# 2x2ine

**Issue #7** **June 2019**



ZXzine

# Table of Contents

Published by:
Timothy Swenson
swenson_t@sbcglobal.net
swensont@lanset.com

ZXzine is published as a service to the Sinclair ZX81 community. Writers are invited to submit articles for publication. Readers are invited to submit article ideas.

Created using Open Source Tools:

- OpenOffice
- Scribus
- Gimp
- SZ81
- EightyOne

Copyright 2019
Timothy Swenson

Creative Commons License
- Attribution
- Non-Commercial
- Share-Alike

You are free:

- To copy, distribute, display, and perform the work.

- To make derivitive works.

- To redistribute the work.

**Editorial** ..... **1**

**QDDASM2** ..... **1**

**T/S 2068 ROM Disassembly** ..... **1**

**T/S 2068 and ZX81 ROM routines** ..... **2**

**VisiCalc vs VuCalc** ..... **6**

**zeditor** ..... **7**

**Othello** ..... **8**

**Boriel ZX Basic Compiler** ..... **10**

## Editorial

Hopefully here is something for everyone in this issue. There is a little history in the Vu-Calc vs Visi-Calc article. For general users, there is the review of the zedit editor, probably the only decent editor that I've seen for the ZX81. For programmers there is articles on ROM routines, an update to QDDASM and a review of the Boriel ZX BASIC compiler.

All of these topics are just the things that have interested me lately. As I think about the ZX81, T/S 2068 or Spectrum, little ideas crop up for articles. As I read the different forums, I can get an idea for something interesting to do or pursue and then write an article about it.

If someone has some suggestions for articles, I'd be willing to hear them and see if it is something that I could do.

## QDDASM2

After the initial release of QDDASM I started using it to disassemble the T/S 2068 ROM. I ran into a few issues when doing that, that necessitated changes to QDDASM.

The first issue was that most documenation on ROM's use hexidecimal notation for locations in ROM. I prefered to use decimal, but I did not want to have to convert everything back and forth between decimal and hex. It was fairly easy to do a search and replace with the code to change the decimal output to hex.

The second issue I ran across was that I was missing some Z80 opcodes that were in the T/S 2068 ROM and QDDASM could not handle. So, I had to dig through the Z80 opcodes and include all the ones that I had left out. This did add a lot more lines of code, but it did not appear to affect the speed of the disassembler when running.

To test my code, I found a listing of all Z80 opcode in the package Z80Pack, which is a Z80 emulator and cross assembler. I took the z80ops.asm and assembled it with Pasmo. For some reason Pasmo could not handle a few of the opcodes, so I had to remove them and test with out them. I ran the new binary through QDDASM2 and after some bug fixing, the input file looked pretty much like the output file.

The general usage of QDDASM2 is the same as the first version. All numbers used on the command line must be in decimal. I did not do any work to convert the command line to take hexidecimal.

With QDDASM2, it is ready to be used for other projects and should be able to disassembly the full Z80 op code set. The program should be portable and compile on almost any platform with a decent C compiler.

## T/S 2068 ROM Disassembly

With changes made to QDDASM (making it now version 2), I am able to start work on the disassembly of the T/S 2068 ROM. I am using the Spectrum ROM disassebmly as my guide.

As I go through the Spectrum ROM, I noticed that there is a number of data statements listed at various points. The problem with data statements is that a disassembly has no way to tell if a byte is an opcode or data. I've been running the T/S 2068 ROM through QDDASM2 then comparing it with the Spectrum ROM. Given that the T/S 2068 ROM changes were fairly minimal in the lower ROM, I can find where the data statements should be in the T/S 2068 ROM. I then skip over X number of bytes and restart the disassembly process at that location. It's a bit tedious and it does take time.

So far I have been able to get to the tokens and character set table in the T/S 2068 ROM. I've been putting in labels similar to the Spectrum ROM to make it easy to compare the two.

This process may take a while. If anyone is interested is seeing what I've done so far, please let me know. I'm hoping the T/S 2068 ROM disassembly might be useful to someone.

## T/S 2068 and ZX81 ROM Routines

As I have been looking into doing assembly on the T/S 2068, I started looking into the T/S 2068 ROM routines. The T/S 2068 Technical Manual provides a listing of the ROM routines, but no description. The Technical Manual does detail the routines that are called from the Function Dispatcher. It looks like the details in the routines in the Function Dispatcher are the same for the ROM routines.

I read through the Technical Manual, looking at the routines and seeing which ones could be useful. If I was a little lost, I found some pointers by looking at the Spectrum ROM and seeing how those routines were used. I generally assumed that the ROM calls were fairly close between the systems. I was able to create a list of the routines and write some example assembly code to test the routine. Below is this document.

After reading through the T/S 2068 Technical Manual, I then started thinking about doing the same for the ZX81 ROM. I read though the ZX82 ROM dissassembly (both in book for and in text file I got of the Internet). As I found interesting routines, I wrote them down, read the description and the assembly to understand what the routine did and how to call it. I was able to create a document on ZX81 ROM routines, along with example assembly code, and included it below.

### T/S 2068 ROM Routines

These routines have been tested and provided with example usage. Descritions
are taken from the Timex Sinclair 2068 Technical Manual.

CLS $08EA 2282

Clear entire screen (primary display file).

```
CALL CLS
```

FP2A $3193 12691
Converts the value on the Calculator Stack to an interger and places it in A.

```
LD A,65
CALL STK_A ; Load A onto Calc Stack
CALL FP2A ; Get number from Calc Stack
RST 16 ; print number
RET
```

FP2BC $3160 12640
Converts the value on the Calculator Stack to an integer and places it in BC.

```
LD B,0
LD C,65
CALL STK_BC ; Load BC onto Calc Stack
CALL FP2BC ; Get number from Calc Stack
LD A,C
RST 16 ; print number
RET
```

K_PRIN $2159 8537

PRINT - Selects Channel 2 and processes items in PRINT statement for
output via WRCH. Start of string located at "pointer". String ends with
$0D (13) or ENTER. CH_ADD is a system variable. K_PRIN is expecting double
quotes around the string. Use single quotes for assembler to know this is
a literal string.

```
LD HL,pointer
LD (CH_ADD),HL
CALL PRINT
RET

pointer DEFB '"Print this!"',$0D
```

OUTPUT $31A1 12705
Converts the value on the Calculator Stack to text and prints it out to the
current channel (usually the screen). It does pop the value from the
Calculator Stack.

```
        CALL  CLS    ; Clear the screen
        LD    B,10
        LD    C,10
        CALL  STK_BC       ; Load BC on Calc
Stack
        CALL  OUTPUT       ; Print Calc Stack
        RET                ; return
```

PARP $03F3 1011

Calls a BEEP like command. HL hold the period or tone of the sound.
DE hold the duration of the sound.

```
        LD   HL,40     ; Pitch of 5
        LD   DE,3000 ; Duration
        CALL PARP
        RET
```

PAUSE $1FEB 8171
Takes the value on the Calculator Stack and PAUSEs for that many cycles.
50 cycles equals 1 second.

```
        CALL CLS
      LD   A,65
      RST  16            ; print an A
        LD   BC,200
      CALL STK_BC  ; Load BC to Calculator
Stack
        CALL PAUSE; PAUSE for 4 seconds
        LD   A,66
        RST  16        ; print an B
        RET
```

PAUSE_1 $1FF2 8178

Pause the program. BC is loaded with the number of cycles to
pause. 50 cycles is equal to 1 second. To pause 4 seconds, then
load BC with 200.

```
        CALL CLS
      LD   A,65
      RST  16        ; print an A
        LD   BC,200    ; PAUSE for 4 seconds
        CALL PAUSE_1
        LD   A,66
        RST  16        ; print an B
        RET
```

PLOTBC $263E 9790

Deals with pixel for coordinates in BC (B=Y; C=X)
Processes using P_FLAG
for Inverse and Over attributes. Updates Attribute File and sets COORDS=BC.

```
        LD B,10
        LD C,5
        CALL PLOTBC
```

PR-STR $21DB 8667

Prints a string

```
        LD   DE,string
        LD   BC,12     ; length of string
        CALL PR_STR
        RET

      string DEFB 'name is cool'
```

PUT_BC $1788 6024
Converts the value in BC and puts it out the the current channel
(usually the screen).

```
        CALL  CLS    ; Clear the screen
        LD    B,10
        LD    C,10
        CALL  PUT_BC       ; Print the value in
BC
        RET                ; return
```

PLOT $2635 9781
Pulls two values from the Calculator Stack (16-bit) and uses them to plot X,Y.

```
        LD   A,10
        CALL STK_A; Load A onto Calc Stack
        LD   A,5
        CALL STK_A; Load A onto the Calc Stack
        CALL PLOT
        RET
```

SETAT $05B2 1458

Set print position to value in BC. B = Line number (0-23); C = Column number (0-31)

```
        LD B,10
        LD C,5
        CALL SETAT
        LD A,char
        RST $10
```

STK_A $30E6 12518
Loads the value of A onto the Calculator Stack. Loads 0 to B, A to C and calls STK_BC.

```
        LD   A,10
        CALL STK_A; Load A onto Calc Stack
        LD   A,5
        CALL STK_A; Load A onto the Calc Stack
        CALL PLOT
        RET
```

STK_BC $1788 6024
Takes the value in BC and loads it to the Calculator Stack

```
        CALL  CLS    ; Clear the screen
        LD    B,10
        LD    C,10
        CALL  STK_BC       ; Load BC on Calc Stack
        CALL  OUTPUT       ; Print Calc Stack
        RET            ; return
```

**ZX81 Rom Routines**

CLS $0A2A
Clears the screen

```
        CALL CLS
```

FINDCHAR $07BD
Determines character typed on keyboard. Output from KSCAN is to be in BC.
Returns pointer to character in HL.

KSCAN $02bb
Scans the keyboard and returns a pointer in HL.
wipes out registers

```
        WAIT  CALL  KSCAN                ; get a key from the keyboard
                LD      B,H
                LD      C,L
                LD      D,C
                INC     D
                JP      Z, WAIT          ; If no key pressed, then loop
                CALL  FINDCHAR   ; Translate keyboard result to character
                LD      A,(HL)           ; Put results into reg a
                CP      $76              ; Is character a / (slash)
                JP      Z, ENDD          ; If so, jump to END
                CALL  PRINT          ; Print character
                JP      WAIT
        ENDD
              RET
```

PAUSE
Pauses the system for X cycles (50 cycles per second). 200 is 4 second pause.

```
        CALL  CLS
        LD      A,$08           ; grey square character
        CALL  PRINT          ; print character
        LD      BC,200
        CALL  PAUSE                ; Pause for 4 seconds
        LD      A,$08
        CALL  PRINT          ; print second character
```

```
        RET


PRINT                   $0010
  Prints character stored in A

        LD      A,$08           ; grey square
character
        CALL    PRINT           ; print character

PRINTAT         $08f5
  Moves cursor to postion of Row (B) and Column
(C).

        LD      B,10
        LD      C,10
        CALL    PRINTAT                 ; set cusor to
position 10,10
        LD      A,$08           ; grey square
character
        CALL    PRINT           ; print character


PRINTFP
  Prints the value at the top of the Calculator Stack.

        LD      A,65
        CALL    STKA            ; Put value in A onto
Calculator Stack
        CALL    PRINTFP                 ; print value
on Calculator stack
      RET


PLOT  $0BB2
  Plot can not be called directly.  Two subroutines
have been created for PLOT
  (LPLOT) and UNPLOT (UPLOT).  Load the X
and Y coordinates into BC before
  calling either subroutine.

        LD      B,10    ; Store Y in B
        LD      C,10    ; Store X in C
        CALL    LPLOT

lplot:
        LD      HL,($4030)
        PUSH    HL              ;SAVE t-ADDR
        LD      A,$98           ;AS LONG AS IT'S
LESS THAN $9E SHOULD PLOT.
        LD      ($4030),A
        CALL    PLOT
        POP     HL
        LD      ($4030),HL      ;RESTORE t-ADDR
        RET

uplot:
        LD      HL,($4030)
        PUSH    HL              ;SAVE t-ADDR
        LD      A,$9E           ;I THINK
ANYTHING GREATER THAN $9E SHOULD
UNPLOT
        LD      ($4030),A
        CALL    PLOT
        POP     HL
        LD      ($4030),HL      ;RESTORE t-ADDR
        RET


SCROLL                  $0C0E
  Scrolls the screen up 1 line

        CALL    CLS
        LD      B,10
        LD      C,10
        CALL    PRINTAT                 ; Set cursor to
10,10
        LD      A,$08           ; grey square
character
        CALL    PRINT           ; print character
        LD      BC,200
        CALL    PAUSE                   ; Pause for 4
seconds
        CALL    SCROLL                  ; SCROLL
        RET

STKA            $151D
  Puts value in A onto the Calculator Stack.

        LD      A,65
        CALL    STKA            ; Put value in A onto
Calculator Stack
        CALL    PRINTFP                 ; print value
on Calculator stack
      RET


STKBC                   $1520
  Pusts the value in BC onto the Calculator Stack.
```

```
        LD      B,10
        LD      C,10
        CALL  STKBC                 ; Put value in
A onto Calculator Stack
        CALL  PRINTFP               ; print value
on Calculator stack
      RET
```

## VuCalc vs VisiCalc

VuCalc on the ZX81 is a fairly simple spreadsheet program. Given that most ZX81's had only 16K of memory, that is a small space to work with for a spreadsheet, for both the program and space for the data.

I understand the limited spreadsheet size (26 x 36 for 936 cells). Only 40 formula's are allowed. Each formula can have up to 32 characters. There is no built in functions.

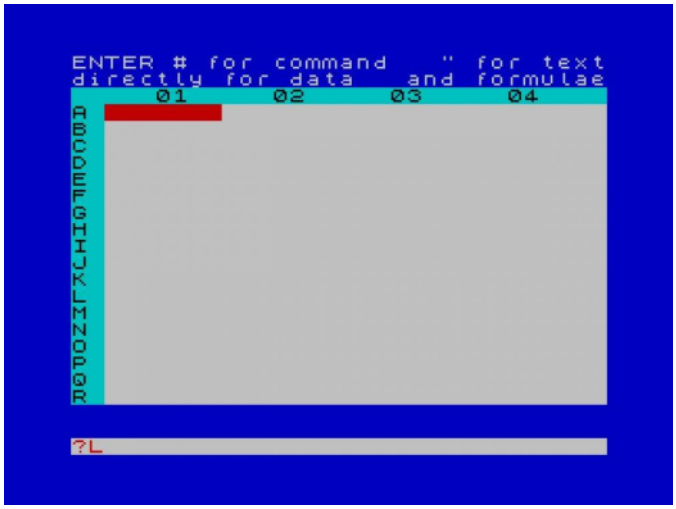
When I was recently tinkering with the T/S 2068 and thought I would look into VuCalc on that platform. The T/S 2068 was sold as a 72K system,




with 24K of ROM and 48K of RAM. That is 3 times the memory of the standard ZX81 configuration. That is plenty of room for a decent spreadsheet.

VuCalc on the T/S 2068 does allow for 2500 cells (50 x 50 grid). More characters are allowed per cell (up to 28). The only function is SUM, that does a summation of the data in a cell group. I was really expecting more. And why is that?

I first learned how to do spreadsheets on an Apple II with VisiCalc. Visicalc was the first electronic speadsheet and became the killer application for the Apple II. Anyone that worked with numbers saw how game changing VisCalc was.




The Apple II was also an 8-bit system like the T/S 2068 and came with a range of memory options. Visicalc took up 20K and required 32K of memory. The most Apple II's were running with 48K, just like the T/S 2068.

VisiCalc had a number of functions that really make it powerful:

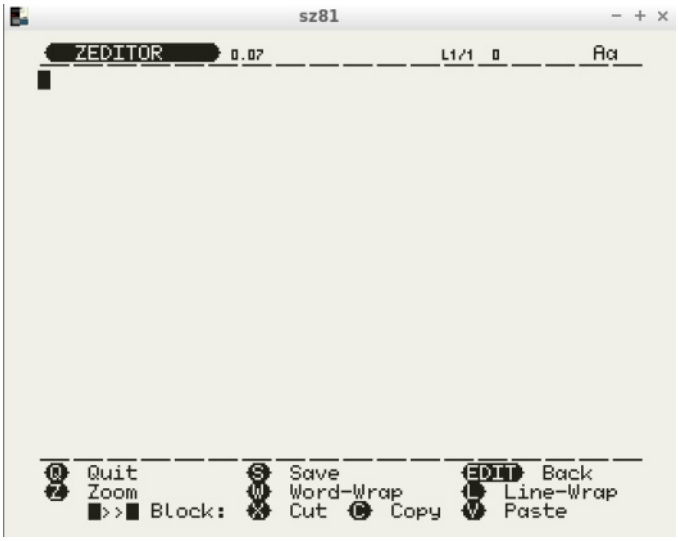
@SUM - Summation
@MIN - Return the minimum value from a range
@MAX - Return the maximum value from a range
@COUNT - Count the number of values in a range
@AVG - Calculate an average for value in a range
@LOOKUP - Look up a value in a one-call range
@SIN - Mathmatical formula

If was done for the Apple II in 32K, why could it not be done for the T/S 2068 with 48K? I think it was the market. The Apple II cost was at least a thousand dollars. VisiCalc was retailing for just under $100. VisCalc cost half what the T/S 2068 was selling for when it was released ($199.99). VuCalc was selling for about $10. I don't think that PSion wanted to invest the funds to make Vu-Calc more powerful. I don't think that they expected T/S 2068 or Spectrum users to have the same requirements as the VisiCalc users.

## zeditor

I recently came across Zeditor, a high-res word processor for ZX81. It is described as a "Unicode Text Editor for the ZX81."

I've seen a few word processors for the ZX81, but they only seem to have a passing resemblence to word processors (or even a text editor). Zeditor looks like a real word processor and works like a real word processor.




Since it is high-res, it supports lower case and upper case letters. It also has word wrap.

Upon loading the .P file, Zeditor starts up and is ready for typing. Across the top of the display is an indicator of how many lines are in the editor and what line number the cursor is on. It is displayed as "1/14", which means that there are 14 lines and the cursor is on the first. To the right of the lines display is a number that indicates the number of characters in the document.

At the bottom of the screen is a reminder that to get to the menu, hit the Edit key (Shift 1). To change the character mode, hit the Graphics key (Shift 9). To get to the command menu, hit Shift-1.

The command menu has a number of commands:

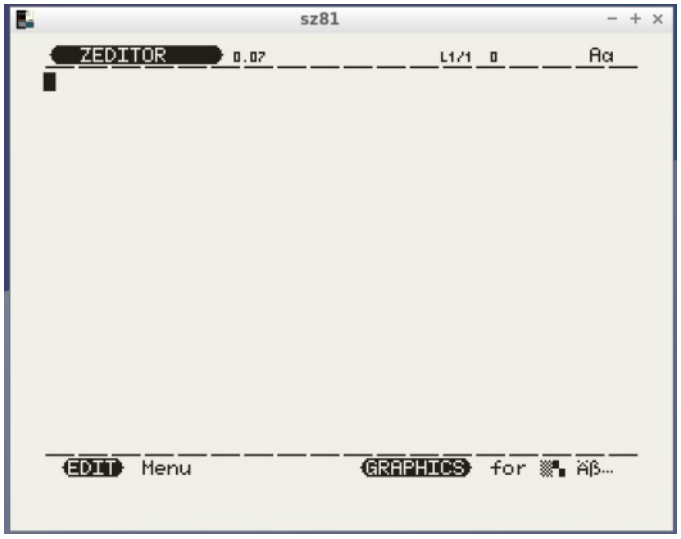
Q - Quit. This exits the editor and goes back to Basic.

Z - Zoom. This changes the font size. The normal font size provides 48 characters across. Changing to the smaller font size the text can be 72 characters across. The smaller font size is harder to see, but it is mostly readable.

S - Save. This saves the text to a REM statement and exits to BASIC so that the text can be saved to a file.

The next set of operations are block based. The user defines a block of text and a number of operations can be used on that block.

W - Word Wrap. Sets the right margin for word wrapping.
L - Line Wrap. Undoes the work of word wrappping.
X - Cut. Cut the block of text and store it in a buffer.
C - Copy. Copy the block of text and store it in a buffer.
V - Paste. This pulls text from the buffer and puts it into the document.

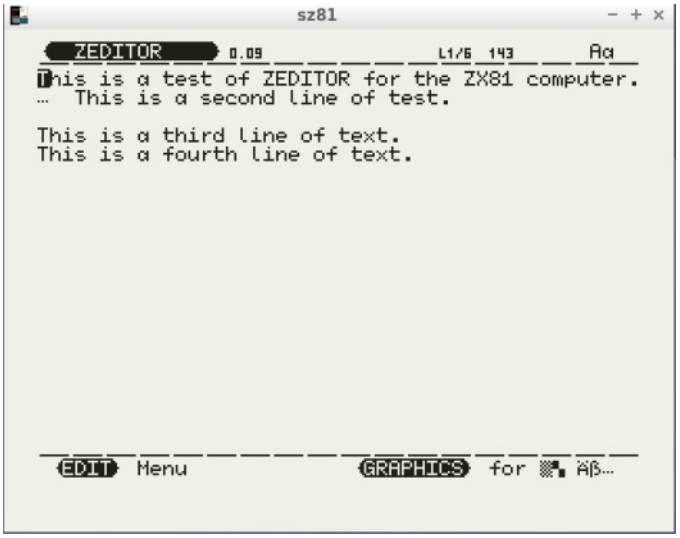



Zeditor stores the text in higher memory. To save the text, it is moved into a REM statement in lower memory where it can be saved. The program calls SAVE to store the document and then re-enters Zeditor, by loading the word processor back into memory. When saving, make sure to rename use a name for the document and not use the original ZEDIT.P file name.

Later, to bring the document up, just load the file to

where the document was saved. It will load and then automatically load Zeditor. So, after use, there will be a number of .P files that are the documents.

There is a printer driver for ZEDIT, zedprint2.p. When done with ZEDIT, use the menu to Quit. Now type in LOAD "zedprint2" to load the printer driver. It will automatically send the text to the printer.




Since ZEDIT stores the text in memory in clear text and that text remains in memory, it is possible to use ZEDIT with other programs. A version of Campus Lisp for the ZX81 has been set up to use ZEDIT for editing Lisp programs.

With sz81:

The way that sz81 works, when it loads a program from the LOAD command, it expects the name of the file to be in all uppercase. When extracted from the zip file, Zeditor's file name is "ZEDITOR.p", but I had to change it to "ZEDITOR.P" for sz81 to load it via LOAD.

When a file is saved via the SAVE command, it will be saved as all lower case. Since the document files can be load directly via the sz81 GUI, there is no issue with their case.

When printing with ZEDIT, sz81 emulates the printer by saving a .pbm file to the .sz81/prtout directory. Once ZEDPRINT2.P is loaded, it will copy document to the printer, which will be a graphics file on disk.

To get the text out of ZEDIT, when quiting from ZEDIT, it will say that the text is a address 32768 and it is X bytes long. With sz81, there is version of the SAVE command that will save a block of memory to disk. Type SAVE like this:

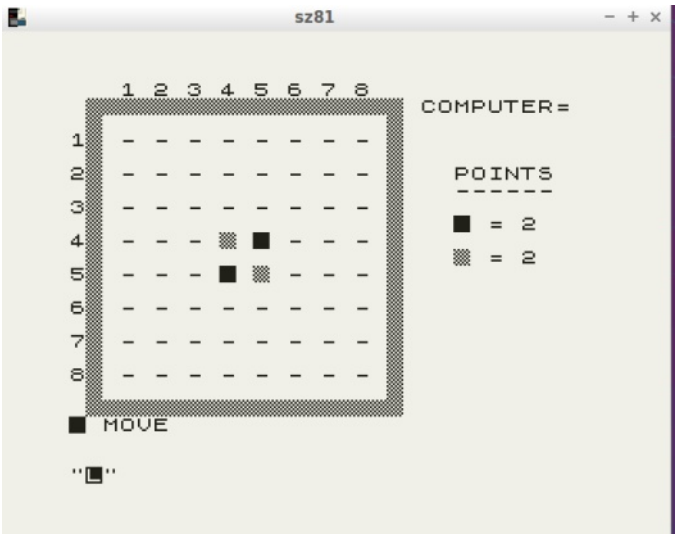
SAVE "FILE.TXT;32678,XX" where XX is the length in bytes. The file "FILE.TXT" will contain the text from the document.

## Othello

I've been looking at learning more about Othello, including reading some books on how to get better at playing the game. Othello (or Reversi) is a common game on computers. For the ZX81 there were four commercial versions that were released (well, four that I could find on archive sites). This is a short review of the four.

### Othello from CDS MicroSystems

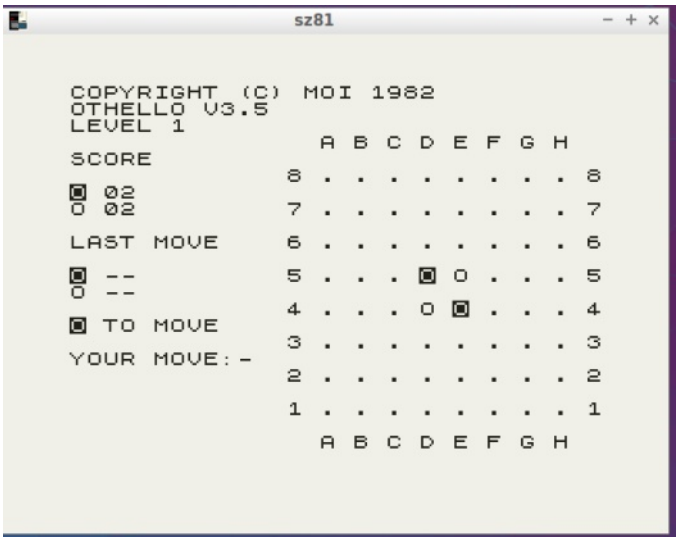
This program gives the options of play the computer, play a person, watch the computer or replay last game. There is a fast mode to speed up the play. The board mostly made up of graphic blocks both black and gray. The spaces on the board are designated in single digit numbers, with the left side being Y and the top being X. The space in entered in Y first and then X. Also on the screen is the computers last move and "points" or a count of the number of pieces that each player has. When the board was full and the game over, the



program did not recognize that condition and asked the player for the next move. Entering any move results in "ERROR TRY AGAIN".

**Othello from Moi**

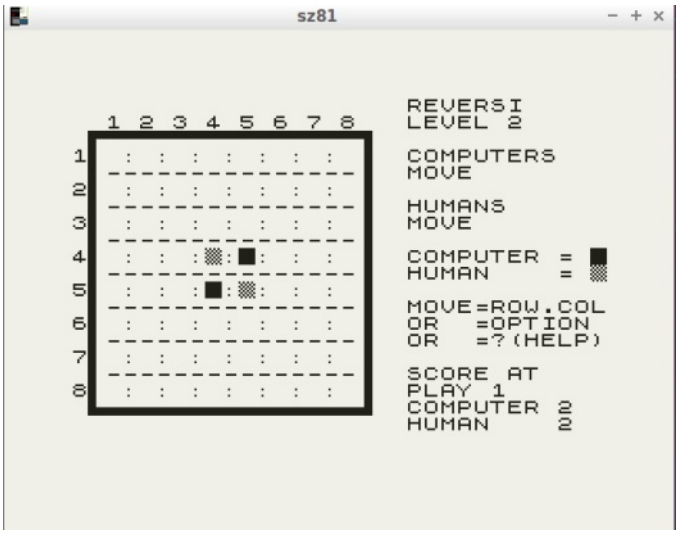
The game was released in 1982 and the program shows the version as 3.5. This game was also released under the Sinclair brand. The options are two person game, you start, zx81 starts, play sample game, or change start position. The board uses latters for the X axis and numbers for the Y axis. Entry is letter and then number. The board is pure text with O and inverse O for the pieces. The




game has an annoying habit of redrawing the whole screen between moves. It can really get distracting while playing. Also on the screen is a listing of the last two moves. There is also a "score" or count of the number of black and white pieces on the board. The game is quick to decide its next move. At the end of the game, when I was the last to move, it quickly ended the game and want to the beginning menu. It did not pause and let me know that I had won. It was not the strongest of players.

**Reversi - Artic**

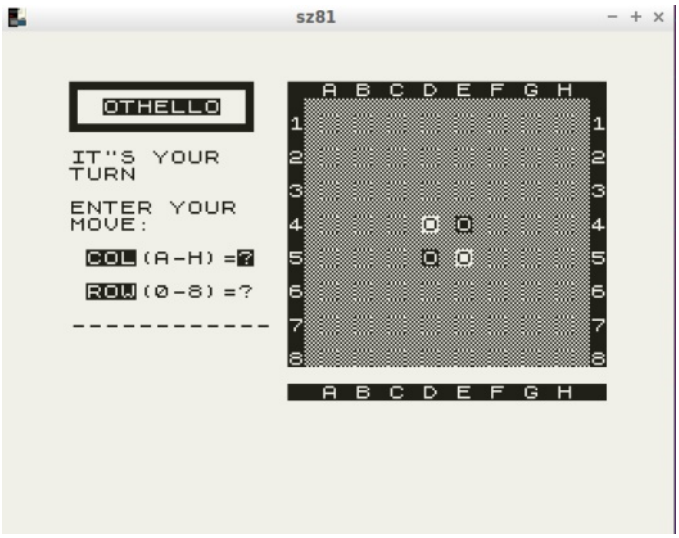
This game allows for the player to choose a skill level (1-4) and then choose if they want to go first. The board is a combination of graphic and text characters. The board has numbers on both X the and Y axis. A move is entered as X.Y, where there must be a period between the numbers. Invalid




moves are rejected by the program. As the computer is deciding its next move, a little dot appears on the board, probably marking each possible move that it is evaluating. It is a little slow to decide the next move. Instead of entering a move, the player can enter "?" to get a menu. The game also has the usual "Score" display for the piece count. It also keeps track of the turn but lists it as "Play". The moves it made were not great. It mised some obvious good moved, but in the end it won by 4 pieces.

**Othello by C Panayi & P.E. Canter Vortex Software**

The options for this game were, play it's best or not, fast or slow, player to be black or white and player to be first or second. This is the only one of the 4 that was written completely in Basic. The board is shows a just gray squares, so it is sort of hard to see exactly where the playing spaces are. Letters are the X axis across the top and number are the Y axis



down the side. The program shows "my last move", so it is easy to keep track of the game. Unlike the other versions, this one does not have the "score" or "point" count of the pieces. If FAST mode it took about 10 seconds for the computer to decide it's move. It switches back to slow mode for your input. The program did not know that I needed to pass, so I typed in A0 (an invalid move) to do a pass. At the end it did a count of pieces and it reckognized that I had won. Since this version was written in BASIC it is possible to do a listing and see how it works.

## Boriel ZX Basic Compiler

I've known about the Boriel ZX Basic Compiler for a number of years, but I thought I would give it a try myself. It is designed for creating programs on the Spectrum, but it could be adapted to other computers.

To test it out, I decided to use my one dimensional celluar autamata program that I've been writing for different computers in different languages. The Spectrum version was designed to run with zmakebas. zmakebas just converts a test file, in Spectrum Basic, to a .tap file that the Spectrum will understand. The Basic is tthe same as the Spectrum.

I was thinking that the Boriel compiler would be the same, but it is not. It will not take an existing Spectrum Basic program and compile it. The compiler as its own Basic syntax, similar to Spectrum Basic, but different enough. This may work well when writing from scratch for the compiler, but not so much when trying to compile an existing program.

I read the documentation on the syntax that the Boriel compiler uses and started modifying my program to fit that syntax. I found that the standard Spectrum IF statement will not work. Boriel's IF statement must have an END IF. I had to go through my program and change the IF statements. I kept the same flow control, but with an IF statement that supports multiple statements, I could have changed my program to take advantage of that structure.

All variables must be declared as in C and Pascal. Not doing so can really goof up a program. I had one undeclared variable that I was using in a FOR NEXT loop. The loop kept going beyond the ending number. I was going from 7 to 0 STEP -1. The loop kept going below 9 to a much higher number. I noticed that I had not declared the variable, declared it and then the loop just worked.

The INPUT command is far different that the one for Sinclair Basic. Instead of it being like this:

```
INPUT a$
```

The syntax is this:

```
a$ = INPUT(20)
```

This allows for the user to type in a string of up to 20 characters. I had to dig in the documentation for that one. The full syntax of the language is on a number of pages on the Boriel Wiki. The input is also on the main part of the screen, and not down at line 23, as in Spectrum Basic.

I also found out that if using the variable a$, the compiler complained if you also used the variable a as an integer variable.

After I was able to adapt my program to the Boriel syntax, it compiled fine and it created a .tap file. I loaded it into EightyOne and the program ran very fast.

One thing that I was wondering, was if the compiler could create a program that would run on the T/S 2068. I created a simple program:

```
print "x"
```

then compiled it and tried it on the T/S 2068 in EightyOne. It ran fine. The issue will be when using a command that will need a Spectrum ROM call to execute. The compiler has a library of Z80 assembly instructions that it uses. It should be possible to find all of the Spectrum ROM calls and replace them with T/S 2068 ROM calls. I think this is something that I might look into.

If one is writing just for the Boriel zx compiler, the

syntax does go beyond what ZX Basic does. The Boriel compiler is based on FreeBasic, so that is one source for idea how the syntax of the language works. One of the best features is that Boriel allows for IF statements that support multiple lines and also supports ELSE. In ZX Basic, the IF statement is like this:

```
IF x = 0 THEN print x: print y: print z
```

Now this can look a little clustered. I've never liked using multiple commands in a single line. Here is the same in Boriel:

```
IF x = 0 THEN
  print x
  print y
  print z
END IF
```

The language also supports ELSE and ELSE IF in the same way that SuperBasic does.

Another flow control command is WHILE ... WEND. The WHILE is used like this:

```
WHILE x = 1
  .......
  .......
WEND
```

Another flow control set is DO .... UNTIl. The WHILE does a check at the start of the loop, DO ... UNTIL does the check at the end. It works like this:

```
DO
  .......
  .......
UNTIL x = 1
```

Each has their own use case. In classical flow control documentation, a DO .. UNTIL is the same as a DO .. WHILE. The WHILE .. WEND is also a WHILE .. DO loop.

Boriel compiler does have a reserved word list on it's Wiki. This is a great resource for discovering the syntax of the language and what commands are different in Boriel versus ZX Basic.

For those programmers not wanting to get into C (with Z88dk) or assembly, the Boriel compiler is a great way to get speed and still work in Basic.