Assembly Language on the ZX81

An Updated Getting Started Guide

```
                LD (keyboard_var),A

_loop           CALL GkeyP                    ; A = key value - need to hold on to this un
                SUB $1D                       ; Adjust '1'...'Z' to be in the range 0 ... 3
                JR C,_loop                    ; Check in range
                CP 35
                JR NC,_loop                   ; Check in range
                LD E,A
                INC A                         ; Adjust to range 1..35 for '1'..'Z'
                LD D,0
                POP HL                                  key table address
                PUSH HL
                ADD HL,DE
                ADD HL,DE                                                y value * 2
                LD E,(HL)                                               jump to the location
                INC HL                                                  HL, it jumps to the m
                LD H,(HL
                LD L,E
                CALL                                                  on to A to here from GKP
                LD A
                OR                                                 dicates return
                JR
                X                                               flag
                LD (ke
                POP HL                                         ack

keyboard_action:
                RET                                  his RET instruction for null routin

_callHL         JP (HL)                       ; Allow RET from the called routine to return
                                              ; caller of _callHL

keyboard_var: DEFB 0

;----------------------------------------------------------
```

By

Timothy Swenson

creative commons
COMMONS DEED

Attribution-NonCommercial-ShareAlike 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works

Under the following conditions:

- **Attribution**	You must give the original author credit.
- **Noncommercial**	You may not use this work for commercial purposes.
- **Share Alike**	If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

For any reuse or distribution, you must make clear to others the license terms of this work. Any of these conditions can be waived if you get permission from the copyright holder. Your fair use and other rights are in no way affected by the above.

This paper will discuss the process of programming in Z80 assembly language on the ZX81, using a MS Windows based system. The use of a ZX81 emulator, a cross compiler, and editing tools make this process easier, faster, and smoother than doing it natively on the ZX81. This paper is not designed to teach the syntax of assembly language. Since we are 30 years removed from the release of the ZX81 and the publication of assembly language books for the ZX81, this paper covers the changes that have taken place in that time.

Introduction

Back in November, 1981, I purchased my first computer, a Sinclair ZX81. I saw an ad in Popular Science magazine advertizing the ZX81 for $150 and 16K memory pack for $100. The computer arrived via mail the day before Thanksgiving. After some learning curves, I was writing programs in BASIC. About 6 months, I decided to learn assembly language programming (also known as machine code). I purchased "Machine Language Programming Made Simple for your Sinclair ZX80/ZX81" and was totally lost. The next book, "Mastering Machine Code on your ZX81" by Toni Baker, was much better and provided a lot more runnable examples. I bought an assembler for the ZX81, but I could not get it to work, so I learned to hand assemble the opcodes to hex, and poke those into REM statements. I did learn to write a number of short programs.

Eventually I moved onto the TS-2068, and the QL, where I stayed for a number of years. I tried to learn 68000 assembly for the QL, but I could not get the hang of it. It has been more than 3 years since I've touched a Sinclair computer, including emulators. I've spent my time doing other hobbies, from photography to local history. I felt like I needed to get back to some programming. I did not have the time for anything long and involved, so I thought that ZX81 assembly would be a good fit.

Having tinkered with a number of emulators for different systems, I knew that I did not want to break out a real ZX81 and set it up, so I planned to use a ZX81 emulator. Having forgotten how to type on the ZX81 keyboard, I did not want to use only the emulator, but I needed some cross assembler that I could run under Windows XP and load the binary into the ZX81 emulator in binary format. Luckily, I found all that I needed with a few good Google searches.

My goal was to write programs in 100% assembly. I did not want to write some routines that could be called from BASIC. I already know how to program in BASIC. I did not want to use any tools that required loading the system above RAMTOP and then loading my own program. I wanted tools or libraries, in source, that could be easily added to my assembly program, and compiled in one go.

I'm not an expert in assembly programming, but when I was looking for information to get me started, did not find a good tutorial for what I needed. As I discovered what works and what does not work, I documented it for my own edification and decided to make that documentation available so that others will not have to re-invent the wheel. I've found that a good way to learn a subject is to document it.

Not all of the code contained herein is my own code. Some I found on the Web, some I found in books, and some I did figure out myself.

Part 1 - Things that you will need

The Emulator

Eightyone is a Windows based ZX81 emulator, written by Michael D. Wynne, that will also emulate a number of other Sinclair and Sinclair-like systems, including the ZX80, Jupiter Ace, Spectrum, Timex Sinclair 2068, and a couple ZX81 clones. It also supports a number of different HI-RES modes on the ZX81. The program is Open Source and the source code is down-loadable if you want it. There is not much of a manual, but the menus are fairly self explanatory. With some tinkering around, I was able to get it to work for me. One thing I noticed with EightyOne, one difference between the settings on the ZX81 and T/S 1000 modes is that the T/S 1000 has a setting called NTSC set to ON. With this set to ON, .p files that are supposed to autorun will not autorun. I just use the ZX81 machine setting and it works fine. With the emulator running on a PC, I really don't if the NTSC setting is important. It might change FRAMES from 50/sec to 60/sec.

EightyOne is available from: www.chuntey.com or www.aptanet.org/eightyone

The Cross Compiler

The Telemark Cross Assembler (TASM) is a MS-DOS-based cross assembler that supports a number of CPU's including the Z80. It is written by Thomas N. Anderson, with its last update in 2002. It is a shareware application, so if you are going to use it a lot, then you are supposed to register it.

Through a ZX81 message board (http://www.rwapservices.co.uk/ZX80_ZX81/forums/)**,** I found a download (temper.zip) that included TASM and a number of TASM templates that are used to have TASM create a .p file. The .p file is a "tape" file supported by a number of ZX81 emulators. After compilation (or cross assembling), the .p file can be loaded into an emulator and run with the LOAD "" command. TASM can be found at:

```
http://home.comcast.net/~tasm/
```

The Books

Years ago I had purchased two books on Assembly (or machine code). One left me horribly confused and lost. The second one made sense and I was able to write a few programs. The second book was "Mastering Machine Code on Your ZX81" by Toni Baker. I still have that book, but I wanted a digital copy to reading on my laptop. Luckily the website, World of Spectrum, has this book, and many others, available. The link to the books is:

```
ftp://ftp.worldofspectrum.org/pub/sinclair/books/
```

Here is a list of others books that you will find useful:

The Complete Timex TS1000 - Sinclair ZX81 ROM Disassembly

This has the full ZX81 ROM disassembly that details all of the ROM routines.

ZX81 Basic Programming

This is the standard book that came with the ZX81 computer. You don't need it for BASIC, but it does cover the layout of memory, the screen, the characters, system variables, and so on. A good reference to have.

Z80 Assembly Language Programming

This book lists all of the possible opcodes for the Z80. Some opcodes that you think should exist, but they may not, so this book will detail the real opcodes, allowing you to workaround any limitations.

I would recommend downloading at least these books. I've found myself using all four of them. The Toni Baker book (even with some errors) is very useful. An updated version is available in HTML format at:

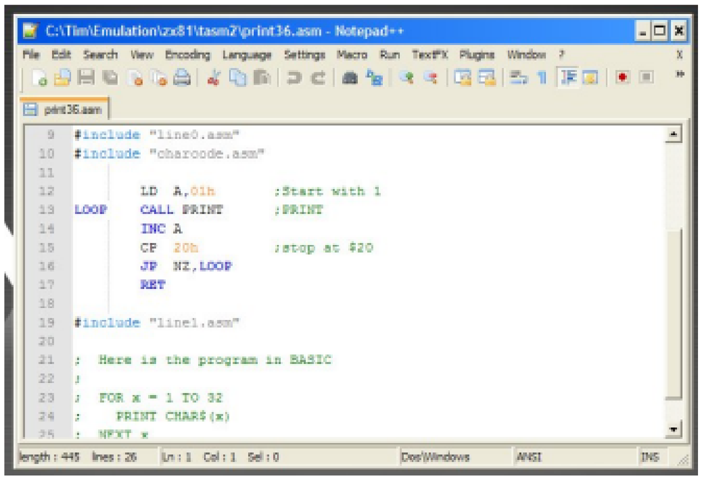
`http://www.users.waitrose.com/~thunor/mmcoyzx81/index.html`

There are a number of other ZX81 books at the above FTP site. Most of them deal with ZX81 Basic but you might find them interesting (and the price is right).

The Editor

When editing source code, built-in Notepad is decent, but it lacks a number of bells and whistles. One thing that I wanted was a way to convert characters to upper case. I like my assembly programs to have the opcodes in upper case, but I find it a pain to type it that way. I prefer to type in lower case and go back later to change it.

Looking around for Open Source text editors, I found Notepad++. Notepad++ is an updated version of Notepad (mostly in name and function) that has a number of features for programming. It does have the upper case conversion command (CTRL-SHIFT-U). It also has text highlighting for Assembly. It remembers indentation and automatically indents for you. It has tabs so that you can edit more than one file at a time. It can be found at: `notepad-plus-plus.org`


C:\Tim\Emulation\zx81\tasm2\print36.asm - Notepad++
File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?
print36.asm
length : 445 lines : 26 Ln : 1 Col : 1 Sel : 0 Dos\Windows ANSI INS

```
#include "line0.asm"
#include "charcode.asm"

        LD   A,01h          ;Start with 1
LOOP    CALL PRINT          ;PRINT
        INC  A
        CP   20h            ;stop at $20
        JP   NZ,LOOP
        RET

#include "line1.asm"

;  Here is the program in BASIC
;
;  FOR x = 1 TO 32
;    PRINT CHAR$(x)
;  NEXT x
```

Part Two - The Basics

Getting Started

To start working in machine code, you will need to get the above tools and have them installed on your computer. I've created a directory called 'zx81' and installed both EightyOne and TASM in their own subdirectories. I've created a subdirectory called 'docs' and stored the PDF documents mentioned above. Any programs that I write are stored in the TASM directory so that they can be found by TASM.

TASM and the ZX81

TASM is a cross compiler for a number of CPU's. It supports the Z80 but it is not specifically designed to support the ZX81. A number of header and footer files have been created to get TASM to work with the ZX81. The templates are:

```
line0.asm
line1.asm
main.asm
sysvars.asm
charcode.asm
```

`line0.asm` sets up the line of BASIC with the REM statement to hold the machine code program. `line1.asm` sets up the RAND USR call to start the machine code program. `main.asm` is the main part of the program and it is where you will insert your code. `sysvars.asm` had a number of defines for variables and ROM routines specific to the ZX81. `charcode.asm` is a listing of the ZX81 character codes that have been defined as a set of constants. Instead of having to type $31 for the L character, it is referenced as _L.

A Short Assembly Program

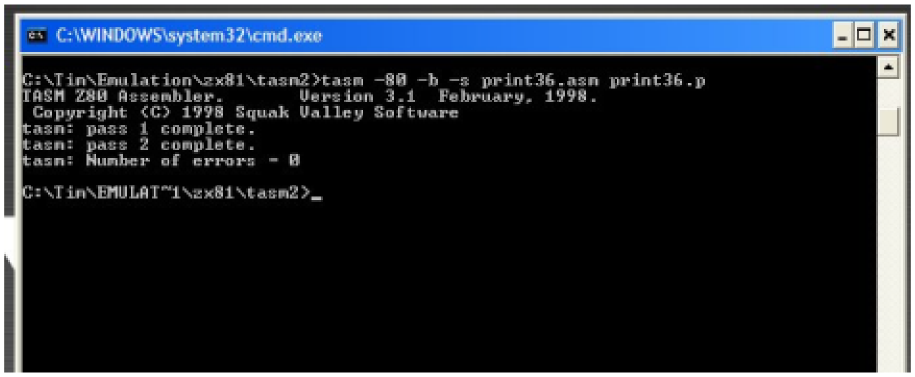
Here is a short example assembly program. The program will print out 36 characters of the ZX81 character set, starting at character 1 (character 0 is SPACE and will not show up).

```
     LD   A,1        ; start with 1
loop                 ; label used for jump
     RST  $10        ; PRINT
     INC  A          ; increment A
     CP   36         ; stop at 36
     JP   NZ,loop    ; if not 36, then goto/jump to LOOP
     RET             ; Return to BASIC
```

Here is the program in BASIC:

```
10 LET A = 1
20 PRINT CHR$(A)
```

```
30 LET A = A + 1
40 IF A <> 36 THEN GOTO 20
50 STOP
```



```
C:\Tim\Emulation\zx81\tasm2>tasm -80 -b -s print36.asm print36.p
TASM Z80 Assembler.       Version 3.1  February, 1998.
 Copyright (C) 1998 Squak Valley Software
tasm: pass 1 complete.
tasm: pass 2 complete.
tasm: Number of errors = 0

C:\Tim\EMULAT~1\zx81\tasm2>_
```

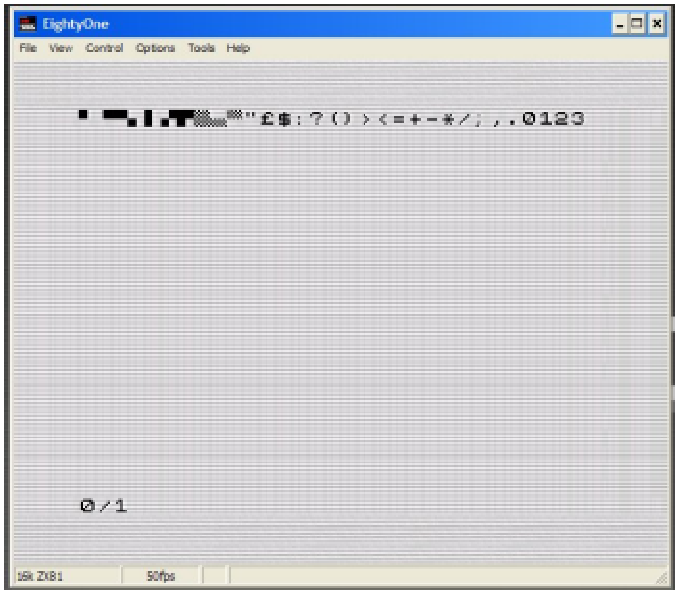
In the `main.asm` file, it shows you were the put your code. Type to above machine code program into the file, editing out the example code already in the file, and save the file. The next step is to execute TASM and have it compile the program. TASM is command line driven, so you will need to open a DOS command line to run it (Start -> Run -> cmd). The command to run TASM is:

```
tasm -80 -b -s main.asm test.p
```

The command line options are:

-80	- Convert to Z80
-b	- Produce binary file
-s	- Write a symbol table file
main.asm	- The source code to compile
test.p	- The resultant binary file.

TASM will take the `main.asm` file, cross assemble it, and create the file `test.p`. You can copy the file `main.asm` to something else, so that you don't confuse which `main.asm` is for which project. Since TASM was written for MS-DOS, it still only understands the 8.3 DOS name scheme, so keep your file names to 8 characters or less.



```
 ▪ ▬▖▌▖▛▒▒"£$:?()><=+-*/;,.0123

0/1
```

For those with Intel systems but not using MS Windows, TASM should run fine under DOS-Box, the MS-DOS emulator for a number of x86 operating systems. The resultant .p file should run with most of the emulators for other operating systems.

Now, we need to test the compiled code in the ZX81 emulator. Start Eightyone by double clicking on `eightyone.exe`. The default setting is for the ZX81, so you don't have to make any changes to Eightyone. On the toolbar, click on File -> Open Tape. Use the menu to navigate to where `test.p` file is located and select `test.p`. The ZX81 will look like it has reset. It will then autotype the LOAD command, load the program, and RUN it. The next thing you should see is 36 characters on the screen.

PRINT and PRINT AT

In the above example the command `rst $10` is used to call the PRINT routine in the ZX81 ROM, but it is also possible to use the CALL opcode and make a call to the PRINT routine. In this example, PRINT has been defined in the header file as $10, which is the location in the ROM for PRINT. The routine takes the value in the A register and prints it to the next location on the screen:

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

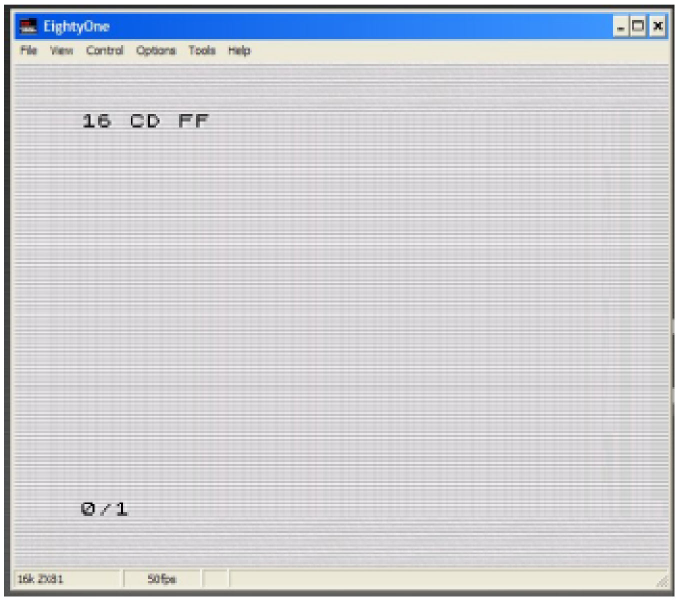
To use the PRINT AT feature of BASIC, a call must be made to the PRINT AT ROM routine before calling PRINT. This routine moves the next print location specified by the B and C Registers. B is Row and C is Column. In this example, 16 is being loaded into both registers, then PRINTAT is being called. This sets the next print position to row 16, column 16. Then the code above can be used to print a character.

```
LD   B,16
LD   C,16
CALL PRINTAT        ; set cursor to position 16,16
```

The full program can be found in **`printat.asm.`**

Printing Hex Values

So far, we've only looked at printing single characters on the screen. What would it take to print a number to the screen? For hexadecimal, this is fairly simple to do. A hexadecimal number will always have two digits. If the value is lower than 16 decimal, then it will have a leading zero. So, looking at a hexadecimal number, $FF, or $06, each digit takes up half of the 8-bit byte, or a single nibble (4 bits). It is necessary to isolate each nibble and determine what number it is. Once it is isolated, the value 28 (or $1C) is added to the number to reflect the right ZX81 character. The ZX81 character 28 is the same as zero.



```
16 CD FF



0/1
```

The left digit needs to be printed out first, so to isolate it, the carry bit is zero'd out and the number is right shifted to move all of the bits over. Having the carry bit zero'd out means that the bits coming into the register will be zero.

To get the right digit, a mask is used to get rid of the left 4 bits. The mask is ANDed with the register, turning the 4 left bits to zero.

Here is the program to test printing hexadecimal numbers with the main point being implemented as a subroutine.

```
     LD   A,$16
     CALL hprint
```

```
          LD   A,$CD
          CALL hprint
          LD   A,$FF
          CALL hprint
          RET

hprint    PUSH AF        ;store the original value of A for later
          AND  $F0       ; isolate the first digit
          RRA
          RRA
          RRA
          RRA
          ADD  A,$1C     ; add 28 to the character code
          CALL PRINT     ;
          POP  AF        ; retrieve original value of A
          AND  $0F       ; isolate the second digit
          ADD  A,$1C     ; add 28 to the character code
          CALL PRINT
          LD   A,$00
          CALL PRINT     ; print a space character
          RET
```

Constants

The use of constants is a way to use a number through out a program, but to define it in one location, so that if it has to change, it only needs to be changed once. Putting constants through out a program as literals, is called "hard coding" and a bad idea.

With TASM, there is a way to define a constant. Let's say that you want to use a graphics character in a program. You first want to use the grey square, which has a character code of 8 or $08. This character constant can be defined in this way:

```
     SQUARE    .equ $08
```

And when you want to print that character, it would be done like this:

```
     LD   A,SQUARE
     CALL PRINT
```

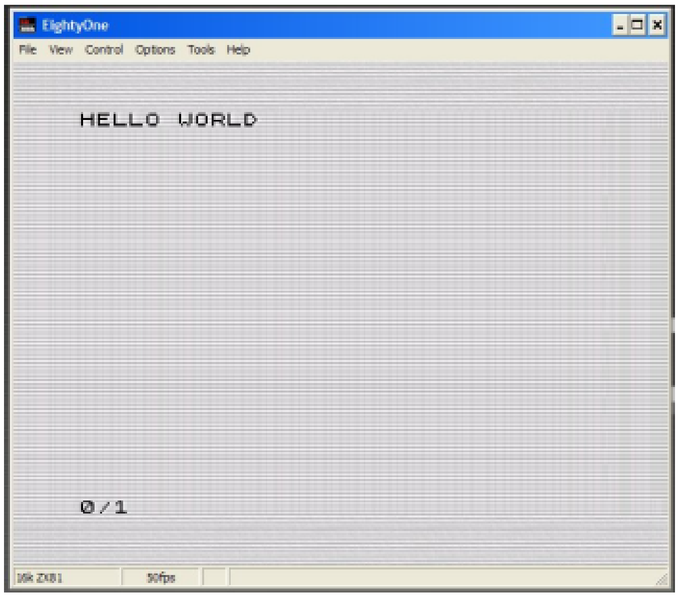
Later, if you decide to change from the grey square to a black square, you only need to change the constant and your whole program will be updated and use the black square ($80).

Hello World

One of the first programs written for any language is one that print out the phrase "Hello World". In BASIC, this is a single line program:

10 PRINT "HELLO WORLD"

For assembly, it is a little more complicated. First, the string "HELLO WORLD" has to be stored somewhere. With TASM, we can use the `.byte` directive to define a number of bytes. The label "`line:`" is used to define where the string starts. Normally, the bytes are listed in decimal or hexidecimal, but with the defines created in `sysvar.asm`, the `_H` sequence has been assigned to the value of the letter H. This is also done with the other letters. To know where the end of the string is, the byte $FF has been put at the end.



```
HELLO WORLD

0/1
```

The program loads the location of "`line`" into HL. Then the character at HL is stored in A. A comparison is made to see if the end of the string has been reached. If not, then the character is printed, HL is incremented to move to the next character and the program loops back.

`hello.asm:`
```
          LD   HL,line        ;load HL with address of line
pline     LD   A,(HL)         ;load A with a character at HL
          CP   $ff            ;The end of the string?
          JP   Z,end          ;if so, then jump to end
          CALL PRINT          ;print character
          INC  HL             ;increment HL to get to next character
          JP   pline          ;jump to beginning of loop
end       RET                 ;exit

line:          .byte  _H,_E,_L,_L,_O,$00,_W,_O,_R,_L,_D,$ff
```

Plotting

Printing is the most common way to get output to the screen. Another way is to plot points on the screen. The ROM routine PLOT is used to plot a point on the screen. To plot a point at X,Y, Y has to be loaded in to the B Register and X into the C register. The T-ADDR system variable is used by the PLOT routine, so you must preserve it (by PUSHing it) before you call PLOT and restore it afterwords. This is a fair bit of code just to plot a point, so it would be best to put most of the code into a subroutine and reuse it. See the next section for a discussion on subroutines.

```
          LD   B,10           ; Store Y in B
          LD   C,10           ; Store X in C
          LD   HL,($4030)     ; Get T-ADDR
          PUSH HL             ;SAVE T-ADDR
          LD   A,$98          ;Less than $9E to plot
          LD   ($4030),A
          CALL PLOT
          POP   HL
          LD    ($4030),HL     ;RESTORE T-ADDR
          RET
```

Part Three - More Advanced Programming

Reading the Keyboard

So far, we've discussed output, now it is time to talk about getting some input. There are two ROM routines used to get input from the keyboard. The first is KEYBOARD or KSCAN. This routine scans the keyboard for a pressed key and returns the key defined by vertical and horizontal keyboard section. See Toni Bakers' book, page 88-89, for a discussion on the hardware. The value, returned in HL, does not mean anything to us, so we have to use another routine DECODE or FINDCHAR, to convert that hardware value into a character code. The DECODE routine wants the hardware code to be in HL and it returns the key code in A.

This example program scans the keyboard, translates the results to a character code, and then prints out the character code to the screen. The keyboard scanning routine is so fast, that a delay is put into the program, else too many characters will appear for each keystroke.

type.asm:

```
wait      CALL      KSCAN     ; get a key from the keyboard
          LD        B,H
          LD        C,L
          LD        D,C
          INC       D
          LD        A,1       ; If no key entered
          JR        Z,wait    ; then loop
          CALL      FINDCHAR  ;Translate keyboard result to character
          LD        A,(HL)    ; Put results into reg a
          CP        $18       ; Is character a / (slash)
          JR        Z,end     ; If so, jump to END
          CALL      PRINT     ; Print character
          LD        BC,$1200  ; Set pause to $1200
delay     DEC       BC        ; Pause routine
          LD        A,B
          OR        C
          JR        NZ,delay

          JP   wait
end
          RET
```

PAUSE, SCROLL, and CLS

There are a couple of more ROM routines that are handy and fairly easy to use. The PAUSE routine works the same way as the BASIC command. The length of the PAUSE (in number of FRAMES) is stored in the FRAMES system variable and a call is made to the PAUSE ROM routine. Like the PAUSE in BASIC, hitting any key will stop the PAUSE and continue with the program.

```
          LD   HL,$FFFF       ; how long to pause
          LD   (FRAMES),HL    ; store it in FRAMES
          CALL PAUSE          ;call the PAUSE routine
```

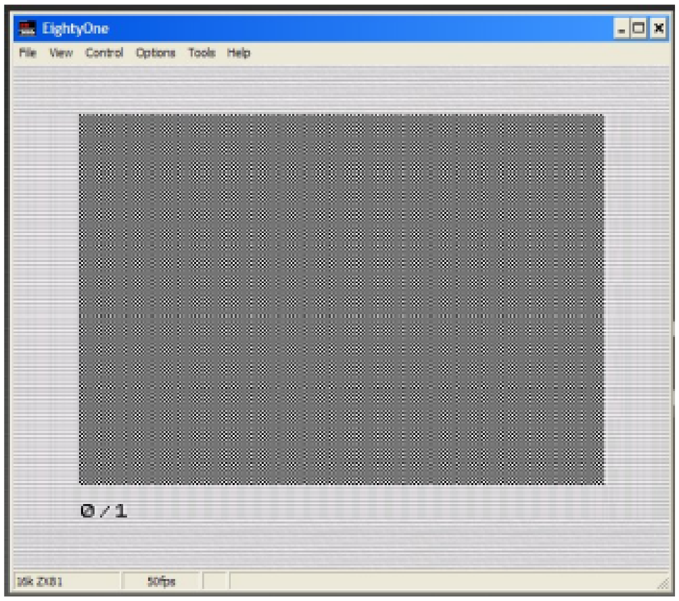
SCROLL is also the same as in BASIC. The routine will scroll up the screen one row of characters. The routine does not need any arguments and is done with a simple CALL:

```
          CALL SCROLL
```

Another ROM routine that is the same in BASIC is CLS, or clear screen. It fulls the screen with space characters. With an expanded system, it also makes sure that the screen is fully expanded and 793 bytes long.

```
          CALL CLS
```

Writing Directly to the Screen



The built in PRINT ROM calls can be a little on the slow side. For faster printing, it is possible to send bytes directly to screen memory. The data can be a simple string or it can be a full screenshot. Before writing directly to the screen, you need to perform a CLS. This expands the screen to it's full size. I've tried the program without the CLS and it will not work.

In this example, the grey square character is printed to the entire screen. HL is loaded with the pointer to the location of the screen. The program then goes through a loop 22 times, copying the grey square character code to the screen.

```
          CALL CLS
          ld   hl,(D_FILE)    ; Get start of display
          ld   c,22           ; line counter (22 lines)
LOOP1     inc  hl             ; get past EOL
          ld   b,32           ; character counter (32 rows)
LOOP2     ld   (HL),$08       ; print grey square character
          inc  hl             ; move to next print position
          djnz LOOP2          ; Do it again until B=0
          dec  c              ; next line
          JR   nz,LOOP1
DONE      RET                  ; exit
```

The next program writes a string directly to the screen. It functions the same as the one above except that the data written to the screen is read from a data section. It is possible to modify this program to read in a whole screen from a data section and output it to the screen. Make sure to remember that at the end of every line on the screen is an EOL marker ($76). Either have it as part of the data section, or make sure to jump over the existing EOL marker and move to the next line.

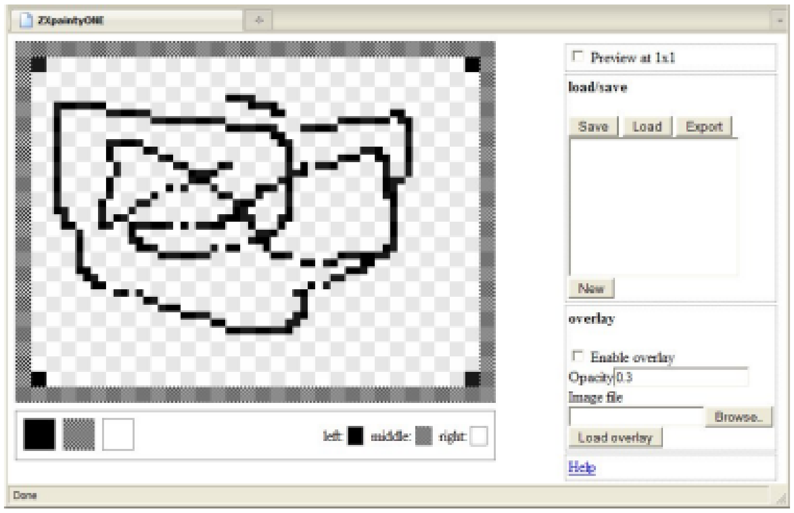
```
          CALL CLS
          LD   HL,(D_FILE)     ; Get start of display
          LD   DE,screen       ; Get start of string
          inc  hl              ; get past first EOL
loop      LD   A,(DE)
          CP   $FF
          JP   Z,done
          LD   (HL),A          ; print character stored in A
          INC  HL              ; move to next print position
          INC  DE              ; move to next character in string
          JP   loop
done RET                       ; exit

screen:

.byte  _H,_E,_L,_L,_O,$00,_W,_O,_R,_L,_D,$FF
```

See screen3.asm for an example of loading an entire screen from data statements to the screen. It is possible to store a number of screens in .byte arrays and load them when needed. Each screen is 793 byes long. It functions the same as the program above except that the data section hold a whole screen worth of data and not just a string.

ZXpaintyOne



While on the subject of loading screens, there is a neat JavaScript program that let's you create ZX81 screens with graphics characters. It loads into a web browser and provides mouse-driven way to draw screens. It only supports the graphics characters and does not allow for adding t ext. Once the screen is drawn, "saving" it creates a long binary description of the characters of the screen. To make the program more compatible with assembly programs, there are two changes that need t o be made. The JavaScript is a text .

HTML file, so it can easily be edited in Notepad++ (which does show line numbers).

1. Insert this as line 265:
   if (x := 62) savedata += ',$'

2. Insert this as line 257:
   savedata += '\n$'

This will put a $ in front of each character to define it as hexadecimal and it will insert commas between the numbers. ZXpaintyOne can be found at the following location:

```
matt.west.co.tt/demoscene/zxpaintyone/
```

Using Variables with TASM

The Z80 registers are limited and they can change when calling ROM routines or when using certain opcodes. There needs to be a way to keep data around so that it can be used in all different parts of the program. The solution is to define parts of memory as locations and used like variables.

Variables are defined with the .byte directive.:

```
x_pos     .byte     16
y_pos     .byte     16
```

This defines two variables and sets both of them to the default value of 16.

To load the variable into the A register, indirect addressing is used. To put x_pos into the B register and y_pos into the A register, do the following:

```
LD   A,(x_pos)
LD   B,A
LD   A,(y_pos)
```

Saving data back into the variable is basically the same indirect addressing:

```
LD   (x_pos),A
LD   (y_pos),A
```

Subroutines

Subroutines are parts of the program that will be called multiple times from different locations. Subroutines are also another way to break up the program into smaller chunks. Given the structure of Assembly language, subroutines function very similar to the GOSUB in BASIC. Subroutines are normally located at the end of main program, similar to BASIC.

This example program moves a character around the screen based on keyboard input from the user. The code for handling each direction (up, down, left, and right) are implemented as subroutines. This program also shows how variables are used.

movechar.asm:

```
x_pos     .byte     16
y_pos     .byte     16

START     LD        A,(X_POS)       ;Get starting location of Char
          LD        B,A
          LD        A,(Y_POS)
          LD        C,A
          CALL      PRINTAT         ; set cusor to position 16,16
          LD        A,$08           ; grey square character
          CALL      PRINT           ; print character
WAIT      CALL      KEYBOARD        ; ROM call to read keyboard
          LD        B,H             ; Get output from HL
          LD        C,L             ; and put into B and C
          LD        D,C
          INC       D
          LD        a,01h
          JR        Z, WAIT
          CALL      DECODE          ; ROM routine to get character
          LD        A,(HL)          ; Get results of DECODE
          CP        $21             ; 5 or LEFT key
          JP        Z, LEFT
          CP        $22             ; 6 or DOWN key
          JP        Z, DOWN
          CP        $23             ; 7 or UP key
          JP        Z, UP
          CP        $24             ; 8 or RIGHT key
          JP        Z, RIGHT
          CP        $26             ; 0 or zero key (use to Quit)
          JP        Z, DONE
          JP        WAIT

LEFT      LD        A,(x_pos)       ; print space at x_pos,y_pos
          LD        B,A
          LD        A,(y_pos)
          LD        C,A
          CALL      PRINTAT
          LD        A,00h
          CALL      PRINT
          LD        A,(x_pos)       ; get x_pos
          DEC       A               ; decrement it
          LD        (x_pos),A       ; store x_pos
          JP        PAUSE
```

```
RIGHT     LD        A,(x_pos)       ; print space at x_pos,y_pos
          LD        B,A
          LD        A,(y_pos)
          LD        C,A
          CALL      PRINTAT
          LD        A,00h
          CALL PRINT
          LD        A,(x_pos)       ; get x_pos
          INC       A               ; increment it
          LD        (x_pos),A       ; store x_pos
          JP        PAUSE1


UP        LD        A,(x_pos)       ; print space at x_pos,y_pos
          LD        B,A
          LD        A,(y_pos)
          LD        C,A
          CALL      PRINTAT
          LD        A,00h
          CALL PRINT
          LD        A,(y_pos)       ; get y_pos
          DEC       A               ; increment it
          LD        (y_pos),A       ; store y_pos
          JP        PAUSE1

DOWN      LD        A,(x_pos)       ; print space at x_pos,y_pos
          LD        B,A
          LD        A,(y_pos)
          LD        C,A
          CALL      PRINTAT
          LD        A,00h
          CALL      PRINT
          LD        A,(y_pos)       ; get y_pos
          INC       A               ; increment it
          LD        (y_pos),A       ; store y_pos

PAUSE1    LD        BC,$1200        ; Set pause to $1200
DELAY     DEC       BC              ; Pause routine
          LD        A,B
          OR        C
          JR        NZ,DELAY        ; loop until 0
          JP        START
DONE
          RET                       ; Return to BASIC
```

Multiplication and Division

Addition and subtraction are built into the Z80, but the programmer has to implement multiplication and subtraction. Both of the approaches below for multiplication and division use the brute force approach and should be fine for smaller numbers.

The simplest form of multiplication is shifting the register to the left. This is the same as multiplying by 2. For division, shifting to the right is the same as dividing by two. If you are multiplying or dividing by two, then use this method.

Multiplication

The brute force approach to multiplication is iterative addition. Meaning that a number is added to itself, X number of times. To get 3 X 5, then the number 3 is added to itself, 5 times (or vice versa). The code below does just that. One operand is loaded into B and the other into C. It is the operand in C that is added to itself, so it should be the larger operand. This will save the number of cycles it takes to calculate the answer. Since any number multiplied by 0 is 0, the routine below first sets the result in HL to 0, then checks to see if either B or C is 0. If so, then it returns with HL set to 0.

```
     LD   B,X
     LD   C,X

multiply:
     LD   HL,0        ; zero out HL
     LD   A,B
     CP   0           ; Is B zero?
     RET  Z           ; If so, return and HL=0
     LD   A,C
     CP   0           ; Is C zero?
     RET  Z           ; If so, return and HL=0
     LD   D,0         ; Zero out D
     LD   E,C         ; load C in E
loop:
     ADD  HL,DE
     DJNZ loop
     RET              ; result is in HL
```

Division

The brute force approach to division is iterative subtraction until subtraction is no longer possible. In the code below, the operation is this: B=HL/E. E is subtracted from HL until E is less than HL. Any remainder is ignored, so the result is the same as INT(HL/E).

```
divide:
     LD   A,E
     CP   0
     RET  Z
     LD   B,0
     LD   D,0
loop:
     SBC  HL,DE
     INC  B
     JR   NC,loop
     DEC  B
```

The end result is in the B register.

Arrays

Arrays are a very common way to store data in a program. The two most common types of arrays are single and two dimensional. A single dimension array is comprised of a single row with lots of elements, with each element being referenced by a number. With an array name of bob, the 3rd element of the array is referenced as bob(3). A two dimensional array is an array composed of both rows and columns. Most of the time, a two dimensional array is also called a table. The reference a single element of a two dimensional array, two numbers are needed. For the array named bob, the element at column 3 row 4, is referenced as bob(3,4).

In reality, all arrays are single dimensional, since computer memory is a long row of memory locations accessed by the address. But, with a little math, it is possible to let the one dimensional memory look like a two dimensional array.

Single Dimension

In a single dimensional array, referencing the Nth element of the array is very easy, but not as trivial as one might think. The array is first defined in the program with a label and the .byte directive:

```
     array:    .byte 0,0,0,0,0,0,0,0
```

This defines array as a single array with 8 elements. The first element is at the memory location of array, and the second one at array+1. So, Nth element of the array is not array+N, but array+N-1. In the example program, the element that is needed is the 4th (the letter D) and it will be printed to the screen.

```
     LD   BC,4
     DEC  BC
     LD   HL,(array)
     ADD  HL,BC
     LD   A,(HL)
     CALL PRINT
     RET

array     .byte     _A,_B,_C,_D,_E,_F,_G
```

Two Dimension

A little math is needed to simulate a two dimensional array in a single dimensional memory system. To determine the location in the array (translating from a 2D array to a 1D array), the following formula is used:

$$(X-1)*array\_size+y-1$$

The formula is designed for starting an array index at 1 and not zero. In the example below, the element that we are seeking is array[3,3]. The array is 4x4 or 4 rows of 4 elements. In the program below, the translation of 2D to 1D is accomplished and the letter K will be printed to the screen.

```
     LD   B,3         ; X into B
     DEC  B           ; (x-1)
     LD   C,col
multiply:             ; (x-1)*col
     LD   HL,0        ; zero out HL
     LD   A,B
     CP   0           ; Is B zero?
     RET  Z           ; If so, return and HL=0
     LD   A,C
     CP   0           ; Is C zero?
     RET  Z           ; If so, return and HL=0
     LD   D,0         ; Zero out D
     LD   E,C         ; load C in E
loop:
     ADD  HL,DE       ; add DE to HL, B times
     DJNZ loop
                      ; X * col is now in HL
     LD   DE,3        ; Y into DE
     DEC  DE          ; (Y-1)
     ADD  HL,DE       ; add result of multiplication and (Y-1)
     LD   DE,array    ; Get start of array
     ADD  HL,DE       ; add offset to array
     LD   A,(HL)
     CALL PRINT
     RET

array     .byte       _A,_B,_C,_D
          .byte       _E,_F,_G,_H
          .byte       _I,_J,_K,_L
          .byte       _M,_N,_O,_P
```

Memory Management

In other computer languages there are ways to ask the operating system for checks of memory. Most languages have a command like malloc in C, and RESPR in QL SuperBasic. With Assembly, there are no operating system call, and on the ZX81 there are no ROM routines for allocating memory.

So, if you need a chunk of memory, the you have to determine where you will get the memory from. With the ZX81, the system has specific locations in memory already allocated. Starting at 16K, the first bit if memory is the system variables, then the program area, then the screen (display file), then BASIC variables, then the calculator stack. From the calculator stack, until and the GOSUB stack (ERR_SP), is a large chuck of free memory. In doing some looking at the system, I found that most of the time that ERR_SP was set at 32764, which is just under memory location $8000 in hex.

If you look at memory in "thousands" in Hex, $4000 is 16384, which is the start of RAM for a 16K system. The next "thousands" are:

$5000	20480
$6000	24576
$7000	28672
$8000	32768

Notice that the difference in these "thousand" memory locations is 4K. Depending on how much memory you need and how large your program is, is it very likely that memory located at either $6000 or $7000 is going to be free and available.

For smaller chunks of memory, TASM supports the .block directive. This functions simaraly to the .byte directive, except that it is for more than a byte and values to not have to be assigned to the block of memory.

Usage is like this: mem_chunk   .block 30

This assigns a block of 30 bytes referenced by the name "mem_chunk". These bytes are allocated in the Program Area of memory, just like the assembly program itself.

TASM Notation

Different assemblers have different notation for directives, such as defining a byte or word, defining constants, etc. TASM uses a rather unique dotted notation with a dot before each notation. This might have roots in the Unix utility roff, because with TASM, some of the dot notations are also commands for printing.

Here is an example of a notation commonly seen with other assemblers:

```
VERSN:    DEFB 0
E_PPC:    DEFW 2

KEYBOARD  EQU $02BB
DECODE    EQU $07BD
```

If you have an assembly program in this notation, instead of having to edit the program, setting up some defines in TASM will allow this program to work with TASM. Here are some example defines that will allow this notation to compile just fine with TASM.

```
#define   DEFB .BYTE
#define   DEFW .WORD
#define   DEFM .TEXT
#define   ORG  .ORG
#define   EQU  .EQU
```

Every time TASM sees `DEFB` it will replace it with `.byte`, which TASM understands.

Assembly Coding Conventions

When writing code in any language, it is good to create coding conventions that you use through your programs. These conventions are some simple formatting rules that will make the programs easier to read and understand. Here are a number of my coding conventions:

CALL Labels
    ROM calls - Upper case
    Local Calls - lower case

By using upper case for ROM calls and lower case for calls to local routines, it will be easier to determine exactly where the call is going.

Variable Labels
    System Variables - Upper case
    Local Variables - lower case

By using uppercase for system variables and lower case for local variables, it will be easier to determine which type of variable is being used.

Appendix I - Rom Calls

This appendix lists a number of useful ROM calls. The names are those used in the ZX81 ROM Disassembly book by Dr. Ian Logan. Alternate names are used, mostly from the book "Mastering Machine Code for the ZX81" by Toni Baker.

CLS **$0A2A**

Clears the screen.

```
     CALL CLS
```

DECODE **$07BD**

Takes the output from KEYBOARD and converts it into a character code. Output from KEYBOARD must be in the BC register pair. A ponter to the character code is returned in the HL register pair. From there it can be loaded into the A register. Alternate name; FINDCHAR

```
     CALL KEYBOARD
     LD   B,H
     LD   C,L
     CALL DECODE
     LD   A,(HL)
```

FAST **$0F23**

Switches the display to FAST mode.

```
     CALL FAST
```

KEYBOARD **$02BB**

Scans the keyboard and returns the horizontal and vertical section for any key that is pressed. Results are returned in HL register pair. Alternate name; KSCAN. See DECODE for example code.

PAUSE **$0F35**

Pauses for a defined period of time. If a key is hit during this time, it will end the pause. The length of the pause is a 16 bit number loaded into the BC registers. This value should not be greater than 32767. Time is in frames; 50 frames a second for European system and 60 frames a second for US systems. For emulators, check their documentation for how many frames they implement.

```
     LD   BC,$0200
     CALL PAUSE
```

PLOT / UNPLOT $0BB2

Plots a point on the screen. The Y value is loaded in the B register and the X value in the C register. The contents of the system variable T_ADDR determines if a PLOT or UNPLOT is done. If T_ADDR is less than $9E, then PLOT is performed. If a is $9E or greater, the UNPLOT is performed. T_ADDR must be saved before it is changed and restored after PLOT is called.

```
LD   B,11
LD   C,11
LD   HL,(T_ADDR)     ;T_ADDR = $4030
PUSH HL              ;SAVE T-ADDR
LD   A,$98
LD   (T_ADDR),A
CALL PLOT
POP  HL
LD   (T_ADDR),HL     ;RESTORE T-ADDR
```

PRINT $0010

Prints the character to the screen that is stored in the A register. Prints the character at the next print location. Row position is loaded into the B register and Column position is loaded into the C register.

```
LD   A,XX
CALL PRINT
```

PRINTAT $08F5

Moves the print position. Row position is loaded into the B register and Column position is loaded into the C register.

```
LD   B,$10
LD   C,$10
CALL PRINTAT
```

SCROLL $0C0E

Scrolls the screen up one line.

```
CALL SCROLL
```

SLOW $0F2B

Switches the display to SLOW mode.

```
CALL SLOW
```

Appendix II
Included Example Programs

array1d.asm	- test of 1 dim array
array2d.asm	- test of 2 dim array
divide.asm	- test of division
hello.asm	- Hello World
hexprt.asm	- print hex number
movechar.asm	- Keyboard input, PRINT AT
multiply.asm	- test of mulitplication
pause.asm	- Testing PAUSE call
plot.asm	- PLOT and UNPLOT
print.asm	- PRINT
printat.asm	- PRINT AT
rnd.asm	- Simple random number generator
screen1.asm	- Write directly to screen
screen2.asm	- Write directly to screen with string
scroll.asm	- Testing SCROLL call
type.asm	- Keyboard input