      de, 29182
        rst     $18
        jr      c, loop8
        call    line4
        jp      line3a
loop8:  call    random63_1
        ld      (a1), hl
;       jp      line5

line3a: ld      hl, (a2)
        ld      ix, a2
        ld      de, 29182
        rst     $18
        jr      c, loop9
        call    line4
        jp      line3b
loop9:  call    random63_1
        ld      (a2), hl
;       jp      line5
line3b: ld      hl, (a3)
        ld      ix, a3
        ld      de, 29182
        rst     $18
        jr      c, loop10
        call    line4
        jp      line3c
loop10: call    random63_1
        ld      (a3), hl
;       jp      line5
line3c: ld      hl, (a4)
        ld      ix, a4
        ld      de, 29182
        rst     $18
        jr      c, loop11
        call    line4
        jp      line3d
loop11: call    random63_1
        ld      (a4), hl
;       jp      line5
line3d: ld      hl, (a5)
        ld      ix, a5
        ld      de, 29182
        rst     $18
        jr      c, loop12
        call    line4
        jp      line3e
loop12: call    random63_1
        ld      (a5), hl
;       jp      line5

line3e: ld      hl, (a6)
        ld      ix, a6
        ld      de, 29182
        rst     $18
        jr      c, loop13
        call    line4
        jp      line3f
loop13: call    random63_1
        ld      (a6), hl
;       jp      line5

line3f: ld      hl, (a7)
        ld      ix, a7
        ld      de, 29182
        rst     $18
        jr      c, loop14
        call    line4
        jp      line3g
loop14: call    random63_1
```

```
        ld      (a7), hl
        jp      line5
line3g: jp      line5
line4:                          ; BASIC Line 4
loop15: call    random          ; POSITION = 28672 + rnd( 0-
255 )
        ld      b, 0
        ld      c, a
        ld      hl, 28672
        add     hl, bc
        ld      (ix), l
        ld      (ix+1), h
        ret
line5:  ld      hl, (a8)        ; BASIC Line 5
        ld      ix, a8
        ld      de, 29182
        rst     $18
        jr      c, loop5a1
        call    line6
        jp      line5b1
loop5a1:ld (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a8), hl
line5b1:ld  hl, (a9)
        ld      ix, a9
        ld      de, 29182
        rst     $18
        jr      c, loop5b1
        call    line6
        jp      line5c
loop5b1:ld (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a9), hl
line5c: ld      hl, (a10)
        ld      ix, a10
        ld      de, 29182
        rst     $18
        jr      c, loop5c
        call    line6
        jp      line5d
loop5c: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a10), hl
line5d: ld      hl, (a11)
        ld      ix, a11
        ld      de, 29182
        rst     $18
        jr      c, loop5d
        call    line6
        jp      line5e
loop5d: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a11), hl
line5e: ld      hl, (a12)
        ld      ix, a12
        ld      de, 29182
        rst     $18
        jr      c, loop5e
        call    line6
        jp      line5f
loop5e: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a12), hl
line5f: ld      hl, (a13)
        ld      ix, a13
        ld      de, 29182
        rst     $18
        jr      c, loop5f
        call    line6
        jp      line5g
loop5f: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a13), hl
line5g: ld      hl, (a14)
        ld      ix, a14
        ld      de, 29182
        rst     $18
        jr      c, loop5g
        call    line6
        jp      line5h
loop5g: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a14), hl
line5h: ld      hl, (a15)
        ld      ix, a15
        ld      de, 29182
        rst     $18
        jr      c, loop5h
        call    line6
        jp      line5i
loop5h: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a15), hl
line5i: ld      hl, (a16)
        ld      ix, a16
        ld      de, 29182
        rst     $18
        jr      c, loop5i
        call    line6
        jp      line5j
loop5i: ld      (hl), 32
        ld      e, 32
        ld      d, 0
        add     hl, de
        ld      (a16), hl
line5j: jp      line1
line6:                          ; BASIC Line 6
loop17: call    random32
        ld      hl, 28671
        add     a, l
        ld      l, a
        ld      (ix), l
        ld      (ix+1), h
        ret
random: push hl
        push bc
        push de
        ld      hl,(seed1)
        ld      b,h
        ld      c,l
        add     hl,hl
        add     hl,hl
        inc     l
        add     hl,bc
        ld      (seed1),hl
        ld      hl,(seed2)
        add     hl,hl
        sbc     a,a
        and     %00101101
        xor     l
        ld      l,a
        ld      (seed2),hl
        add     hl,bc
        ld      a, l
        pop     de
        pop     bc
        pop     hl
        ret
random32:push   hl      ; 0-32 ONLY.   Result in A.
        push    bc
        push    de
r_loop0: ld     hl,(seed3)
        ld      b,h
        ld      c,l
```

```
                add     hl,hl
                add     hl,hl
                inc     l
                add     hl,bc
                ld      (seed3),hl
                ld      hl,(seed4)
                add     hl,hl
                sbc     a,a
                and     %00101101
                xor     l
                ld      l,a
                ld      (seed4),hl
                add     hl,bc
                ld      a, l
                cp      32
                jr      nc, r_loop0
                pop     de
                pop     bc
                pop     hl
                ret


random63_1:                             ; 0-63 ONLY.  Result in A.
                push    bc
                push    de
                push    hl
r_loop1:        ld      hl,(seed3)
                ld      b,h
                ld      c,l
                add     hl,hl
                add     hl,hl
                inc     l
                add     hl,bc
                ld      (seed3),hl
                ld      hl,(seed4)
                add     hl,hl
                sbc     a,a
                and     %00101101
                xor     l
                ld      l,a
                ld      (seed4),hl
                add     hl,bc
                ld      a, l
                cp      63
                jr      nc, r_loop1
        pop     hl
                ld      (hl), a
                ld      e, 32
                ld      d, 0
                add     hl, de
                pop     de
                pop     bc
                ret

random63_2:                             ; 0-63 ONLY.  A = A + 64.  Result in A.
                push    hl
                push    bc
                 push   de
r_loop2:        ld      hl,(seed3)
                ld      b,h
                ld      c,l
                add     hl,hl
                add     hl,hl
                inc     l
                add     hl,bc
                ld      (seed3),hl
                ld      hl,(seed4)
                add     hl,hl
                sbc     a,a
                and     %00101101
                xor     l
                ld      l,a
                ld      (seed4),hl
                add     hl,bc
                ld      a, l
                cp      63
                jr      nc, r_loop2
                add     a, 64
                pop     de
                pop     bc
```

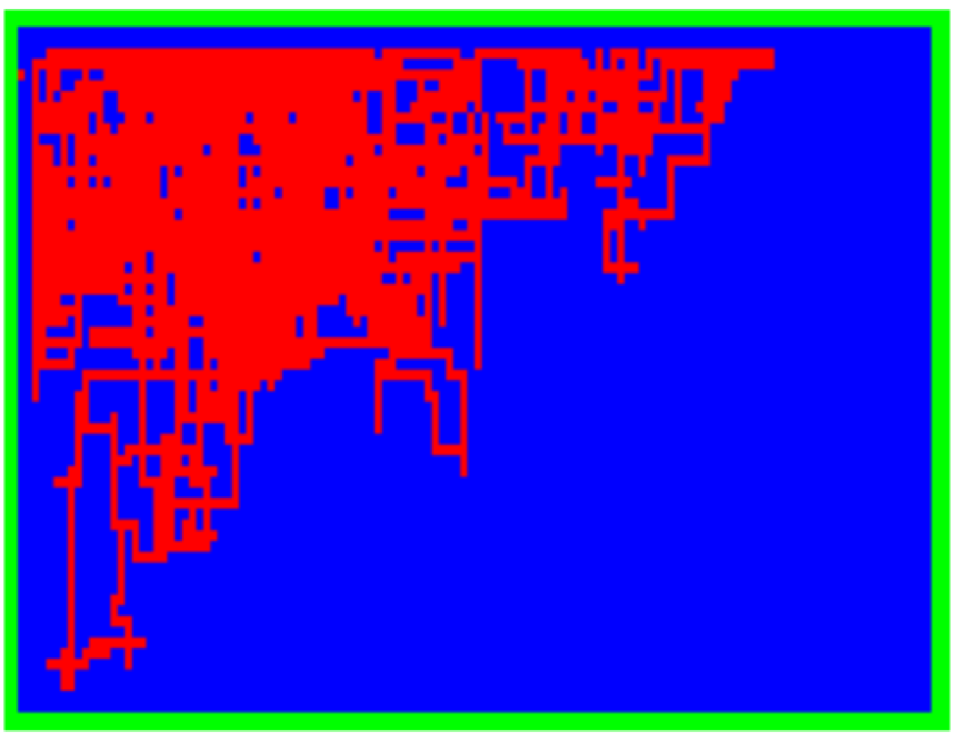
```
                pop     hl
                ret

seed1:          defb    1234
seed2:          defb    5678, 0
seed3:          defb    8765
seed4:          defb    4321, 0

a1:             defw    0
a2:             defw    0
a3:             defw    0
a4:             defw    0
a5:             defw    0
a6:             defw    0
a7:             defw    0
a8:             defw    0
a9:             defw    0
a10:            defw    0
a11:            defw    0
a12:            defw    0
a13:            defw    0
a14:            defw    0
a15:            defw    0
a16:            defw    0
```



SQUIGGLY

Another beaut little listing that creates a half-cool effect that has been floating around in the back of my mind since the early days of learning BASIC. Very incredibly mind-numbingly finger-nail-bitingly slow! So, we'll speed it up a tad. Currently at 131 bytes for the BASIC version and 162 bytes for the asm version. Take away the VZ snapshot header and it is nearly on par. No doubt it can go way smaller, though I've spent an hour on it already, and it will do me for this booklet.

```
10  MODE(1):COLOR4:X=64:Y=32
20  A=RND(4)
30  IFA=1ANDX>3,X=X-1
40  IFA=2ANDX<125,X=X+1
50  IFA=3ANDY<61,Y=Y+1
60  IFA=4ANDX>3,Y=Y-1
70  SET(X,Y):GOTO20
```



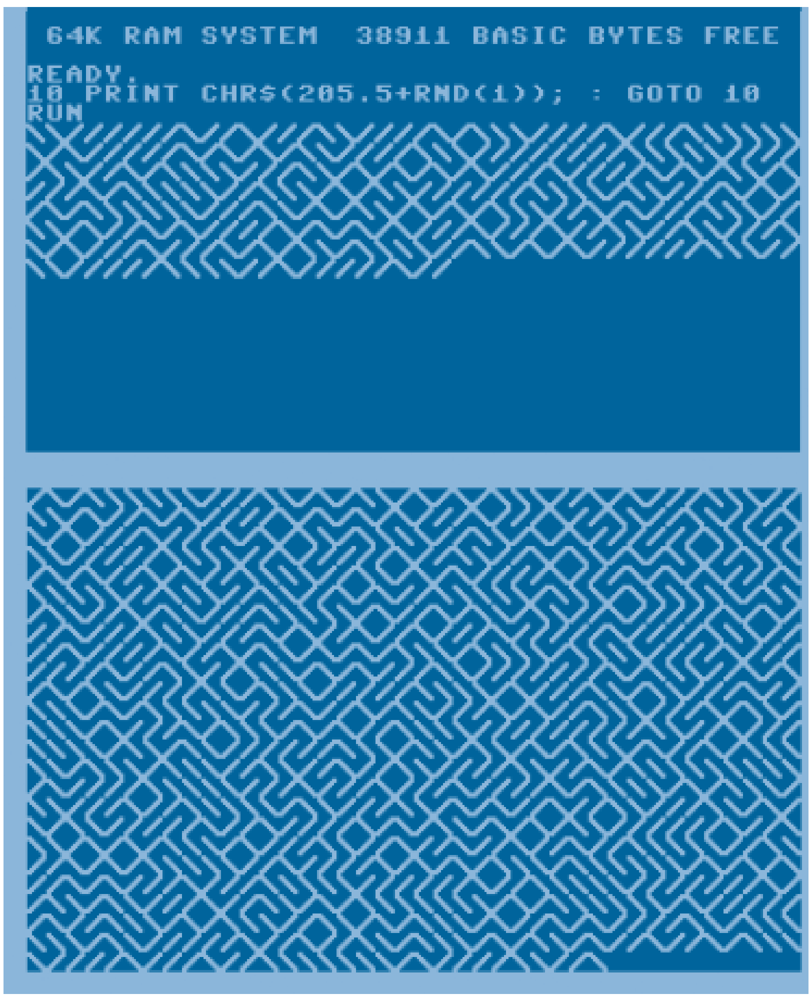
The asm listing is (*) 1000 times quicker than the BASIC listing.
Reference (*) pure guestimation with absolute zilch science behind this fact

```
        ORG     $8000
        ld      a,8             ; mode (1)
        ld      ($6800),a
loop00: ld      hl, $7000       ; clear mode(1) screen
        ld      de, $7001
        xor     a
        ld      (hl), a
        ld      bc, 2048
        ldir
        ld      c, 3            ; Show in RED
        ld      l, 64           ; X = 64
        ld      h, 32           ; y = 32
loop0:  push    hl
rando equ $+1
        ld      hl,23           ; random number generator
        ld      a,r
        ld      d,a
        ld      e,(hl)
        add     hl,de
        add     a,l
        xor     h
        ld      (rando), hl
        ld      a, l            ; a = RND(255)
        pop     hl
        cp      192             ; Is 192 or greater?
        jp      nc, here2       ; Then jump!
        cp      128             ; Is 128 or greater 128 to 191
        jp      nc, here3       ; Then jump!
        cp      64              ; Is 64 to 127?
        jp      nc, here4       ; Then jump!
        inc     l               ; L = X        H = Y
        ld      a, l            ; This all INC or DEC both X,Y
        cp      126             ; Then checks if in bounds.
        jr      nz, here5       ; INC X. If X = 126 then X=126
        dec     l
        jp      here5
here2:  dec     l               ; INC Y
        ld      a, l            ; IF Y = 1 then Y=1.
        cp      1
        jr      nz, here5
        inc     l
        jp      here5
here3:  inc     h               ; H = Y
        ld      a, h
        cp      62              ; IF Y=62 then y=62
        jr      nz, here5
        dec     h
        jp      here5
here4:  dec     h
        ld      a, h
        cp      1               ; IF y=1 then y=1
        jr      nz, here5
        inc     h
here5:
vz_plot1:push   bc      ; c=colour, SET(L,H)  ie: H=Y, L=X
        push    hl      ;
        ld      a, l    ; get x
        sla     l       ; calculate screen offset
        srl     h
        rr      l
        srl     h
        rr      l
        srl     h
        rr      l
        and     $3              ; pixel offset
        inc     a
        ld      b, %11111100
pset3:  rrc     b
        rrc     b
        rrc     c
        rrc     c
        dec     a
        jr      nz, pset3
        ld      de, $7000
        add     hl, de
        ld      a, (hl)
        and     b
        or      c
        ld      (hl), a         ; SET(X,Y) pixel
        pop     hl
        pop     bc
        jp      loop0           ; jump back for another shot
```

MAZE

This particular BASIC to ASM example comes from the PDF book titled "**10 PRINT CHR$(205.5+RND(1)); : GOTO 10**". Yes, that is the name of it. Written by ten fellows, and is available as a free download at https://10print.org/ It is a good read regarding the philosophical side of programming.

It is written towards to Commodore 64, and produces the following cool screen effects.



That's great! But it doesn't work on the VZ. We need some little changes.

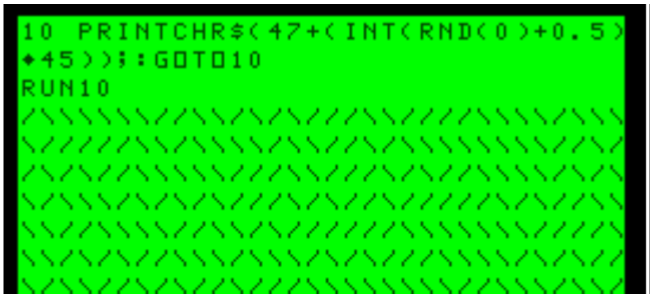


Fig. Example 1

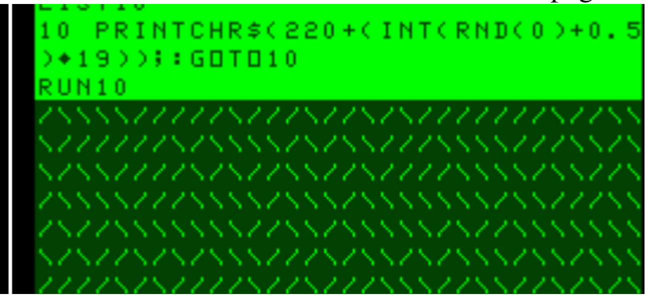


Fig. Example 2

Running these the first thing to notice is how slow they run!. Converting these from BASIC to ASM, the first thing that you will notice is that you will find it hard to view, because it now runs just way too quickly.

First thing is to break it down into its parts. We need to print two characters randomly on the screen, and then just start over again. The printing can be done by the ROM print-single-character routine at $33A after loading register A with whichever random character that we wish to display. Second thing we need to do is to work out a (working) random number generator. As per comment (#5) below, simply using one big iteration is not going to work with the single-character-display. We either need to LOOP and display 64 characters, display a <CR> then start again, OR, we could have one big iteration, have a counter from 1 to 64. And a jump back to the start. If the counter hits 65, then we reset the counter, display a <carriage Return> and jump back to the start. I find the former idea nicer.

A simple random number generator that will work for this example is the following:

```
loopy: LD       A, R
seed equ        $ + 1
       XOR      0
       rrca
       ld       (seed),a
```

Register A will be a rough and ready random number between 0 to 255. The random sequence is fairly poor though, and only after a few hundred iterations, the so called random-number-sequence will start over again.

Next up is the displaying of the alternate characters. In this case, the two slashes. Forward slash and back slash. Characters 47 and 92 and 220 and 239 for the inversed slashes. We need to load these into register A, call the rom routine, and by magic they are displayed on the screen.

Finally we need a 1 to 64 Loop to print 64 characters, display a single carriage return, and start over again, to get around our little display issue mentioned in (#5).

Example 1

```
; Maze. 55 byte version. (PASMO)        Uses characters 47 and 92.    /  and  \
;
       org     $8000
start: ld      b, 64       ; DO-LOOP for 64 chars (2 lines).                                  (#5)
loopy: LD      A, R        ; Grab a value from the R Register. Very poor useless random number.  (#1)
seed equ       $ + 1       ; label pointer for self-modifying code, for further randomness.    (#2)
       XOR     0           ; This XOR's the next byte (the seed) with Reg A. Adds randomness to A.
       rrca                ; Rotates reg A to the right with bit 0 moved to bit 7.             (#3)
       ld      (seed),a    ; Place A into where (seed) is; it will be XOR'd on next loop.
       AND     1           ; Mask the result in reg A to then be a '0' or '1' ?
       ld      a, 47       ; Regardless of the result, pre-set the character to be chr$(47)
       JR      Z, display  ; Jump to display char 47 if the AND mask failed (zero).
       ld      a, 92       ; Therefore now we set the other display character chr$(92)
display:call   $033a       ; Print the character that is in register A.
       djnz    loopy       ; END DO-LOOP. 64 times. Then continues.                           (#4)
       ld      a, $0d      ; Setting up to print a single <CR>                                 (#5)
       call    $033a       ; print <CR>. This forces a next line.
       jp      start       ; And lets start all over again.
end
```

(#1) The R register, also known as the Refresh register. It is a counter that is updated at every instruction, and tends to be somewhat sometimes functionally random. Thusly it can be used as a very poor (next to useless) random number generator. Works ok for this situation where we just want a random zero or one.

(#2) "$" is a literal right-here, and "$+1" is a right-here-but-add-another-byte-onwards.

(#3) RRCA is used to attempt to rotate the bits to further create more randomness.

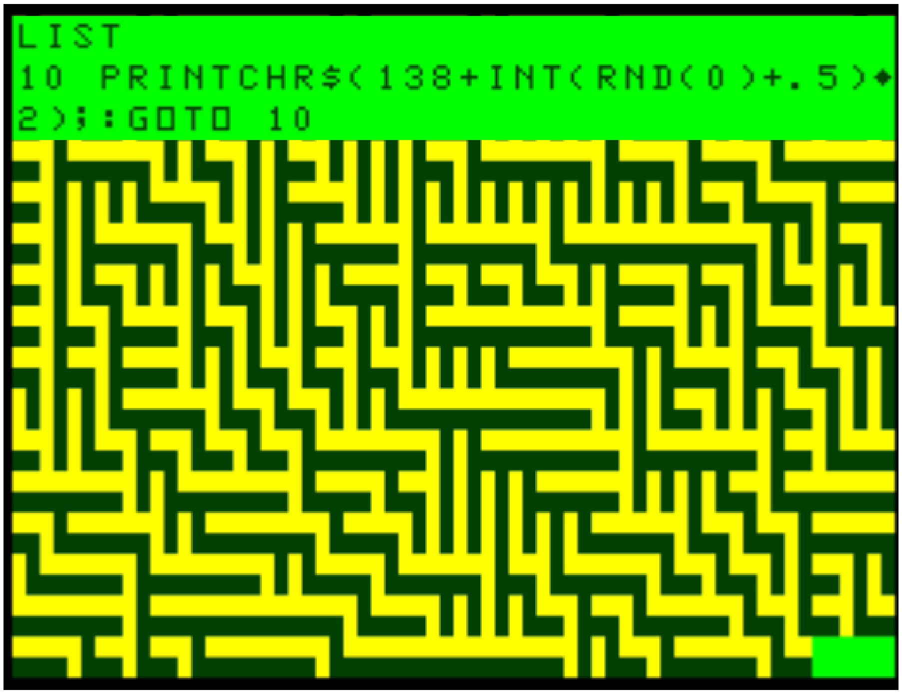
(#4) Instead of a "For I=1to64: do-stuff : NEXT I", in asm we use a loop-decrement-counter, and the quickest/easiest method is the "LD B, <value> // LABEL: Do-stuff // DJNZ LABEL". This will loop forever back to LABEL: whilst B is not zero.

(#5) The CALL $033A is a rom routine t print a single character that is sitting in the A register. Due to the nature of the VZ, only 64 characters can be printed (two lines0 before it requires a NEXT-LINE / CR to get to a new line. We need to print 64 characters, which will fill the first two lines, then display a <CR>, then we can continue on and produce another 64 characters. Without this, the cursor will continue to print everything at the end of line 2 (ie: PRINT@64, <value>)

Example 2 - listing for inversed Slashes demo example.

```
; Maze. 55 byte version. (PASMO)        Uses characters 220 and 239.  INVERSE  /  and  \
;
       org     $8000
start: ld      b, 64       ; DO-LOOP for 64 chars (2 lines).                                  (#5)
loopy: LD      A, R        ; Grab a value from the R Register. Very poor useless random number.  (#1)
seed equ       $ + 1       ; label pointer for self-modifying code, for further randomness.    (#2)
       XOR     0           ; This XOR's the next byte (the seed) with Reg A. Adds randomness to A.
       rrca                ; Rotates reg A to the right with bit 0 moved to bit 7.             (#3)
       ld      (seed),a    ; Place A into where (seed) is; it will be XOR'd on next loop.
       AND     1           ; Mask the result in reg A to then be a '0' or '1' ?
       ld      a, 220      ; Regardless of the result, pre-set the character to be chr$(220)
       JR      Z, display  ; Jump to display char 220 if the AND mask failed (zero).
       ld      a, 239      ; Therefore now we set the other display character chr$(239)
display:call   $033a       ; Print the character that is in register A.
       djnz    loopy       ; END DO-LOOP. 64 times. Then continues.                           (#4)
       ld      a, $0d      ; Setting up to print a single <CR>                                 (#5)
       call    $033a       ; print <CR>. This forces a next line.
       jp      start       ; And lets start all over again.
end
```

Or, by changing the characters, we can turn it into a proper looking maze for the VZ:

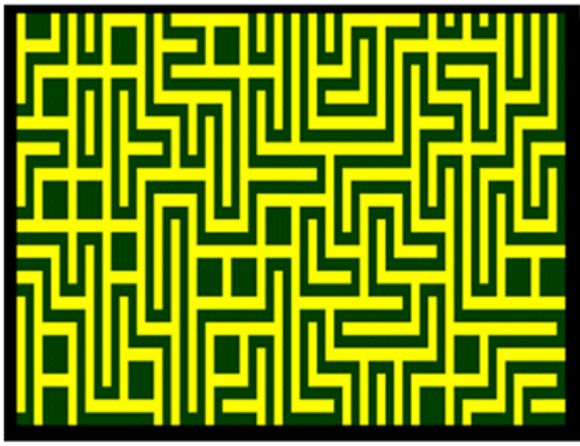


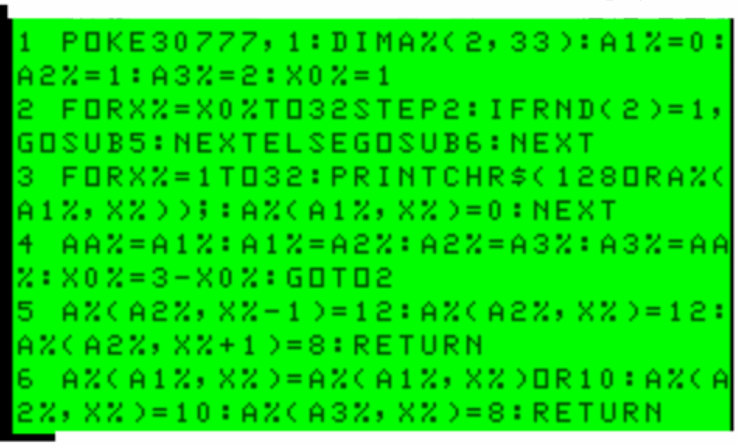
<u>Example 3</u>

```
; Maze. 55 byte version. (PASMO)        Uses block characters 138 and 140.
;
        org     $8000
start: ld       b, 64       ; DO-LOOP for 64 chars (2 lines).                                   (#5)
loopy: LD       A, R        ; Grab a value from the R Register. Very poor useless random number.  (#1)
seed equ        $ + 1       ; label pointer for self-modifying code, for further randomness.      (#2)
        XOR     0           ; This XOR's the next byte (the seed) with Reg A. Adds randomness to A.
        rrca                ; Rotates reg A to the right with bit 0 moved to bit 7.               (#3)
        ld      (seed),a    ; Place A into where (seed) is; it will be XOR'd on next loop.
        AND     1           ; Mask the result in reg A to then be a '0' or '1' ?
        ld      a, 138      ; Regardless of the result, pre-set the character to be chr$(138)
        JR      Z, display ; Jump to display char 138 if the AND mask failed (zero).
        ld      a, 140      ; Therefore now we set the other display character chr$(140)
display:call    $033a       ; Print the character that is in register A.
        djnz    loopy       ; END DO-LOOP. 64 times. Then continues.                              (#4)
        ld      a, $0d      ; Setting up to print a single <CR>                                   (#5)
        call    $033a       ; print <CR>. This forces a next line.
        jp      start       ; And lets start all over again.
end
```

The last example is a maze generator written by Emerson Costa for the MMC1000 which is another Z80 and 6847 computer. Adding in percentage signs to BASIC variables declares the variables to be integers – by default they are declared as floats. It does quicken things up a tad by the interpreter.

We can slightly re-arrange the BASIC listing to remove the two gosubs and place them onto the same callee IF lines – as per the second basic listing below. The removal of the percentage signs was done so just purely to clean up the layout of the code. Percentage signs everywhere seems to add unnecessary viewing complexity. Add them in later if wanting to. One thing to note is that the BASIC listing is around 321 bytes, whilst the assembly near-equivalent is 246 bytes which is based on the second BASIC listing. The difference in speed between either of the two BASIC listings to the ASM is rather mind blowing fast. With a quick copy & paste, and assemble with PASMO or SJASM with RBINARY, you should be viewing this within no time.






```
10 DIM A(2,33):A1=0:A2=1:A3=2:C=1
20 FOR X = C TO 32STEP 2: Z=RND(2)
30 IFZ=1, A(A2,X-1)=12:A(A2,X)=12:A(A2,X+1)=8 : NEXT
40 IFZ=2, A(A1,X)=A(A1,X)OR10:A(A2,X)=10:A(A3,X)=8:NEXT
50 FORX=1TO32:PRINTCHR$(128ORA(A1,X));:A(A1,X)=0:NEXTX
60 AA=A1:A1=A2:A2=A3:A3=AA:C=3-C: GOTO 20
```

```
        ORG     $8000
        LD      DE, A0          ; Fill 4x 33 byte arrays with zero.
        LD      b, 132          ; This saves a lot of unncessary "DEFB 0" below.
        LD      A,0             ; Setting A=0 for a 132 loop of 'LD (DE), 0'
        LDIR                    ; Loop and repeat for 132 times.
        LD      C, 1            ; Reg C = var C
ST0:    LD      B, C            ; Reg B = var X
LOOP1:LD        E, B            ; Reg E = array offset 0-32.
RANDOM2:PUSH            BC              ; Z=RND(2). Output: RND 0-3 in Reg A.
SEED1 EQU       $+1
        ld      hl,1234
        ld      b,h
        ld      c,l
        add     hl,hl
        add     hl,hl
        inc     l
        add     hl,bc
        ld      (SEED1),hl      ; use self mod code to store another random seed
SEED2 EQU       $+1
        ld      hl,5678
        add     hl,hl
        sbc     a,a
        and     %00101101
        xor     l
        ld      l,a
        ld      (SEED4),hl      ; use self mod code to store another random seed
        add     hl,bc
        ld      a, l            ; Reg A = RND(255)
        and     3               ; Reg A = RND(3)
        POP     BC
LINE30:CP       2               ; IF Z=2 THEN
        JR      Z, LINE40       ; goto LINE40
        LD      IX, A2          ; ELSE LINE30.  IX="A2 array"
        LD      D, 0
        ADD     IX, DE          ; DE=array offset from for-to-next
```

```
        LD      (IX), 12                ; A(A2,X)=12
        DEC     IX              ;   x=x-1 --> X-1
        LD      (IX), 12                ; A(A2,X-1)=12
        INC     IX              ;   x=x+1 --> back to X
        INC     IX              ;   x=x+1 --> X+1
        LD      (IX), 8         ; A(A2,X+1)=8
        JP      line30b
LINE40:LD       IX, A1          ; IX=A1 array.
        ADD     IX, DE          ; DE=array offset from for-to-next
        LD      A, (IX)
        OR      10
        LD      (IX), A         ; A(A1,X)=A(A1,X) OR 10
        LD      IX, A2          ; IX=A2 array
        ADD     IX, DE
        LD      (IX), 10                ; A(A2,X)=10
        LD      IX, A3          ; IX=A3 array.
        ADD     IX, DE
        LD      (IX), 8         ; A(A3,X)=8
line30b:INC     B               ; 2x 'INC B' = STEP2 from for-to-next
        INC     B
        LD      A, B            ; comparison for loop
        CP      32              ; If > 32
        JR      C, LOOP1        ; then jump
LINE50:LD       IX, A1          ; set A1 array
        LD      B, 32           ; FORX=1TO32
        PUSH    DE              ; store DE
        LD      DE, $7000 + 480-32 ; Get destination for POKE
Loop2: LD       A, (IX)         ; load IX to OR yellow blocks. Begin of FOR-TO-NEXT loop
        OR      128+16                  ; +16 for POKE blocks
        LD      (DE), A         ; POKE@DE,A1-blocks
        INC     E               ; INC POKE offset
        LD      A, 0
        LD      (IX), A         ; A(A1,X)=0
        INC     IX              ; inc A1 array offset.
        DJNZ    loop2           ; NEXTX.                End of FOR-TO-NEXT loop
        POP     DE              ; restore DE
        LD      A, 13           ; FORCE a <CR> at each offset 32
        CALL    $033A           ; Write out character <CR>
LINE60:LD       B, 33           ; AA = A1
        LD      IX, A0          ; destination
        LD      IY, A1          ; source
        CALL    move            ; move array
        LD      B, 33           ; A1 = A2
        LD      IX, A1          ; destination
        LD      IY, A2          ; source
        CALL    move            ; move array
        LD      B, 33           ; A2 = A3
        LD      IX, A2          ; destination
        LD      IY, A3          ; source
        CALL    move            ; move array
        LD      B, 33           ; A3 = AA
        LD      IX, A3          ; destination
        LD      IY, A0          ; source
        CALL    move            ; move array
        LD      A, 3            ; C=3-c
```

```
        SUB     c
        LD      C, A
        JP      ST0             ; GOTO 20
move:   LD      A, (IY)         ; move array
        LD      (IX), A         ; IY=source
        INC     IX              ; IX=destination.
        INC     IY
        DJNZ    move            ; Loop B number of times.
        ret
        DEFB    0
A0      EQU     $
A1      EQU     $ + 33
A2      EQU     $ + 66
A3      EQU     $ + 99
```

CONCLUSION

None. This is not one of those books!

References

Google. ‘Z80 opcodes’, ‘Z80 flags’, ‘INC IX’, ‘porn’, ‘ten-liner competition’.

VZ200 Technical Reference Manual.

Zen and the art of Metaphysics of Quality applied to VZ BASIC to Assembly Language conversion and other waffle, page 10.

VZ300 Technical Reference Manual.