

# Retro Programming

<http://www.retroprogramming.com/feeds/posts/default?alt=rss>

Are you the publisher? [Claim](#) or [contact us](#) about this channel

Embed this content in your HTML



[Search](#)

Report adult content:



click to rate



Account: ([login](#))



More Channels



Showcase

[Browse the Latest Snapshot](#)

[Browsing All Articles \(44 Articles\)](#)

[Live Browser](#)

[older](#) | [1](#) | **(Page 2)** | [3](#) | [newer](#)

➔ **04/15/12--12:37: Itsy Forth: Implementing the Primitives**

0

0



**Itsy Forth** is a tiny subset of the Forth programming language. So far we've looked at the [Forth outer interpreter](#), [inner interpreter and dictionary](#). This time we'll define the words required to complete the interpreter.

## Peek and Poke

The Forth words to read and write memory are @ and !:

➔ @ - ( **addr -- x** ) read **x** from **addr**

➔ ! - ( **x addr --** ) store **x** at **addr**

➔ c@ - ( **addr -- char** ) read **char** from **addr**

( before -- after ) shows the contents of the stack before and after the word executes. Here's how @, c@ and ! are implemented. Remember we're keeping the top element of the data stack in the

## ➤ Channel Catalog

[Subsection Catalog](#)

## ➤ Articles on this Page

(showing articles 21 to 40 of 44)

[04/15/12--12:37: Itsy Forth: Impleme...](#)

[06/23/12--13:55: Itsy Forth: The Com...](#)

[08/23/12--13:35: Mouse: a Language f...](#)

[09/04/12--17:30: Itsy: Documenting t...](#)

[09/18/12--15:10: Programming Editors...](#)

[07/04/13--16:56: Silicon Dreams & Th...](#)

[08/01/13--11:24: ZX Spectrum Koch \(L...](#)

[10/18/13--16:45: Video Gaming 1979-1...](#)

[11/11/13--15:35: The Centre for Comp...](#)

[02/09/14--14:46: The Spring 2014 Cor...](#)

[03/15/14--17:05: Plotting the Mandel...](#)

[01/03/14--06:00: Fast Z80 Bit Reversal](#)

[12/08/14--13:29: Z80 Size Programmin...](#)

[12/15/14--14:09: Z80 Size Programmin...](#)

[03/30/15--08:29: Z80 Size Programmin...](#)

[04/06/15--15:16: Z80 Size Programmin...](#)

[07/26/15--23:18: Z80 Size Programmin...](#)

[10/03/15--17:56: The Matrix Digital ...](#)

[05/27/16--13:31: Langton's Ant for t...](#)

[05/29/16--01:19: Divide and Conquer ...](#)

bx register.

```
primitive '@',fetch
mov bx,word[bx]
jmp next
```

```
primitive '!',store
pop word[bx]
pop bx
jmp next
```

```
primitive 'c@',c_fetch
mov bl,byte[bx]
mov bh,0
jmp next
```

## Manipulating the Stack

- **drop - ( x -- ) remove x from the stack**
- **dup - ( x -- x x ) add a copy of x to the stack**
- **swap - ( x y -- y x ) exchange x and y**
- **rot - ( x y z -- y z x ) rotate x, y and z**

```
primitive 'drop',drop
pop bx
jmp next
```

```
primitive 'dup',dupe
push bx
jmp next
```

```
primitive 'swap',swap
pop ax
push bx
```

(showing articles 21 to 40 of 44)

```
xchg ax,bx
jmp next
```

```
primitive 'rot',rote
pop dx
pop ax
push dx
push bx
xchg ax,bx
jmp next
```

## Flow Control

if, else, then, begin and again all compile to `branch` or `0branch`.

➔ `0branch - ( x -- ) jump if x is zero`

➔ `branch - ( -- ) unconditional jump`

➔ `execute - ( xt -- ) call the word at xt`

➔ `exit - ( -- ) return from the current word`

The destination address for the jump is compiled in the cell straight after the `branch` or `0branch` instruction. `execute` stores the return address on the return stack and `exit` removes it.

```
primitive '0branch',zero_branch
lodsw
test bx,bx
jne zerob_z
xchg ax,si
zerob_z pop bx
jmp next

primitive 'branch',branch
mov si,word[si]
jmp next
```

```
primitive 'execute',execute
mov di,bx
pop bx
jmp word[di]

primitive 'exit',exit
mov si,word[bp]
inc bp
inc bp
jmp next
```

## Variables and Constants

- ➔ **tib - ( -- addr ) address of the input buffer**
- ➔ **#tib - ( -- addr ) number of characters in the input buffer**
- ➔ **>in - ( -- addr ) next character in input buffer**
- ➔ **state - ( -- addr ) true = compiling, false = interpreting**
- ➔ **dp - ( -- addr ) first free cell in the dictionary**
- ➔ **base - ( -- addr ) number base**
- ➔ **last - ( -- addr ) the last word to be defined**

```
constant 'tib',t_i_b,32768
variable '#tib',number_t_i_b,0
variable '>in',to_in,0
variable 'state',state,0
variable 'dp',dp,freemem
variable 'base',base,10
```

```

variable 'last',last,final

; execution token for constants
doconst push bx
        mov bx,word[di+2]
        jmp next

; execution token for variables
dovar  push bx
        lea bx,[di+2]
        jmp next

```

## Compilation

- ➔ , - ( x -- ) compile *x* to the current definition
- ➔ c, - ( char -- ) compile *char* to the current definition
- ➔ lit - ( -- ) push the value in the cell straight after *lit*

```

primitive ',',comma
mov ax,word[val_dp]
xchg ax,bx
add word[val_dp],2
mov word[bx],ax
pop bx
jmp next

```

```

primitive 'c,',c_comma
mov ax,word[val_dp]
xchg ax,bx
inc word[val_dp]
mov byte[bx],al
pop bx
jmp next

```

```

primitive 'lit',lit

```

```
push bx
lodsw
xchg ax,bx
jmp next
```

## Maths / Logic

➡ **+ - ( x y -- z ) calculate  $z=x+y$  then return z**

➡ **= - ( x y -- flag ) return true if  $x=y$**

```
primitive '+',plus
pop ax
add bx,ax
jmp next
```

```
primitive '=',equals
pop ax
sub bx,ax
sub bx,1
sbb bx,bx
jmp next
```

## Handling Strings

➡ **count - ( addr -- addr2 len ) *addr* contains a counted string. Return the address of the first character and the string's length**

➡ **>number - ( double addr len -- double2 addr2 len2 ) convert string to number**

*addr* contains a string of *len* characters which >number attempts to convert to a number using the current number base. >number returns the portion of the string which can't be converted, if any.

If you're a Forth purist this is where Itsy starts to get ugly :-)

```

        primitive 'count',count
        inc bx
        push bx
        mov bl,byte[bx-1]
        mov bh,0
        jmp next

        primitive '>number',to_number
        pop di
        pop cx
        pop ax
to_numl test bx,bx
        je to_numz
        push ax
        mov al,byte[di]
        cmp al,'a'
        jc to_nums
        sub al,32
to_nums cmp al,'9'+1
        jc to_numg
        cmp al,'A'
        jc to_numh
        sub al,7
to_numg sub al,48
        mov ah,0
        cmp al,byte[val_base]
        jnc to_numh
        xchg ax,dx
        pop ax
        push dx
        xchg ax,cx
        mul word[val_base]
        xchg ax,cx
        mul word[val_base]
        add cx,dx
        pop dx
        add ax,dx
        dec bx
        inc di

```

```
        jmp to_num1
to_numz push ax
to_numh push cx
        push di
        jmp next
```

## Terminal Input / Output

➔ **accept - ( addr len -- len2 ) read a string from the terminal**

➔ **emit - ( char -- ) display *char* on the terminal**

➔ **word - ( char -- addr ) parse the next word in the input buffer**

`accept` reads a string of characters from the terminal. The string is stored at *addr* and can be up to *len* characters long. `accept` returns the actual length of the string.

`word` reads the next word from the terminal input buffer, delimited by *char*. The address of a counted string is returned. The string length will be 0 if the input buffer is empty.

```
        primitive 'accept',accept
        pop di
        xor cx,cx
acceptl call getchar
        cmp al,8
        jne acceptn
        jcxz acceptb
        call outchar
        mov al,' '
        call outchar
        mov al,8
        call outchar
        dec cx
        dec di
        jmp acceptl
acceptn cmp al,13
```



```
        je acceptz
        cmp cx,bx
        jne accepts
acceptb  mov al,7
        call outchar
        jmp acceptl
accepts  stosb
        inc cx
        call outchar
        jmp acceptl
acceptz  jcxz acceptb
        mov al,13
        call outchar
        mov al,10
        call outchar
        mov bx,cx
        jmp next

getchar  mov ah,7
        int 021h
        mov ah,0
        ret

outchar  xchg ax,dx
        mov ah,2
        int 021h
        ret

        primitive 'word',word
        mov di,word[val_dp]
        push di
        mov dx,bx
        mov bx,word[val_t_i_b]
        mov cx,bx
        add bx,word[val_to_in]
        add cx,word[val_number_t_i_b]
wordf    cmp cx,bx
        je wordz
        mov al,byte[bx]
        inc bx
        cmp al,dl
        je wordf
```

```

wordc  inc di
        mov byte[di],al
        cmp cx,bx
        je wordz
        mov al,byte[bx]
        inc bx
        cmp al,dl
        jne wordc
wordz  mov byte[di+1],32
        mov ax,word[val_dp]
        xchg ax,di
        sub ax,di
        mov byte[di],al
        sub bx,word[val_t_i_b]
        mov word[val_to_in],bx
        pop bx
        jmp next

        primitive 'emit',emit
        xchg ax,bx
        call outchar
        pop bx
        jmp next

```

## Searching the Dictionary

➡ **find - ( addr -- addr2 flag ) look up word in the dictionary**

`find` looks in the Forth dictionary for the word in the counted string at *addr*. One of the following will be returned:

- ➡ **flag = 0, addr2 = counted string - if word not found**
- ➡ **flag = 1, addr2 = call address if word is immediate**
- ➡ **flag = -1, addr2 = call address if word is not immediate**

```
primitive 'find',find
mov di,val_last
findl  push di
       push bx
       mov cl,byte[bx]
       mov ch,0
       inc cx
findc  mov al,byte[di+2]
       and al,07Fh
       cmp al,byte[bx]
       je findm
       pop bx
       pop di
       mov di,word[di]
       test di,di
       jne findl
findnf push bx
       xor bx,bx
       jmp next
findm  inc di
       inc bx
       loop findc
       pop bx
       pop di
       mov bx,1
       inc di
       inc di
       mov al,byte[di]
       test al,080h
       jne findi
       neg bx
findi  and ax,31
       add di,ax
       inc di
       push di
       jmp next
```

## Initialisation


➡ **abort - ( -- ) initialise Itsy then jump to** interpret

abort initialises the stacks and a few variables before running the outer interpreter. When Itsy first runs it jumps to abort to set up the system.

```
primitive 'abort',abort
xor ax,ax
mov word[val_state],ax
mov word[val_to_in],ax
mov word[val_number_t_i_b],ax
xchg ax,bp
mov sp,-256
mov si,xt_interpret+2
jmp next
```

## Up and Running?

Itsy is now around 900 bytes and it's time to give the interpreter a quick test run:



```
C:\>itsy

16 base !
79 73 74 49
emit emit emit emit
Itsy_
```

Everything seems to be working fine. Next we'll define the compiler so we can continue building Itsy from the Itsy prompt :-)

➔ **06/23/12--13:55: Itsy Forth: The Compiler**

0



0



**Itsy Forth** is a 1kB subset of the Forth programming language. Itsy was developed top-down, implementing only the functions required to get the compiler up and running. So far we've looked at the following:

➔ **The Outer (text) Interpreter**

➔ **The Inner (address) Interpreter and Dictionary**

➔ **The Primitives**

Next we'll define the words to complete the compiler.

## Colon Definitions

➔ **: - ( -- ) define a new Forth word, taking the name from the input buffer**

➔ **; - ( -- ) complete the Forth word being compiled**

: sets state to true to enter compile mode then creates a header for the new word. ; adds exit to the end of the word then sets state to false to end compile mode.

For example, : here dp @ ; creates a new Forth word which returns the contents of the variable dp.

```
: :
-1 state !
create
(;code)
docolon dec bp
      dec bp
      mov word[bp],si
      lea si,[di+2]
      jmp next
```

```

: ;
['] exit ,
0 state !
; immediate

```

## Creating Headers

- ➔ **create - ( -- ) build a header for a new word in the dictionary, taking the name from the input buffer**
- ➔ **(;code) - ( -- ) replace the *xt* of the word being defined with a pointer to the code immediately following (;code)**

create adds a new header to the dictionary which includes a link to the previous entry, a name and execution token (*xt*). The *xt* initially points to *dovar* but can be modified using (;code).

For example, `: variable create 0 , ;` creates a new Forth word to define variables (*dovar* is the default *xt* for created words).

```

: create
dp @ last @ , last !
32 word count
+ dp ! 0 ,
(;code)
dovar  push bx
      lea bx,[di+2]
      jmp next

      primitive ' (;code)',do_semi_code
      mov di,word[val_last]
      mov al,byte[di+2]
      and ax,31
      add di,ax
      mov word[di+3],si
      mov si,word[bp]
      inc bp
      inc bp
      jmp next

```

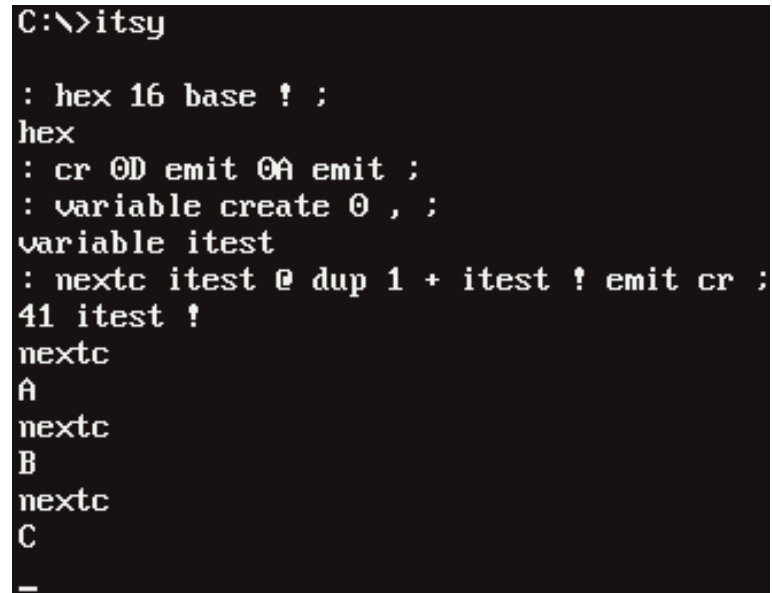
## Constants

➔ **constant - ( x -- ) create a new constant with the value x, taking the name from the input buffer**

For example, `0 constant false` adds a new constant to the dictionary. When executed, `false` will push 0 on the stack.

```
: constant
create ,
(;code)
doconst push bx
      mov bx,word[di+2]
      jmp next
```

## Testing the Compiler



```
C:\>itsy

: hex 16 base ! ;
hex
: cr 0D emit 0A emit ;
: variable create 0 , ;
variable itest
: nextc itest @ dup 1 + itest ! emit cr ;
41 itest !
nextc
A
nextc
B
nextc
C
_
```

It's time to give Itsy a quick test run. First we implement a few standard words: `hex` to switch to base 16, `cr` to move the cursor to the next line and `variable` to define new variables.

Next a simple test. We add a variable `itest` initialised to 041h (ASCII 'A') and a procedure to display and increment `itest`. Then the moment of truth... A B C it works!

## Itsy Forth: The Next Step

What's next for Itsy Forth? First I'd like to implement the ANS core wordset from the Itsy prompt, then perhaps experiment with compiling to native code. In the meantime, here's the code for the current version of Itsy:

### macros.asm

```
%define link 0
%define immediate 080h

%macro head 4
%%link dw link
%define link %%link
%strlen %%count %1
db %3 + %%count,%1
xt_ %+ %2 dw %4
%endmacro

%macro primitive 2-3 0
head %1,%2,%3,$+2
%endmacro

%macro colon 2-3 0
head %1,%2,%3,docolon
%endmacro

%macro constant 3
head %1,%2,0,doconst
val_ %+ %2 dw %3
%endmacro

%macro variable 3
head %1,%2,0,dovar
val_ %+ %2 dw %3
%endmacro
```



## itsy.asm

```
%include "macros.asm"

        org 0100h
        jmp xt_abort+2

; -----
; Variables
; -----

        variable 'state',state,0

        variable '>in',to_in,0

        variable '#tib',number_t_i_b,0

        variable 'dp',dp,freemem

        variable 'base',base,10

        variable 'last',last,final

        constant 'tib',t_i_b,32768

; -----
; Initialisation
; -----

        primitive 'abort',abort
        mov ax,word[val_number_t_i_b]
        mov word[val_to_in],ax
        xor bp,bp
        mov word[val_state],bp
        mov sp,-256
        mov si,xt_interpret+2
        jmp next

; -----
; Compilation
; -----
```

```
primitive ',',comma
mov di,word[val_dp]
xchg ax,bx
stosw
mov word[val_dp],di
pop bx
jmp next

primitive 'lit',lit
push bx
lodsw
xchg ax,bx
jmp next

; -----
; Stack
; -----

primitive 'rot',rote
pop dx
pop ax
push dx
push bx
xchg ax,bx
jmp next

primitive 'drop',drop
pop bx
jmp next

primitive 'dup',dupe
push bx
jmp next

primitive 'swap',swap
pop ax
push bx
xchg ax,bx
jmp next

; -----
; Maths / Logic
```

```
; -----  
  
    primitive '+',plus  
    pop ax  
    add bx,ax  
    jmp next  
  
    primitive '=',equals  
    pop ax  
    sub bx,ax  
    sub bx,1  
    sbb bx,bx  
    jmp next  
  
; -----  
; Peek and Poke  
; -----  
  
    primitive '@',fetch  
    mov bx,word[bx]  
    jmp next  
  
    primitive '!',store  
    pop word[bx]  
    pop bx  
    jmp next  
  
; -----  
; Inner Interpreter  
; -----  
  
next    lodsw  
        xchg di,ax  
        jmp word[di]  
  
; -----  
; Flow Control  
; -----  
  
    primitive '0branch',zero_branch  
    lodsw  
    test bx,bx
```

```

    jne zerob_z
    xchg ax,si
zerob_z pop bx
        jmp next

    primitive 'branch',branch
    mov si,word[si]
    jmp next

    primitive 'execute',execute
    mov di,bx
    pop bx
    jmp word[di]

    primitive 'exit',exit
    mov si,word[bp]
    inc bp
    inc bp
    jmp next

; -----
; String
; -----

    primitive 'count',count
    inc bx
    push bx
    mov bl,byte[bx-1]
    mov bh,0
    jmp next

    primitive '>number',to_number
    pop di
    pop cx
    pop ax
to_num1 test bx,bx
        je to_numz
    push ax
    mov al,byte[di]
    cmp al,'a'
    jc to_nums
    sub al,32
```

```

to_nums cmp al,'9'+1
        jc to_numg
        cmp al,'A'
        jc to_numh
        sub al,7
to_numg sub al,48
        mov ah,0
        cmp al,byte[val_base]
        jnc to_numh
        xchg ax,dx
        pop ax
        push dx
        xchg ax,cx
        mul word[val_base]
        xchg ax,cx
        mul word[val_base]
        add cx,dx
        pop dx
        add ax,dx
        dec bx
        inc di
        jmp to_num1
to_numz push ax
to_numh push cx
        push di
        jmp next

; -----
; Terminal Input / Output
; -----

        primitive 'accept',accept
        pop di
        xor cx,cx
accept1 call getchar
        cmp al,8
        jne acceptn
        jcxz acceptb
        call outchar
        mov al,' '
        call outchar
        mov al,8

```

```
        call outchar
        dec cx
        dec di
        jmp acceptl
acceptn  cmp al,13
        je acceptz
        cmp cx,bx
        jne accepts
acceptb  mov al,7
        call outchar
        jmp acceptl
accepts  stosb
        inc cx
        call outchar
        jmp acceptl
acceptz  jcxz acceptb
        mov al,13
        call outchar
        mov al,10
        call outchar
        mov bx,cx
        jmp next

        primitive 'word',word
        mov di,word[val_dp]
        push di
        mov dx,bx
        mov bx,word[val_t_i_b]
        mov cx,bx
        add bx,word[val_to_in]
        add cx,word[val_number_t_i_b]
wordf    cmp cx,bx
        je wordz
        mov al,byte[bx]
        inc bx
        cmp al,dl
        je wordf
wordc    inc di
        mov byte[di],al
        cmp cx,bx
        je wordz
        mov al,byte[bx]
```

```
        inc bx
        cmp al,dl
        jne wordc
wordz   mov byte[di+1],32
        mov ax,word[val_dp]
        xchg ax,di
        sub ax,di
        mov byte[di],al
        sub bx,word[val_t_i_b]
        mov word[val_to_in],bx
        pop bx
        jmp next

        primitive 'emit',emit
        xchg ax,bx
        call outchar
        pop bx
        jmp next

getchar mov ah,7
        int 021h
        mov ah,0
        ret

outchar xchg ax,dx
        mov ah,2
        int 021h
        ret

; -----
; Dictionary Search
; -----

        primitive 'find',find
        mov di,val_last
findl   push di
        push bx
        mov cl,byte[bx]
        mov ch,0
        inc cx
findc   mov al,byte[di+2]
        and al,07Fh
```

```

        cmp al,byte[bx]
        je findm
        pop bx
        pop di
        mov di,word[di]
        test di,di
        jne findl
findnf  push bx
        xor bx,bx
        jmp next
findm   inc di
        inc bx
        loop findc
        pop bx
        pop di
        mov bx,1
        inc di
        inc di
        mov al,byte[di]
        test al,080h
        jne findi
        neg bx
findi   and ax,31
        add di,ax
        inc di
        push di
        jmp next

; -----
; Colon Definition
; -----

        colon ':',colon
        dw xt_lit,-1,xt_state,xt_store,xt_create
        dw xt_do_semi_code
docolon dec bp
        dec bp
        mov word[bp],si
        lea si,[di+2]
        jmp next

        colon ';',semicolon,immediate

```



```

        dw xt_lit,xt_exit,xt_comma,xt_lit,0,xt_state
        dw xt_store,xt_exit

; -----
; Headers
; -----

        colon 'create',create
        dw xt_dp,xt_fetch,xt_last,xt_fetch,xt_comma
        dw xt_last,xt_store,xt_lit,32,xt_word,xt_count
        dw xt_plus,xt_dp,xt_store,xt_lit,0,xt_comma
        dw xt_do_semi_code
dovar   push bx
        lea bx,[di+2]
        jmp next

        primitive ' (;code)',do_semi_code
        mov di,word[val_last]
        mov al,byte[di+2]
        and ax,31
        add di,ax
        mov word[di+3],si
        mov si,word[bp]
        inc bp
        inc bp
        jmp next

; -----
; Constants
; -----

        colon 'constant',constant
        dw xt_create,xt_comma,xt_do_semi_code
doconst push bx
        mov bx,word[di+2]
        jmp next

; -----
; Outer Interpreter
; -----

final:

```

```

colon 'interpret',interpret
interpret dw xt_number_t_i_b,xt_fetch,xt_to_in,xt_fetch
           dw xt_equals,xt_zero_branch,intpar,xt_t_i_b
           dw xt_lit,50,xt_accept,xt_number_t_i_b,xt_store
           dw xt_lit,0,xt_to_in,xt_store
intpar dw xt_lit,32,xt_word,xt_find,xt_dupe
        dw xt_zero_branch,intnf,xt_state,xt_fetch
        dw xt_equals,xt_zero_branch,intexc,xt_comma
        dw xt_branch,intdone
intexc dw xt_execute,xt_branch,intdone
intnf dw xt_dupe,xt_rote,xt_count,xt_to_number
       dw xt_zero_branch,intskip,xt_state,xt_fetch
       dw xt_zero_branch,intnc,xt_last,xt_fetch,xt_dupe
       dw xt_fetch,xt_last,xt_store,xt_dp,xt_store
intnc dw xt_abort
intskip dw xt_drop, xt_drop, xt_state, xt_fetch
         dw xt_zero_branch,intdone,xt_lit,xt_lit,xt_comma
         dw xt_comma
intdone dw xt_branch,interpret

freemem:

```

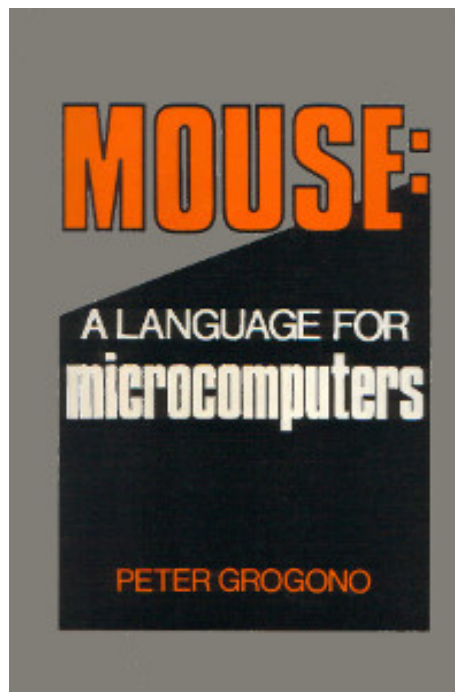
📌 **08/23/12--13:35: Mouse: a Language for Microcomputers by Peter Grogono**

0  0   

**Mouse** is a stack-based interpreted programming language descended from MUSYS, an earlier language for a DEC PDP/8 with 4096 words of memory.

In **Mouse: a Language for Microcomputers**, Grogono defines the language and develops an interpreter step-by-step as he introduces each feature of the language.

The first few chapters cover the basics: postfix expressions, variables, control structures, pointers and macros. Later chapters include two complete Mouse interpreters, one in Pascal, the other in Z80 assembly language.



Here's a brief summary of the Mouse language. Instructions pop their operands from and push their result on the stack. ( before -- after ) shows the stack effects of each operation.

## Maths / Logic

Mouse is looking pretty Forthlike so far! \ is the equivalent of MOD in Forth:

- ➔ + - ( x y -- z ) calculate z, the sum of x+y
- ➔ - - ( x y -- z ) calculate z, the difference of x-y
- ➔ \* - ( x y -- z ) calculate z, the product of x\*y
- ➔ / - ( x y -- z ) calculate z, the quotient of x/y
- ➔ \ - ( x y -- z ) calculate z, the remainder of x/y
- ➔ < - ( x y -- z ) if x<y then z is true, otherwise false
- ➔ = - ( x y -- z ) if x=y then z is true, otherwise false
- ➔ > - ( x y -- z ) if x>y then z is true, otherwise false

## Input / Output

Only ? doesn't have an exact equivalent in Forth. ? ' is the same as Forth's KEY. ! ' is the same as EMIT and ! is .:

- ➔ ? - ( -- x ) read a number x from the keyboard
- ➔ ? ' - ( -- x ) read a character x from the keyboard
- ➔ ! - ( x -- ) display a number x
- ➔ ! ' - ( x -- ) display a character x

➔ **"..." - ( -- ) display the quoted string**

## Peek and Poke

: is the equivalent of ! in Forth. . is the equivalent of @:

➔ **: - ( x addr -- ) store x in address *addr***

➔ **. - ( addr -- x ) read x from address *addr***

## Control Structures

[ ... | ... ] is similar to Forth's IF ... ELSE ... THEN.

( ... ↑ ... ) is similar to Forth's BEGIN ... WHILE ... REPEAT:

➔ **[ - ( x -- ) if x is false, jump to the matching | or ]**

➔ **| - ( -- ) jump to the matching ] (not always implemented)**

➔ **] - ( -- ) end a [ ... | ... ] structure**

➔ **( - ( -- ) start a loop**

➔ **↑ - ( x -- ) exit loop if x is false (often rendered as ^)**

➔ **) - ( -- ) end loop, jump back to matching (**

➔ **~ - ( -- ) the remainder of the line is a comment**

## Macro Definitions

➔ **#x; - ( -- ) call macro x**

➔ **\$x - ( -- ) define macro x**

➔ **@ - ( -- ) end macro definition**

➔ **% - ( x -- z ) access macro parameter**

Macros are the Mouse equivalent of subroutines. A macro is defined by \$x... @ and called with #x;. Parameters can be passed between the macro name and semicolon. For example #x,7,5,9; will pass the parameters 7, 5 and 9 to x.

A macro accesses it's parameters using %: 1% for the first, 2% for the second, etc. A parameter is evaluated every time it's accessed and can be almost any valid Mouse code.

A macro has 26 local variables, A to Z. 26 macro names are available, A to Z.

## Example Code

Here are a few classic examples:

➔ **Hello World (which recently celebrated it's 40th birthday)**

➔ **Fibonacci Numbers (the typical bad example of recursion)**

➔ **Greatest Common Divisor (a better example of recursion)**

### Hello, World

```
"Hello, World!"$
```

Displays the string *Hello, World*. The exclamation mark isn't printed. An exclamation in a string instructs the interpreter to print a line break. All Mouse programs end with \$.

### Fibonacci Numbers

```
$F
1% N:                ~ store parameter in N
N. 2 < [ N. ]         ~ if N < 2 then return N
N. 1 > [ #F, N. 1 - ;  ~ otherwise calculate F(N-1)
                  #F, N. 2 - ; ~ | and F(N-2)
                  + ]       ~ | and return their sum
@
```

F calculates Fibonacci numbers using the recurrence relation  $F_n = F_{n-1} + F_{n-2}$  with  $F_0 = 0$ ,  $F_1 = 1$ . Note this is really slow,  $O(\phi^n)$ . Calculating  $F_{40}$  takes 6 minutes. There are better ways to calculate Fibonacci numbers!

## Greatest Common Divisor

```
$G
1% X: 2% Y:           ~ store parameters
X. Y. = [ X. ]         ~ if X = Y then GCD = X
X. Y. > [ #G, X. Y. - , Y. ; ] ~ otherwise subtract the
X. Y. < [ #G, Y. X. - , X. ; ] ~ | smallest from the
@                       ~ | largest and recurse
```

Another example of recursion. G calculates the GCD using Dijkstra's method.

## Further Reading

- ➔ **Grogono, Peter***Mouse: A Language for Microcomputers*. New York: Petrocelli Books, 1983.
- ➔ **David G Simpson** has published several [interpreters for Mouse](#).

➔ **09/04/12--17:30: [Itsy: Documenting the Bit-Twiddling & Voodoo Magic](#)** 0  0   

Over the last few months I've been developing Itsy, a tiny interpreter for a subset of Forth. An overview can be found in the following posts:

- ➔ [Itsy-Forth: the Outer Interpreter of a 1K Tiny Compiler](#)
- ➔ [Itsy-Forth: the Dictionary and Inner Interpreter](#)

## ➤ Itsy Forth: Implementing the Primitives

### ➤ Itsy Forth: The Compiler

To save time I've described the system as a whole while skipping some of the implementation details. Mike Adams noticed my omission and performed a complete analysis of Itsy, fully commenting the code.

Mike's analysis will make it much easier to change the threading model and port Itsy to a microcontroller when I finally get round to it. Thanks Mike.

Here's Itsy with all the bit-twiddling hacks and voodoo magic documented by Mike:

## macros.asm

```
; Itsy Forth - Macros
;   Written by John Metcalf
;   Commentary by Mike Adams
;
; Itsy Forth was written for use with NASM, the "Netwide Assembler"
; (http://www.nasm.us/). It uses a number of macros to deal with the tedium
; of generating the headers for the words that are defined in Itsy's source
; code file. The macros, and the explanations of what they're doing, are
; listed below:
;-----
; First, two variables are defined for use by the macros:
; link is the initial value for the first link field that'll
; be defined. It's value will be updated with each header
; that's created.
%define link 0

; A bitmask that'll be called "immediate" will be used to
; encode the flag into the length bytes of word names in order
; to indicate that the word will be of the immediate type.
%define immediate 080h
;-----
; The first macro defined is the primary one used by the others, "head".
; It does the lion's share of the work for the other macros that'll be
; defined afterwards. Its commands perform the following operations:

; The first line of the macro declares it's name as "head".
```

```
; The 4 in this line signifies that it expects to receive
; 4 parameters when it's invoked: the string that will be the
; word's name and will be encoded into the header along with
; the string's name; an "execution tag" name that will have the
; prefix "xt_" attached to it and will be used as a label for
; the word's code field; a flag that will be 080h if the word
; will be immediate and a 0 otherwise; and the label for the
; word's runtime code, whose address will be put into the
; word's code field.
%macro head 4

; Okay, what we're doing in this odd-looking bit of code is
; declaring a variable called "%link" that's local only to this
; macro and is independent of the earlier variable we declared
; as "link". It's a label that will represent the current
; location in the object code we're creating. Then we lay down
; some actual object code, using the "dw" command to write the
; current value of "link" into the executable file.
    %link dw link

; Here's one of the tricky parts. We now redefine the value of
; "link" to be whatever the current value of "%link" is, which
; is basically the address of the link field that was created
; during this particular use of this macro. That way, the next
; time head is called, the value that will be written into the
; code in the "dw" command above will be whatever the value of
; "%link" was during THIS use of the macro. This way, each time
; head is called, the value that'll be written into the new
; link field will be the address that was used for the link
; field the previous time head was called, which is just how
; we want the link fields to be in a Forth dictionary. Note that
; the first time that head is called, the value of link was
; predefined as 0, so that the link field of the first word in
; the dictionary will contain the value of 0 to mark it as
; being the first word in the dictionary.
%define link %link

; Now the name field. The first argument passed to head is the
; string defining the new word's name. The next line in the macro
; measures the length of the string (the "%1" tells it that it's
; supposed to look at argument #1) and assigns it to a macro-local
; variable called "%count".
%strlen %count %1

; In this next line, we're writing data into the object code on
; a byte-by-byte basis. We first write a byte consisting of the
; value of argument 3 (which is 080h if we're writing the header
; for an immediate word or a 0 otherwise) added to the length of
; the name string to produce the length byte in the header. Then
```



```

; we write the name string itself into the file.
db%3 + %%count,%1

; Okay, don't get confused by the "+" in this next line. Take
; careful note of the spaces; the actual command is "%+", which
; is string concatenation, not numeric addition. We're going to
; splice a string together. The first part consists of the "xt_",
; then we splice the macro's 2nd argument onto it. The resulting
; string is used as the header's "execution tag", the address of
; it's code field. This label is then used for the "dw" command
; that writes the value of argument #4 (the address of the word's
; runtime code) into the header's code field.
    xt_ %+ %2dw%4

; As you might guess, the next line marks the end of the
; macro's definition. The entire header's been defined at this
; point, and we're now ready for the data field, whether it's
; composed of assembly code, a list of Forth words, or the
; numeric data for a variable or constant.
%endmacro

; For example, calling head with the following line:
;
;     head,'does> ',does,080h,docolon
;
; will produce the following header code...
;
;         dw (address of link of previous header)
;         db 085h,'does>'
;     xt_does  dw docolon
;
; ...and records the address of this header's link field so that it can
; be written into the link field of the next word, just as the address
; of the previous link field was written into this header.
; This method saves the programmer a lot of tedium in manually generating
; the code for word headers when writing a Forth system's kernel in
; assembly language. Note that argument #2 is surrounded by single quotes.
; That's the format that the assembler expects to see when being told to
; lay down a string of characters byte-by-byte in a db command, so they
; have to be present when they're given as an arg to this macro so that
; the macro puts them in their proper place.

;-----
; The next macro is called "primitive", and is used for setting up a header
; for a word written in assembly language.
;
; Here we declare the definition of the macro called "primitive".
; Note, though, the odd manner in which the number of required
; arguments is stated. Yes, that really does mean that it can

```

```
; take from 2 to 3 arguments. Well, what does it do if the user
; only gives it 2? That's what that 0 is: the default value that's
; to be used for argument #3 if the user doesn't specify it. Most
; of the time he won't; the only time arg #3 will be specifically
; given will be if the user is defining an immediate word.
```

```
%macro primitive 2-30
```

```
; All primitive does is to pass its arguments on to head, which
; does most of the actual work. It passes on the word name and
; the execution tag name as-is. Parameter #3 will be given the
; default value of 0 unless the user specifically states it.
; This is meant to allow the user to add "immediate" to the
; macro invocation to create an immediate word. The 4th arg,
; "$+2", means that when head goes to write the address of the
; run-time code into the code field, the address it's going to
; use will be 2 bytes further along than the code field address,
; i.e. the address of the start of the code immediately after
; the code field. (The "$" symbol is used by most assemblers
; to represent the address of the code that's currently being
; assembled.)
```

```
    head %1,%2,%3,$+2
```

```
; End of the macro definition.
```

```
%endmacro
```

```
;-----
; The macro "colon" operates very similarly to "primitive", except that
; it's used for colon definitions:
;
```

```
; Declare the macro, with 2 to 3 arguments, using 0 for the default
; value of arg #3 if one isn't specifically given.
```

```
%macro colon 2-30
```

```
; Pass the args on to head, using docolon as the runtime code.
```

```
    head %1,%2,%3,docolon
```

```
; End of macro definition.
```

```
%endmacro
```

```
;-----
; The rest of the macros all require a specific number of arguments, since
; none of them have the option of being immediate. This one defines
; a constant:
```

```
; Macro name is, unsurprisingly, "constant", and gets 3 arguments.
; As with head and primitive, the first 2 are the word's name and
; the label name that'll be used for the word. The third argument
; is the value that we want the constant to hold.
```

```
%macro constant 3
```

```
; Use the head macro. Args 1 and 2, the names, get passed on as-is.
; Constants are never defined as immediate (though it's an intriguing
; idea; a constant whose value is one thing when compiling and
; another when interpreting might be useful for something), so arg #3
; passed on to head is always a 0, and arg #4 will always be doconst,
; the address of the runtime code for constants.
```

```
    head %1,%2,0,doconst
```

```
; Similar to the way that the label is created for the execution
; tags, here we create a label for the data field of the constant,
; though this time we're prefixing the name with "val_" instead
; of the "xt_" used for the execution tags. Then we use a dw to
; write constant's arg #3, the constant's value, into the code.
```

```
    val_ %+ %2dw%3
```

```
; End of the definition.
```

```
%endmacro
```

```
;-----
; The macro for variables is very similar to the one for constants.
```

```
; Macro name "variable", 3 arguments, with arg #3 being the
; initial value that will be given to the variable.
```

```
%macro variable 3
```

```
; Just like in "constant", except that the runtime code is dovar.
```

```
    head %1,%2,0,dovar
```

```
; Exact same line as used in "constant", with the same effects.
```

```
    val_ %+ %2dw%3
```

```
; End of the definition.
```

```
%endmacro
```

```
;-----
;
; That's the last of the macros. They're accessed through the
; "%include macros.asm" command near the beginning of Itsy's
; source code file. Or, if you prefer, you can remove the
; %include command and splice the above code directly
; into itsy.asm in its place.
;
;-----
```

## itsy.asm

```

; Itsy Forth
;   Written by John Metcalf
;   Commentary by John Metcalf and Mike Adams
;
; Itsy Forth was written for use with NASM, the "Netwide Assembler"
; that's available for free download (http://www.nasm.us/).
; The command line for assembling Itsy is:
;
;     nasm itsy.asm -fbin -o itsy.com
;
; If you wish to have an assembly listing, give it this command:
;
;     nasm itsy.asm -fbin -l itsy.lst -o itsy.com
;
;-----
; Implementation notes:
;
; Register Usage:
;   sp - data stack pointer.
;   bp - return stack pointer.
;   si - Forth instruction pointer.
;   di - pointer to current XT (CFA of word currently being executed).
;   bx - TOS (top of data stack). The top value on the data stack is not
;         actually kept on the CPU's data stack. It's kept in the BX register.
;         Having it in a register like this speeds up the operation of
;         the primitive words. They don't have to take the time to pull a
;         value off of the stack; it's already in a register where it can
;         be used right away!
;   ax, cd, dx - Can all be freely used for processing data. The other
;         registers can still be used also, but only with caution. Their
;         contents must be pushed to the stack and then restored before
;         exiting from the word or calling any other Forth words. LOTS of
;         potential for program crashes if you don't do this correctly.
;         The notable exception is the DI register, which can (and is, below)
;         used pretty freely in assembly code, since the concept of a pointer
;         to the current CFA is rather irrelevant in assembly.
;
;
; Structure of an Itsy word definition:
;   # of
;   Bytes:  Description:
;   -----
;   2      Link Field. Contains the address of the link field of the
;           definition preceding this one in the dictionary. The link
;           field of the first def in the dictionary contains 0.
;   Varies Name Field. The first byte of the name field contains the length
;           of the name; succeeding bytes contain the ASCII characters of
;           the name itself. If the high bit of the length is set, the

```

```

;          definition is tagged as being an "immediate" word.
;      2      Code Field. Contains the address of the executable code for
;              the word. For primitives, this will likely be the address
;              of the word's own data field. Note that the header creation
;              macros automatically generate labels for the code field
;              addresses of the words they're used to define, though the
;              CFA labels aren't visible in the code shown below. The
;              assembler macros create labels, known as "execution tags"
;              or XTs, for the code field of each word.
;      Varies Data Field. Contains either a list of the code field addresses
;              of the words that make up this definition, or assembly-
;              language code for primitives, or numeric data for variables
;              and constants and such.

```

```

;-----
;
; Beginning of actual code.
;
; Include the definitions of the macros that are used in NASM to create
; the headers of the words. See macros.asm for more details.
;-----
%include"macros.asm"

;-----
; Define the location for the stack. -256 decimal = 0ff00h
;-----
stack0 equ -256

;-----
; Set the starting point for the executable code. 0100h is the standard
; origin for programs running under MS-DOS or its equivalents.
;-----
org0100h

;-----
; Jump to the location of the start of Itsy's initialization code.
;-----
jmp xt_abort+2

; -----
; System Variables
; -----

; state - ( -- addr ) true = compiling, false = interpreting
;          variable 'state',state,0

; >in - ( -- addr ) next character in input buffer
;          variable '>in',to_in,0

```

```

; #tib - ( -- addr ) number of characters in the input buffer
variable '#tib',number_t_i_b,0

; dp - ( -- addr ) first free cell in the dictionary
variable 'dp',dp,freemem

; base - ( -- addr ) number base
variable 'base',base,10

; last - ( -- addr ) the last word to be defined
; NOTE: The label "final:" must be placed immediately before
; the last word defined in this file. If new words are added,
; make sure they're either added before the "final:" label
; or the "final:" label is moved to the position immediately
; before the last word added.
variable 'last',last,final

; tib - ( -- addr ) address of the input buffer
constant 'tib',t_i_b,32768

; -----
; Initialisation
; -----

; abort - ( -- ) initialise Itsy then jump to interpret
primitive 'abort',abort
movax,word[val_number_t_i_b] ; Load AX with the value contained
; in the data field of #tib (which
; was pre-defined above as 0).
movword[val_to_in],ax; Save the same number to >in.
xorbp,bp; Clear the bp register, which is going
; to be used as the return stack
; pointer. Since it'll first be
; decremented when a value is pushed
; onto it, this means that the first
; value pushed onto the return stack
; will be stored at 0FFFEh and 0FFFFh,
; the very end of memory space, and
; the stack will grow downward from
; there.
movword[val_state],bp; Clear the value of state.
movsp,stack0 ; Set the stack pointer to the value
; defined above.
movsi,xt_interpret+2; Initialize Itsy's instruction pointer
; to the outer interpreter loop.
jmp next ; Jump to the inner interpreter and
; actually start running Itsy.

```

```

; -----
; Compilation
; -----

; , - ( x -- ) compile x to the current definition.
;   Stores the number on the stack to the memory location currently
;   pointed to by dp.
;   primitive ',',comma
movdi,word[val_dp] ; Put the value of dp into the DI register.
xchgax,bx; Move the top of the stack into AX.
stosw; Store the 16-bit value in AX directly
; into the address pointed to by DI, and
; automatically increment DI in the
; process.
movword[val_dp],di; Store the incremented value in DI as the
; new value for the dictionary pointer.
popbx; Pop the new stack top into its proper place.
jmp next          ; Go do the next word.

; lit - ( -- ) push the value in the cell straight after lit.
;   lit is the word that is compiled into a definition when you put a
;   "literal" number in a Forth definition. When your word is compiled,
;   the CFA of lit gets stored in the definition followed immediately
;   by the value of the number you put into the code. At run time, lit
;   pushes the value of your number onto the stack.
;   primitive 'lit',lit
pushbx; Push the value in BX to the stack, so that now it'll
; be 2nd from the top on the stack. The old value is
; still in BX, though. Now we need to get the new
; value into BX.
lodsw; Load into the AX register the 16-bit value pointed
; to by the SI register (Itsy's instruction pointer,
; which this op then automatically increments SI by 2).
; The net result is that we just loaded into AX the
; 16-bit data immediately following the call to lit,
; which'll be the data that lit is supposed to load.
xchgax,bx; Now swap the contents of the AX and BX registers.
; lit's data is now in BX, the top of the stack, where
; we want it. Slick, eh?
jmp next          ; Go do the next word.

; -----
; Stack
; -----

; rot - ( x y z -- y z x ) rotate x, y and z.
;   Standard Forth word that extracts number 3rd from the top of the stack
;   and puts it on the top, effectively rotating the top 3 values.
;   primitive 'rot',rote

```

```

popdx; Unload "y" from the stack.
popax; Unload "x" from the stack. Remember that "z" is
; already in BX.
pushdx; Push "y" back onto the stack.
pushbx; Push "z" down into the stack on top of "y".
xchgax,bx; Swap "x" into the BX register so that it's now
; at the top of the stack.
jmp next      ; Go do the next word.

; drop - ( x -- ) remove x from the stack.
      primitive 'drop',drop
popbx; Pop the 2nd item on the stack into the BX register,
; writing over the item that was already at the top
; of the stack in BX. It's that simple.
jmp next      ; Go do the next word.

; dup - ( x -- x x ) add a copy of x to the stack
      primitive 'dup',dupe
pushbx; Remember that BX is the top of the stack. Push an
; extra copy of what's in BX onto the stack.
jmp next      ; Go do the next word.

; # swap - ( x y -- y x ) exchange x and y
      primitive 'swap',swap
popax; Pop "x", the number 2nd from the top, into AX.
pushbx; Push "y", the former top of the stack.
xchgax,bx; Swap "x" into BX to become the new stack top. We
; don't care what happens to the value of "y" that
; ends up in AX because that value is now safely
; in the stack.
jmp next      ; Go do the next word.

; -----
; Maths / Logic
; -----

; + - ( x y -- z ) calculate z=x+y then return z
      primitive '+',plus
popax; Pop the value of "x" off of the stack.
addbx,ax; Add "x" to the value of "y" that's at the top of the
; stack in the BX register. The way the opcode is
; written, the result is left in the BX register,
; conveniently at the top of the stack.
jmp next      ; Go do the next word.

; = - ( x y -- flag ) return true if x=y
      primitive '=',equals
popax; Get the "x" value into a register.
subbx,ax; Perform BX-AX (or y-x) and leave result in BX. If x and

```



```

; y are equal, this will result in a 0 in BX. But a zero
; is a false flag in just about all Forth systems, and we
; want a TRUE flag if the numbers are equal. So...
subbx,1; Subtract 1 from it. If we had a zero before, now we've
; got a -1 (or 0ffffh), and a carry flag was generated.
; Any other value in BX will not generate a carry.
sbbbx,bx; This has the effect of moving the carry bit into the BX
; register. So, if the numbers were not equal, then the
; "sub bx,1" didn't generate a carry, so the result will
; be a 0 in the BX (numbers were not equal, result is
; false). If the original numbers on the stack were equal,
; though, then the carry bit was set and then copied
; into the BX register to act as our true flag.
; This may seem a bit cryptic, but it produces smaller
; code and runs faster than a bunch of conditional jumps
; and immediate loads would.
jmp next    ; Go do the next word.

; -----
; Peek and Poke
; -----

; @ - ( addr -- x ) read x from addr
; "Fetch", as the name of this word is pronounced, reads a 16-bit number from
; a given memory address, the way the Basic "peek" command does, and leaves
; it at the top of the stack.
      primitive '@',fetch
movbx,word[bx] ; Read the value in the memory address pointed to by
; the BX register and move that value directly into
; BX, replacing the address at the top of the stack.
jmp next      ; Go do the next word.

; ! - ( x addr -- ) store x at addr
; Similar to @, ! ("store") writes a value directly to a memory address, like
; the Basic "poke" command.
      primitive '!',store
popword[bx] ; Okay, this is a bit slick. All in one opcode, we pop
; the number that's 2nd from the top of the stack
; (i.e. "x" in the argument list) and send it directly
; to the memory address pointed to by BX (the address
; at the top of the stack).
popbx; Pop whatever was 3rd from the top of the stack into
; the BX register to become the new TOS.
jmp next      ; Go do the next word.

; -----
; Inner Interpreter
; -----

```

```

; This routine is the very heart of the Forth system. After execution, all
; Forth words jump to this routine, which pulls up the code field address
; of the next word to be executed and then executes it. Note that next
; doesn't have a header of its own.
next    lodsw; Load into the AX register the 16-bit value pointed
; to by the SI register (Itsy's instruction pointer,
; which this op then automatically increments SI by 2).
; The net result is that we just loaded into AX the
; CFA of the next word to be executed and left the
; instruction pointer pointing to the word that
; follows the next one.
xchgdi,ax; Move the CFA of the next word into the DI register.
; We have to do this because the 8086 doesn't have
; an opcode for "jmp [ax]".
jmpword[di] ; Jump and start executing code at the address pointed
; to by the value in the DI register.

; -----
; Flow Control
; -----

; 0branch - ( x -- ) jump if x is zero
; This is the primitive word that's compiled as the runtime code in
; an IF...THEN statement. The number compiled into the word's definition
; immediately after 0branch is the address of the word in the definition
; that we're branching to. That address gets loaded into the instruction
; pointer. In essence, this word sees a false flag (i.e. a zero) and
; then jumps over the words that comprise the "do this if true" clause
; of an IF...ELSE...THEN statement.
    primitive '0branch',zero_branch
lodsw; Load into the AX register the 16-bit value pointed
; to by the SI register (Itsy's instruction pointer,
; which this op then automatically increments SI by 2).
; The net result is that we just loaded into AX the
; CFA of the next word to be executed and left the
; instruction pointer pointing to the word that
; follows the next one.
testbx,bx; See if there's a 0 at the top of the stack.
jne zerob_z ; If it's not zero, jump.
xchgax,si; If the flag is a zero, we want to move the CFA of
; the word we want to branch to into the Forth
; instruction pointer. If the TOS was non-zero, the
; instruction pointer is left still pointing to the CFA
; of the word that follows the branch reference.
zerob_z popbx; Throw away the flag and move everything on the stack
; up by one spot.
jmp next    ; Oh, you know what this does by now...

; branch - ( addr -- ) unconditional jump

```

```

; This is one of the pieces of runtime code that's compiled by
; BEGIN/WHILE/REPEAT, BEGIN/AGAIN, and BEGIN/UNTIL loops. As with 0branch,
; the number compiled into the dictionary immediately after the branch is
; the address of the word in the definition that we're branching to.
    primitive 'branch',branch
movsi,word[si] ; The instruction pointer has already been
; incremented to point to the address immediately
; following the branch statement, which means it's
; pointing to where our branch-to address is
; stored. This opcode takes the value pointed to
; by the SI register and loads it directly into
; the SI, which is used as Forth's instruction
; pointer.
jmp next

; execute - ( xt -- ) call the word at xt
    primitive 'execute',execute
movdi,bx; Move the jump-to address to the DI register.
popbx; Pop the next number on the stack into the TOS.
jmpword[di] ; Jump to the address pointed to by the DI register.

; exit - ( -- ) return from the current word
    primitive 'exit',exit
movsi,word[bp] ; The BP register is used as Itsy's return stack
; pointer. The value at its top is the address of
; the instruction being pointed to before the word
; currently being executed was called. This opcode
; loads that address into the SI register.
incbp; Now we have to increment BP twice to do a manual
; "pop" of the return stack pointer.
incbp;
jmp next ; We jump to next with the SI now having the address
; pointing into the word that called the one we're
; finishing up now. The result is that next will go
; back into that calling word and pick up where it
; left off earlier.

; -----
; String
; -----

; count - ( addr -- addr2 len )
; count is given the address of a counted string (like the name field of a
; word definition in Forth, with the first byte being the number of
; characters in the string and immediately followed by the characters
; themselves). It returns the length of the string and a pointer to the
; first actual character in the string.
    primitive 'count',count
incbx; Increment the address past the length byte so

```

```

; it now points to the actual string.
pushbx; Push the new address onto the stack.
movbl,byte[bx-1] ; Move the length byte into the lower half of
; the BX register.
movbh,0; Load a 0 into the upper half of the BX reg.
jmp next

; >number - ( double addr len -- double2 addr2 zero ) if successful, or
;           ( double addr len -- int      addr2 nonzero ) on error.
; Convert a string to an unsigned double-precision integer.
; addr points to a string of len characters which >number attempts to
; convert to a number using the current number base. >number returns
; the portion of the string which can't be converted, if any.
; Note that, as is standard for most Forths, >number attempts to
; convert a number into a double (most Forths also leave it as a double
; if they find a decimal point, but >number doesn't check for that) and
; that it's called with a dummy double value already on the stack.
; On return, if the top of the stack is 0, the number was successfully
; converted. If the top of the stack is non-zero, there was an error.
    primitive '>number',to_number
; Start out by loading values from the stack
; into various registers. Remember that the
; top of the stack, the string length, is
; already in bx.
popdi; Put the address into di.
popcx; Put the high word of the double value into cx
popax; and the low word of the double value into ax.
to_num1 testbx,bx; Test the length byte.
je to_numz      ; If the string's length is zero, we're done.
; Jump to end.
pushax; Push the contents of ax (low word) so we can
; use it for other things.
movax,byte[di] ; Get the next byte in the string.
cmpal,'a'; Compare it to a lower-case 'a'.
jc to_nums      ; "jc", "jump if carry", is a little cryptic.
; I think a better choice of mnemonic would be
; "jb", "jump if below", for understanding
; what's going on here. Jump if the next byte
; in the string is less than 'a'. If the chr
; is greater than or equal to 'a', then it may
; be a digit larger than 9 in a hex number.
subal,32; Subtract 32 from the character. If we're
; converting hexadecimal input, this'll have
; the effect of converting lower case to
; upper case.
to_nums cmpal,'9'+1; Compare the character to whatever character
; comes after '9'.
jc to_numg      ; If it's '9' or less, it's possibly a decimal
; digit. Jump for further testing.

```

```
cmpal,'A'; Compare the character with 'A'.
jc to_numh      ; If it's one of those punctuation marks
; between '9' and 'A', we've got an error.
; Jump to the end.
subal,7; The character is a potentially valid digit
; for a base larger than 10. Resize it so
; that 'A' becomes the digit for 11, 'B'
; signifies a 11, etc.
to_numg subal,48; Convert the digit to its corresponding
; number. This op could also have been
; written as "sub al,'0'"
movah,0; Clear the ah register. The AX reg now
; contains the numeric value of the new digit.
cmpal,byte[val_base] ; Compare the digit's value to the base.
jnc to_numh      ; If the digit's value is above or equal to
; to the base, we've got an error. Jump to end.
; (I think using "jae" would be less cryptic.)
; (NASM's documentation doesn't list jae as a
; valid opcode, but then again, it doesn't
; list jnc in its opcode list either.)
xchgax,dx; Save the digit value in AX by swapping it
; the contents of DX. (We don't care what's
; in DX; it's scratchpad.)
popax; Recall the low word of our accumulated
; double number and load it into AX.
pushdx; Save the digit value. (The DX register
; will get clobbered by the upcoming mul.)
xchgax,cx; Swap the low and high words of our double
; number. AX now holds the high word, and
; CX the low.
mulword[val_base] ; 16-bit multiply the high word by the base.
; High word of product is in DX, low in AX.
; But we don't need the high word. It's going
; to get overwritten by the next mul.
xchgax,cx; Save the product of the first mul to the CX
; register and put the low word of our double
; number back into AX.
mulword[val_base] ; 16-bit multiply the low word of our converted
; double number by the base, then add the high
addcx,dx; word of the product to the low word of the
; first mul (i.e. do the carry).
popdx; Recall the digit value, then add it in to
addax,dx; the low word of our accumulated double-
; precision total.
; NOTE: One might think, as I did at first,
; that we need to deal with the carry from
; this operation. But we just multiplied
; the number by the base, and then added a
; number that's already been checked to be
```

```

; smaller than the base. In that case, there
; will never be a carry out from this
; addition. Think about it: You multiply a
; number by 10 and get a new number whose
; lowest digit is a zero. Then you add another
; number less than 10 to it. You'll NEVER get
; a carry from adding zero and a number less
; than 10.
decbx; Decrement the length.
incdi; Inc the address pointer to the next byte
; of the string we're converting.
jmp to_num1      ; Jump back and convert any remaining
; characters in the string.
to_numz pushax; Push the low word of the accumulated total
; back onto the stack.
to_numh pushcx; Push the high word of the accumulated total
; back onto the stack.
pushdi; Push the string address back onto the stack.
; Note that the character count is still in
; BX and is therefore already at the top of
; the stack. If BX is zero at this point,
; we've successfully converted the number.
jmp next        ; Done. Return to caller.

; -----
; Terminal Input / Output
; -----

; accept - ( addr len -- len2 ) read a string from the terminal
; accept reads a string of characters from the terminal. The string
; is stored at addr and can be up to len characters long.
; accept returns the actual length of the string.
        primitive 'accept',accept
popdi; Pop the address of the string buffer into DI.
xorcx,cx; Clear the CX register.
accept1 call getchar ; Do the bios call to get a chr from the keyboard.
cmpal,8; See if it's a backspace (ASCII character 08h).
jne acceptn      ; If not, jump for more testing.
jcxz acceptb     ; "Jump if CX=0". If the user typed a backspace but
; there isn't anything in the buffer to erase, jump
; to the code that'll beep at him to let him know.
call outchar ; User typed a backspace. Go ahead and output it.
moval,' ' ; Then output a space to wipe out the character that
call outchar ; the user had just typed.
moval,8; Then output another backspace to put the cursor
call outchar ; back into position to read another character.
deccx; We just deleted a character. Now we need to decrement
decdi; both the counter and the buffer pointer.
jmp accept1      ; Then go back for another character.

```

```

acceptn cmpal,13; See if the input chr is a carriage return.
je acceptz ; If so, we're done. jump to the end of the routine.
cmpcx,bx; Compare current string length to the maximum allowed.
jne accepts ; If the string's not too long, jump.
acceptb moval,7; User's input is unusable in some way. Send the
call outchar ; BEL chr to make a beep sound to let him know.
jmp acceptl ; Then go back and let him try again.
accepts stosb; Save the input character into the buffer. Note that
; this opcode automatically increments the pointer
; in the DI register.
inccx; But we have to increment the length counter manually.
call outchar ; Echo the input character back to the display.
jmp acceptl ; Go back for another character.
acceptz jcxz acceptb ; If the buffer is empty, beep at the user and go
; back for more input.
moval,13; Send a carriage return to the display...
call outchar ;
moval,10; ...followed by a linefeed.
call outchar ;
movbx,cx; Move the count to the top of the stack.
jmp next ;

; word - ( char -- addr ) parse the next word in the input buffer
; word scans the "terminal input buffer" (whose address is given by the
; system constant tib) for words to execute, starting at the current
; address stored in the input buffer pointer >in. The character on the
; stack when word is called is the one that the code will look for as
; the separator between words. 999 times out of 1000,; this is going to
; be a space.
    primitive 'word',word
movdi,word[val_dp] ; Load the dictionary pointer into DI.
; This is going to be the address that
; we copy the input word to. For the
; sake of tradition, let's call this
; scratchpad area the "pad".
pushdi; Save the pad pointer to the stack.
movdx,bx; Copy the word separator to DX.
movbx,word[val_t_i_b] ; Load the address of the input buffer
movcx,bx; into BX, and save a copy to CX.
addbx,word[val_to_in] ; Add the value of >in to the address
; of tib to get a pointer into the
; buffer.
addcx,word[val_number_t_i_b] ; Add the value of #tib to the address
; of tib to get a pointer to the last
; chr in the input buffer.
wordf cmpcx,bx; Compare the current buffer pointer to
; the end-of-buffer pointer.
je wordz ; If we've reached the end, jump.
moval,byte[bx] ; Get the next chr from the buffer

```

```

incbx; and increment the pointer.
cmpal,dl; See if it's the separator.
je wordf                ; If so, jump.
wordc    incdi; Increment our pad pointer. Note that
; if this is our first time through the
; routine, we're incrementing to the
; 2nd address in the pad, leaving the
; first byte of it empty.
movbyte[di],al; Write the new chr to the pad.
cmpcx,bx; Have we reached the end of the
; input buffer?
je wordz                ; If so, jump.
moval,byte[bx]          ; Get another byte from the input
incbx; buffer and increment the pointer.
cmpal,dl; Is the new chr a separator?
jne wordc                ; If not, go back for more.
wordz    movbyte[di+1],32; Write a space at the end of the text
; we've written so far to the pad.
movax,word[val_dp]       ; Load the address of the pad into AX.
xchgax,di; Swap the pad address with the pad
subax,di; pointer then subtract to get the
; length of the text in the pad.
; The result goes into AX, leaving the
; pad address in DI.
movbyte[di],al; Save the length byte into the first
; byte of the pad.
subbx,word[val_t_i_b]    ; Subtract the base address of the
; input buffer from the pointer value
; to get the new value of >in...
movword[val_to_in],bx; ...then save it to its variable.
popbx; Pop the value of the pad address
; that we saved earlier back out to
; the top of the stack as our return
; value.
jmp next

; emit - ( char -- ) display char on the terminal
primitive 'emit',emit
xchgax,bx; Move our output character to the AX register.
call outchar ; Send it to the display.
popbx; Pop the argument off the stack.
jmp next

getchar movah,7; This headerless routine does an MS-DOS Int 21h call,
int021h; reading a character from the standard input device into
movah,0; the AL register. We start out by putting a 7 into AH to
ret; identify the function we want to perform. The character
; gets returned in AL, and then we manually clear out
; AH so that we can have a 16-bit result in AX.

```



```

outchar xchgax,dx; This headerless routine does an MS-DOS Int 21h call,
movah,2; sending a character in the DL register to the standard
int021h; output device. The 2 in the AH register identifies what
ret; function we want to perform.

; -----
; Dictionary Search
; -----

; find - ( addr -- addr2 flag ) look up word in the dictionary
; find looks in the Forth dictionary for a word with the name given in the
; counted string at addr. One of the following will be returned:
;   flag =  0, addr2 = counted string --> word was not found
;   flag =  1, addr2 = call address  --> word is immediate
;   flag = -1, addr2 = call address  --> word is not immediate
primitive 'find',find
movdi,val_last    ; Get the address of the link field of the last
; word in the dictionary. Put it in DI.
findl  pushdi; Save the link field pointer.
pushbx; Save the address of the name we're looking for.
movcl,byte[bx]    ; Copy the length of the string into CL
movch,0; Clear CH to make a 16 bit counter.
inccx; Increment the counter.
findc  moval,byte[di+2] ; Get the length byte of whatever word in the
; dictionary we're currently looking at.
andal,07Fh; Mask off the immediate bit.
cmpal,byte[bx]    ; Compare it with the length of the string.
je findm          ; If they're the same, jump.
popbx; Nope, can't be the same if the lengths are
popdi; different. Pop the saved values back to regs.
movdi,word[di]    ; Get the next link address.
testdi,di; See if it's zero. If it's not, then we've not
jne findl        ; hit the end of the dictionary yet. Then jump
; back and check the next word in the dictionary.
findnf pushbx; End of dictionary. Word wasn't found. Push the
; string address to the stack.
xorbx,bx; Clear the BX register (make a "false" flag).
jmp next        ; Return to caller.
findm  incdi; The lengths match, but do the chrs? Increment
; the link field pointer. (That may sound weird,
; especially on the first time through this loop.
; But remember that, earlier in the loop, we
; loaded the length byte out the dictionary by an
; indirect reference to DI+2. We'll do that again
; in a moment, so what in effect we're actually
; doing here is incrementing what's now going to
; be treated as a string pointer for the name in
; the dictionary as we compare the characters

```

```

; in the strings.)
incbx; Increment the pointer to the string we're
; checking.
loop findc      ; Decrements the counter in CX and, if it's not
; zero yet, loops back. The same code that started
; out comparing the length bytes will go through
; and compare the characters in the string with
; the chrs in the dictionary name we're pointing
; at.
popbx; If we got here, then the strings match. The
; word is in the dictionary. Pop the string's
; starting address and throw it away. We don't
; need it now that we know we're looking at a
; defined word.
popdi; Restore the link field address for the dictionary
; word whose name we just looked at.
movbx,1; Put a 1 at the top of the stack.
incdi; Increment the pointer past the link field to the
incdi; name field.
moval,byte[di]  ; Get the length of the word's name.
testal,080h; See if it's an immediate.
jne findi      ; "test" basically performs an AND without
; actually changing the register. If the
; immediate bit is set, we'll have a non-zero
; result and we'll skip the next instruction,
; leaving a 1 in BX to represent that we found
; an immediate word.
negbx; But if it's not an immediate word, we fall
; through and generate a -1 instead to get the
; flag for a non-immediate word.
findi andax,31; Mask off all but the valid part of the name's
; length byte.
adddi,ax; Add the length to the name field address then
incdi; add 1 to get the address of the code field.
pushdi; Push the CFA onto the stack.
jmp next      ; We're done.

; -----
; Colon Definition
; -----

; : - ( -- ) define a new Forth word, taking the name from the input buffer.
; Ah! We've finally found a word that's actually defined as a Forth colon
; definition rather than an assembly language routine! Partly, anyway; the
; first part is Forth code, but the end is the assembly language run-time
; routine that, incidentally, executes Forth colon definitions. Notice that
; the first part is not a sequence of opcodes, but rather is a list of
; code field addresses for the words used in the definition. In each code
; field of each defined word is an "execution tag", or "xt", a pointer to

```

```

; the runtime code that executes the word. In a Forth colon definition, this
; is going to be a pointer to the docolon routine we see in the second part
; of the definition of colon itself below.
colon ':',colon
dw xt_lit,-1; If you write a Forth routine where you put an
; integer number right in the code, such as the
; 2 in the phrase, "dp @ 2 +", lit is the name
; of the routine that's called at runtime to put
; that integer on the stack. Here, lit pushes
; the -1 stored immediately after it onto the
; stack.
dw xt_state      ; The runtime code for a variable leaves its
; address on the stack. The address of state,
; in this case.
dw xt_store      ; Store that -1 into state to tell the system
; that we're switching from interpret mode into
; compile mode. Other than creating the header,
; colon doesn't actually compile the words into
; the new word. That task is performed in
; interpret, but it needs this new value stored
; into state to tell it to do so.
dw xt_create      ; Now we call the word that's going to create the
; header for the new colon definition we're going
; to compile.
dw xt_do_semi_code ; Write, into the code field of the header we just
; created, the address that immediately follows
; this statement: the address of the docolon
; routine, which is the code that's responsible
; for executing the colon definition we're
; creating.
docolon decbp; Here's the runtime code for colon words.
decbp; Basically, what docolon does is similar to
; calling a subroutine, in that we have to push
; the return address to the stack. Since the 80x86
; doesn't directly support more than one stack and
; the "real" stack is used for data, we have to
; operate the Forth virtual machine's return stack
; manually. So, first, we manually decrement the
; return stack pointer twice to point to where
; we're going to save the return address.
movword[bp],si; Then we write that address directly from the
; instruction pointer to that location.
leasi,[di+2]      ; We now have to tell Forth to start running the
; words in the colon definition we just started.
; The value in DI was left pointing at the code
; field of the word that we just started that just
; jumped into docolon. By loading into the
; instruction pointer the value that's 2 bytes
; later, at the start of the data field, we're

```

```

; loading into the IP the address of the first
; word in that definition. Execution of the other
; words in that definition will occur in sequence
; from here on.
jmp next          ; Now that we're pointing to the correct
; instruction, go do it.

; ; - ( -- ) complete the Forth word being compiled
;         colon ';',semicolon,immediate
; Note above that ; is immediate, the first such
; word we've seen here. It needs to be so because
; it's used only during the compilation of a colon
; definition and we want it to execute rather than
; just being stored in the definition.
dw xt_lit,xt_exit ; Put the address of the code field of exit onto
; the stack.
dw xt_comma       ; Store it into the dictionary.
dw xt_lit,0; Now put a zero on the stack...
dw xt_state       ; along with the address of the state variable.
dw xt_store       ; Store the 0 into state to indicate that we're
; done compiling a word and are now back into
; interpret mode.
dw xt_exit        ; exit is the routine that finishes up the
; execution of a colon definition and jumps to
; next in order to start execution of the next
; word.

; -----
; Headers
; -----

; create - ( -- ) build a header for a new word in the dictionary, taking
; the name from the input buffer
;         colon 'create',create
dw xt_dp,xt_fetch ; Get the current dictionary pointer.
dw xt_last,xt_fetch ; Get the LFA of the last word in the dictionary.
dw xt_comma       ; Save the value of last at the current point in
; the dictionary to become the link field for
; the header we're creating. Remember that comma
; automatically increments the value of dp.
dw xt_last,xt_store ; Save the address of the link field we just
; created as the new value of last.
dw xt_lit,32; Parse the input buffer for the name of the
dw xt_word        ; word we're creating, using a space for the
; separation character when we invoke word.
; Remember that word copies the parsed name
; as a counted string to the location pointed
; to by dp, which not coincidentally is
; exactly what and where we need it for the

```

```

; header we're creating.
dw xt_count      ; Get the address of the first character of the
; word's name, and the name's length.
dw xt_plus       ; Add the length to the address to get the addr
; of the first byte after the name, then store
dw xt_dp,xt_store ; that address as the new value of dp.
dw xt_lit,0; Put a 0 on the stack, and store it as a dummy
dw xt_comma      ; placeholder in the new header's CFA.
dw xt_do_semi_code ; Write, into the code field of the header we just
; created, the address that immediately follows
; this statement: the address of the dovar
; routine, which is the code that's responsible
; for pushing onto the stack the data field
; address of the word whose header we just
; created when it's executed.
dovar pushbx; Push the stack to make room for the new value
; we're about to put on top.
leabx,[di+2]      ; This opcode loads into bx whatever two plus the
; value of the contents of DI might be, as opposed
; to a "mov bx,[di+2]", which would move into BX
; the value stored in memory at that location.
; What we're actually doing here is calculating
; the address of the data field that follows
; this header so we can leave it on the stack.
jmp next          ;

; # (;code) - ( -- ) replace the xt of the word being defined with a pointer
; to the code immediately following (;code)
; The idea behind this compiler word is that you may have a word that does
; various compiling/accounting tasks that are defined in terms of Forth code
; when its being used to compile another word, but afterward, when the new
; word is executed in interpreter mode, you want your compiling word to do
; something else that needs to be coded in assembly. (;code) is the word that
; says, "Okay, that's what you do when you're compiling, but THIS is what
; you're going to do while executing, so look sharp, it's in assembly!"
; Somewhat like the word DOES>, which is used in a similar manner to define
; run-time code in terms of Forth words.
    primitive ' (;code) ',do_semi_code
movdi,word[val_last] ; Get the LFA of the last word in dictionary
; (i.e. the word we're currently in the middle
; of compiling) and put it in DI.
moval,byte[di+2]      ; Get the length byte from the name field.
andax,31; Mask off the immediate bit and leave only
; the 5-bit integer length.
adddi,ax; Add the length to the pointer. If we add 3
; to the value in DI at this point, we'll
; have a pointer to the code field.
movword[di+3],si; Store the current value of the instruction
; pointer into the code field. That value is

```

```

; going to point to whatever follows (;code) in
; the word being compiled, which in the case
; of (;code) had better be assembly code.
movsi,word[bp]      ; Okay, we just did something funky with the
; instruction pointer; now we have to fix it.
; Directly load into the instruction pointer
; the value that's currently at the top of
; the return stack.
incbp; Then manually increment the return stack
incbp; pointer.
jmp next            ; Done. Go do another word.

; -----
; Constants
; -----

; constant - ( x -- ) create a new constant with the value x, taking the name
; from the input buffer
colon 'constant',constant
dw xt_create        ; Create the constant's header.
dw xt_comma          ; Store the constant's value into the word's
; data field.
dw xt_do_semi_code ; Write, into the code field of the header we just
; created, the address that immediately follows
; this statement: the address of the doconst
; routine, which is the code that's responsible
; for pushing onto the stack the value that's
; contained in the data field of the word whose
; header we just created when that word is
; invoked.
doconst pushbx; Push the stack down.
movbx,word[di+2] ; DI should be pointing to the constant's code
; field. Load into the top of the stack the
; value 2 bytes further down from the code field,
; i.e. the constant's actual value.
jmp next            ;

; -----
; Outer Interpreter
; -----

; -----
; NOTE! The following line with the final: label MUST be
; immediately before the final word definition!
; -----

final:

```

```

colon 'interpret',interpret
interp dw xt_number_t_i_b ; Get the number of characters in the input
dw xt_fetch ; buffer.
dw xt_to_in ; Get the index into the input buffer.
dw xt_fetch ;
dw xt_equals ; See if they're the same.
dw xt_zero_branch ; If not, it means there's still some text in
dw intpar ; the buffer. Go process it.
dw xt_t_i_b ; if #tib = >in, we're out of text and need to
dw xt_lit ; read some more. Put a 50 on the stack to tell
dw50; accept to read up to 50 more characters.
dw xt_accept ; Go get more input.
dw xt_number_t_i_b ; Store into #tib the actual number of characters
dw xt_store ; that accept read.
dw xt_lit ; Reposition >in to index the 0th byte in the
dw0; input buffer.
dw xt_to_in ;
dw xt_store ;
intpar dw xt_lit ; Put a 32 on the stack to represent an ASCII
dw32; space character. Then tell word to scan the
dw xt_word ; buffer looking for that character.
dw xt_find ; Once word has parsed out a string, have find
; see if that string matches the name of any
; words already defined in the dictionary.
dw xt_dupe ; Copy the flag returned by find, then jump if
dw xt_zero_branch ; it's a zero, meaning that the string doesn't
dw intnf ; match any defined word names.
dw xt_state ; We've got a word match. Are we interpreting or
dw xt_fetch ; do we want to compile it? See if find's flag
dw xt_equals ; matches the current value of state.
dw xt_zero_branch ; If so, we've got an immediate. Jump.
dw intexc ;
dw xt_comma ; Not immediate. Store the word's CFA in the
dw xt_branch ; dictionary then jump to the end of the loop.
dw intdone ;
intexc dw xt_execute ; We found an immediate word. Execute it then
dw xt_branch ; jump to the end of the loop.
dw intdone ;
intnf dw xt_dupe ; Okay, it's not a word. Is it a number? Copy
; the flag, which we've already proved is 0,
; thereby creating a double-precision value of
; 0 at the top of the stack. We'll need this
; shortly when we call >number.
dw xt_rote ; Rotate the string's address to the top of
; the stack. Note that it's still a counted
; string.
dw xt_count ; Use count to split the string's length byte
; apart from its text.
dw xt_to_number ; See if we can convert the text into a number.

```

```

dw xt_zero_branch ; If we get a 0 from 0branch, we got a good
dw intskip        ; conversion. Jump and continue.
dw xt_state       ; We had a conversion error. Find out whether
dw xt_fetch       ; we're interpreting or compiling.
dw xt_zero_branch ; If state=0, we're interpreting. Jump
dw intnc          ; further down.
dw xt_last        ; We're compiling. Shut the compiler down in an
dw xt_fetch       ; orderly manner. Get the LFA of the word we
dw xt_dupe        ; were trying to compile. Set aside a copy of it,
dw xt_fetch       ; then retrieve from it the LFA of the old "last
dw xt_last        ; word" and resave that as the current last word.
dw xt_store       ;
dw xt_dp          ; Now we have to save the LFA of the word we just
dw xt_store       ; tried to compile back into the dictionary
; pointer.
intnc dw xt_abort ; Whether we were compiling or interpreting,
; either way we end up here if we had an
; unsuccessful number conversion. Call abort
; and reset the system.
intskip dw xt_drop ; >number was successful! Drop the address and
dw xt_drop         ; the high word of the double-precision numeric
; value it returned. We don't need either. What's
; left on the stack is the single-precision
; number we just converted.
dw xt_state        ; Are we compiling or interpreting?
dw xt_fetch        ;
dw xt_zero_branch  ; If we're interpreting, jump on down.
dw intdone         ;
dw xt_lit          ; No, John didn't stutter here. These 4 lines are
dw xt_lit          ; how '[' lit , , " get encoded. We need to store
dw xt_comma        ; lit's own CFA into the word, followed by the
dw xt_comma        ; number we just converted from text input.
intdone dw xt_branch ; Jump back to the beginning of the interpreter
dw interpret       ; loop and process more input.

```

freemem:

```

; That's it! So, there you have it! Only 33 named Forth words...
;
; , @ >in dup base word abort 0branch interpret
; + ! lit swap last find create constant (;code)
; = ; tib drop emit state accept >number
; : dp rot #tib exit count execute
;
; ...plus 6 pieces of headerless code and run-time routines...
;
; getchar outchar docolon dovar doconst next
;
; ...are all that's required to produce a functional Forth interpreter

```



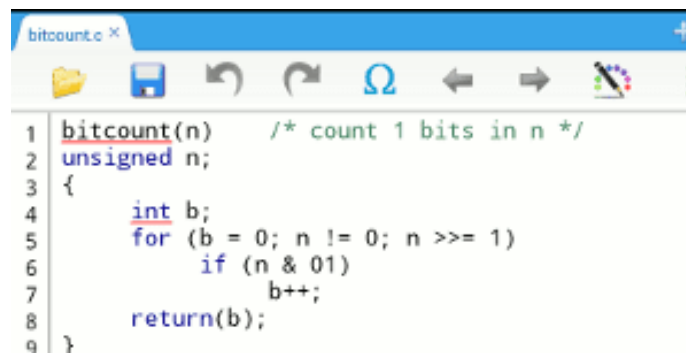
```
; capable of compiling colon definitions, only 978 bytes long! Granted,
; it's lacking a number of key critical words that make it nigh unto
; impossible to do anything useful, but this just goes to show just
; how small a functioning Forth system can be made.
```

### 🚀 09/18/12--15:10: [Programming Editors for Android on the Nexus 7](#)

0  0   

A few days ago I received a Nexus 7 for programming on the go. The Nexus features a 1.3 GHz quad-core CPU, 1 GBRAM and runs Android 4.1 (Jelly Bean).

The Nexus is supplied without a text editor so I started searching for the perfect programming editor. After experimenting with dozens, here are the top candidates:



The [920 Text Editor](#) supports multiple open files and syntax highlighting for 35 languages.

(click thumbnail for the full screenshot)

[DroidEdit](#) is ad-supported and features syntax highlighting for 26 languages and multiple open files. Upgrading to DroidEdit Pro adds support for FTP and Dropbox.

```

1 bitcount(n) /* count 1 bits in n */
2 unsigned n;
3
4 int b;
5 for (b = 0; n != 0; n >>= 1)
6     if (n & 01)
7         b++;
8     return(b);

```

```

1 bitcount(n) /* count 1 bits in n */
2 unsigned n;
3 {
4     int b;
5     for (b = 0; n != 0; n >>= 1)
6         if (n & 01)
7             b++;
8     return(b);
9 }

```

[Jota](#) is a freeware editor with syntax highlighting for 20+ languages, regular expression search and the ability to share directly to Twitter, Facebook, etc.

```

1 bitcount(n) /* count 1 bits in n */
2 unsigned n;
3 {
4     int b;
5     for (b = 0; n != 0; n >>= 1)
6         if (n & 01)
7             b++;
8     return(b);
9 }

```

[Jota+](#) allows two open files and has syntax highlighting for 20+ file formats. Jota+ Pro adds support for multiple open files, Dropbox, Box and SkyDrive.

[Touchqode](#) supports highlighting for 8 languages, has built in FTP access and a custom keyboard. Upgrading to Touchqode Pro adds several features including a GitHub viewer.

```
bitcount(n)    /* count 1 bits in n */
unsigned n;
{
    int b;
    for (b = 0; n != 0; n >>= 1)
        if (n & 01)
            b++;
    return(b);
}
```

Unfortunately none of the above supports code folding which would be really handy on the 7" screen. In the end I've settled for Jota+ with [CodeWhisk](#), a replacement keyboard with faster access to numbers and symbols. Which editor / keyboard combo are you using on Android? :-)

📅 **07/04/13--16:56: [Silicon Dreams & The Vintage Computer Festival](#)**

0 👍 0 👎 🔗 ☰



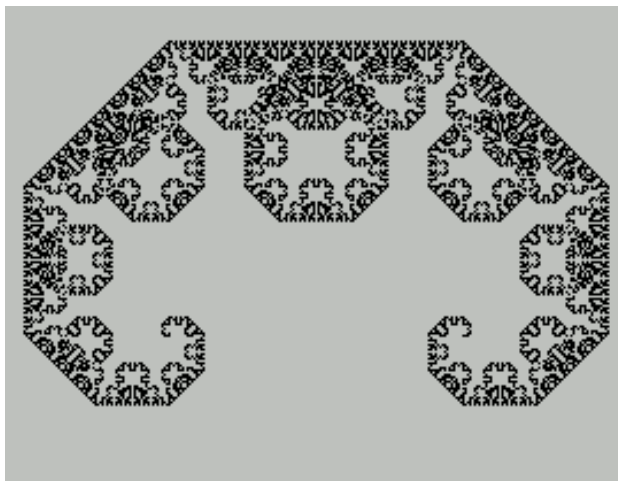
This weekend the biggest event in the U.K.'s retro computing calendar will be hosted at the [Snibston Discovery Museum](#) in Leicestershire. The [Vintage Computer Festival](#) follows

2010's highly successful event with most of the main exhibitors returning. Some of the highlights include:

- ➔ **Raspberry Jam - a hands-on workshop for the Raspberry Pi single board computer organised by the Centre for Computing History.**
- ➔ **The Amiga is well represented at the show with demonstrations of MorphOS 3.2 and the latest AmigaOne X1000.**
- ➔ **15+ exhibitors will be displaying a wide range of historic / unusual computers including RISC OS running on the Raspberry Pi.**
- ➔ **Look out for the Spectranet ethernet board in action. Tweet direct from a ZX Spectrum complete with nixie tube tweetometer!**

The event runs from 5th - 7th July. Tickets are £15 for the day, or £20 for the weekend. We'll be there when the gates open at 10am. Is anyone else planning to attend?

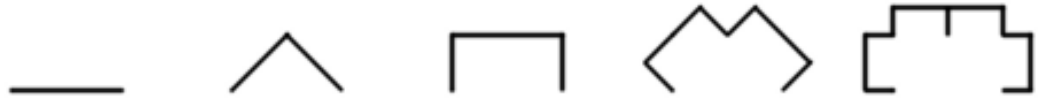
➔ **08/01/13--11:24: ZX Spectrum Koch (Lévy C) Curve** 0  0   



A few years ago I submitted a couple of type-in programs ([C-Curve](#) and [Curtains](#)) to *Your Sinclair* and they featured in the penultimate issue (August 1993).

Encouraged by a shiny new YS badge I sent off a new batch of programs. Unfortunately it was too late. The September issue would be *Your Sinclair's* "Big Final Issue".

**C-Curve** is one of the simplest fractal curves. It starts with a straight line. To find the next iteration, each line is replaced by two lines at 90°:



Here's a later 69 byte version of the program which plots the fractal in approximately 1.52 seconds! Assemble with [Pasm0](#) (pasm0 ccurve.asm ccurve.bin), load the binary to address 65467 in your favourite emulator and run using RANDOMIZE USR 65467 :-)

```
org 65467
ld de,49023 ; d = position on x axis
              ; e = position on y axis
ld bc,3840  ; b = number of iterations
              ; c = initial direction
RECURSE:
  djnz DOWN

  ld a,6      ; check direction
  and c       ; c=0, left
  rrca        ; c=2, up
  rrca        ; c=4, right
  add a,a     ; c=6, down
  dec a
  jr nc,XMOVE

  add a,e     ; adjust y position +/-1

  ld e,a      ; calculate high byte of screen pos
  rrca
  scf
  rra
  rrca
  xor e
  and 88
  xor e
  and 95
  ld h,a
  sub h
```

```

XMOVE:
    add a,d      ; adjust x position +/-1

    ld d,a      ; calculate low byte of screen pos
    rlca
    rlca
    rlca
    xor e
    and 199
    xor e
    rlca
    rlca
    ld l,a

    ld a,7      ; calculate bit position of pixel
    and d
    ld b,a
    inc b
    ld a,1
SHIFTBIT:
    rrca
    djnz SHIFTBIT

    xor (hl)    ; plot
    ld (hl),a
    ret

DOWN:
    inc c      ; turn 45° clockwise
    call RECURSE
    inc b
    dec c      ; turn 90° anti-clockwise
    dec c
    call RECURSE
    inc b
    inc c      ; turn 45° clockwise
    ret

```

Finally here's a short type-in program to poke the code into a real Spectrum!

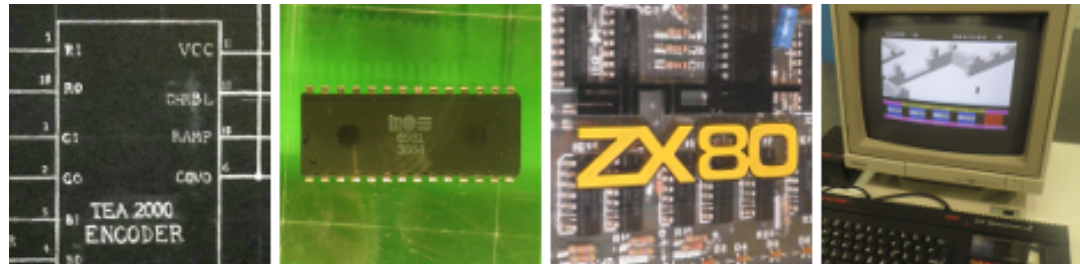
```

10 REM *Koch 69* John Metcalf
20 CLS : FOR p=65468 TO 65534
STEP 3: READ a: POKE p,a/65
536: POKE p+1,a-256*INT(a
/256): POKE p-1,a/256-256*
PEEK p: NEXT p: RANDOMIZE U
SR 65467
30 DATA 8327359,271,3215481,45
2111,8851261,929667,1007415
,990891,5760427,6219367,901
6455,460551,15051463,436999
,4091655,4628996,81423,1658
4622,292553,14945217,851725
,12637183,265161

```

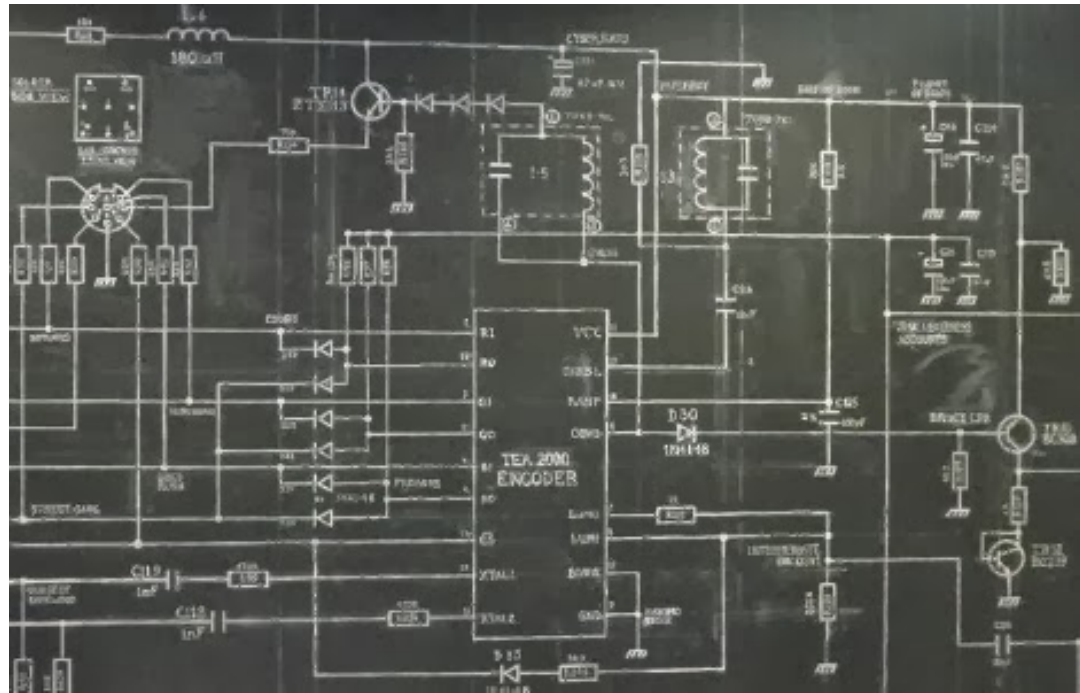
+3 BASIC

➔ **10/18/13--16:45: Video Gaming 1979-1989 at the NCCD** 0  0   



[The National Centre for Craft & Design](http://retro666.rssing.com/chan-7149388/all_p2.html) is hosting an event to celebrate the golden age of video games, "*Revolution in the Bedroom, War in the Playground: Video Gaming 1979-1989*". The exhibition runs from 19th October to 5th January in the main gallery.

The exhibition focuses on bedroom programmers, 8-bit games design and magazine cover art with classic games running on several computers. If you're in the Sleaford area, it's definitely worth a visit.



Detail from Simon Patterson's 15 metre chalkboard circuit diagram







Cover artwork by Oliver Frey



▶ **11/11/13--15:35: The Centre for Computing History in Cambridge**

0



0



[The Centre for Computing History](#) is a short walk from Cambridge city centre and is home to a sizeable collection of computers. The museum actively encourages visitors to sit down, try out a few games and even have a go at BASIC programming.

The museum's collection ranges from mechanical calculators and mainframes to home computers and games consoles. Most of the home computers and consoles are switched on and running classic games.

If you're interested in the history of computing (particularly home computing), the centre is the perfect place for a day out.



*Relaxen und watschen der Blinkenlights - the MITS Altair 8800*



`PRINT CHR$(205.5+RND(1));` - Commodore PET 2001





PLOT 48,56:DRAW 160,0,65536 - ZX Spectrum 48K



Intel MDS 80 Microprocessor Development System



HP1000 F Series minicomputer

➔ **02/09/14--14:46: The Spring 2014 Core War Tournament**

0



0



In May 1984 A K Dewdney introduced Core War, a game played between two or more computer programs in the memory of a virtual computer. The aim of the game is to disable all opponents and survive the longest. A variety of strategies have evolved for Core War, each with their own strengths and weaknesses.

To celebrate the 30th anniversary in May, *The Spring Core War Tournament* will be held at *The Centre for Computing History* in Cambridge UK. The Centre was established to tell the story of the Information Age and presents an interactive collection of computers and artifacts.



Entries can be up to 25 instructions and will compete in three different core sizes, 800 (tiny), 8000 (standard) and 55440 (large). A program's final score will be calculated as follows:

$$\text{final\_score} = 2 \times \text{standard\_score} + \text{tiny\_score} + \text{large\_score}$$

The program with the highest final score will be awarded the first prize, \$50 and a signed copy of *The Armchair Universe* by A K Dewdney. The top program in each core size will be awarded a signed copy of *Life As It Could Be* by Thure Etzold, a technothriller which explores the possibility of programs escaping the confines of the Core War virtual computer.

Entries can be sent via email or delivered to The Centre on the day of the tournament (date tbc). Players can submit up to two entries. All entries will be published at the end of the tournament.

The provisional deadline is 01 May 2014. Updates will be posted on [news:rec.games.corewar](mailto:news:rec.games.corewar), <http://corewar.eu>, [#corewars](https://twitter.com/corewars) on [irc.freenode.net](https://www.freenode.net) and on twitter using the hashtag [#corewar](https://twitter.com/corewar). Good Luck!

## Technical Details:



Players may enter up to two programs. Programs face each other in a one-on-one round robin, no p-space, no self-fights, no read/write limits. Entries must be your own work. Extended ICWS'94 Draft Redcode applies with the following settings:

➔ `pmars -s 800 -p 800 -c 8000 -l 25 -d 25`

➔ `pmars -s 8000 -p 8000 -c 80000 -l 25 -d 100`

➔ `pmars -s 55440 -p 10000 -c 500000 -l 25 -d 200`

Entries may use the run-time variables (CORESIZE, MAXPROCESSES, etc) to tailor the program for each core size, but the program must still behave essentially the same. Some allowed examples include:

➔ **tweaking the steps / constants**

➔ **adding an extra bombing line to the core clear**

➔ **including an extra SPL/MOV pair to a paper**

Completely changing the program's behaviour or swapping / adding extra components for each core size is not allowed.

## Further Details:

More information about Core War can be found at:

➔ <http://corewar.co.uk>

➔ <http://www.corewar.info>

➔ <http://users.obs.carnegiescience.edu/birk/COREWAR/>

Software is available from:

➔ <http://corewar.co.uk/pmars>

➔ <http://corewar.co.uk/wendell>

➔ <http://harald.ist.org/ares>

Core War can be played online at:

➔ <http://koth.org>

➔ <http://sal.discontinuity.info>

For help, advice and updates see:

➔ <news:rec.games.corewar>

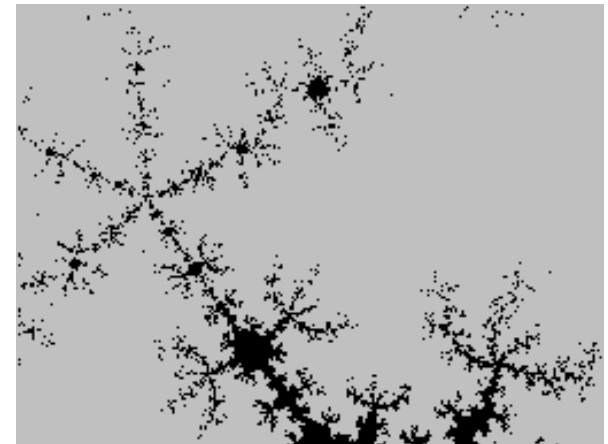
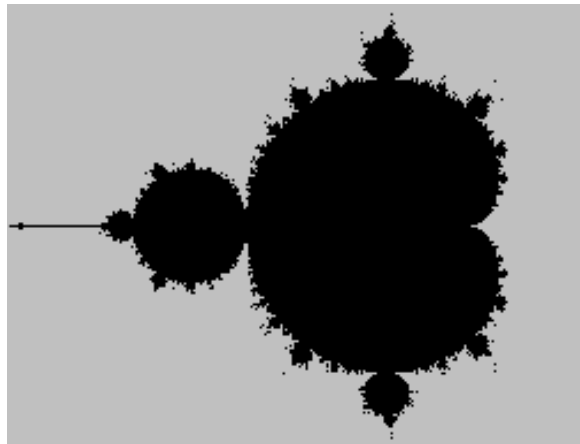
➔ <http://corewar.eu>

➔ <irc://irc.freenode.net/#COREWARS>

The Centre for Computing History has a website at:

➔ <http://computinghistory.org.uk>

➔ **03/15/14--17:05: [Plotting the Mandelbrot Set on the ZX Spectrum](#)**



**The Mandelbrot set** is a fractal which iterates the equation  $z_{n+1} = z_n^2 + c$  in the complex plane and plots which points tend to infinity. Plotting the set with Sinclair BASIC takes over 24 hours so I was curious how much faster it would be in assembly.

It turns out if we use fast 16-bit fixed-point arithmetic we can plot the Mandelbrot in about 5 minutes. To minimise multiplications each iteration is calculated as:

$$r_{n+1} = (r_n + i_n) \times (r_n - i_n) + x$$

$$i_{n+1} = 2 \times i_n \times r_n + y$$

The following test is used to detect points which tend to infinity:

$$|i_n| + |r_n| \geq 2 \times \sqrt{2}.$$

```

org 60000
ld de,255*256+191
XLOOP:
push de
ld hl,-180    ; x-coordinate
ld e,d
call SCALE
ld (XPOS),bc
pop de
YLOOP:
push de
ld hl,-96     ; y-coordinate
call SCALE
ld (YPOS),bc
ld hl,0
ld (IMAG),hl
ld (REAL),hl
ld b,15       ; iterations
ITER:
push bc
ld bc,(IMAG)
ld hl,(REAL)
or a
sbc hl,bc
ld d,h
ld e,l
add hl,bc
add hl,bc

```

```

call FIXMUL
ld de,(XPOS)
add hl,de
ld de,(REAL)
ld (REAL),hl
ld hl,(IMAG)
call FIXMUL
rla
adc hl,hl
ld de,(YPOS)
add hl,de
ld (IMAG),hl
call ABSVAL
ex de,hl
ld hl,(REAL)
call ABSVAL
add hl,de
ld a,h
cp 46          ; 46 ≅ 2 × √ 2 << 4
pop bc
jr nc,ESCAPE
djnz ITER
pop de
call PLOT
db 254          ; trick to skip next instruction
ESCAPE:
pop de
dec e
jr nz,YLOOP
dec d
jr nz,XLOOP
ret

FIXMUL:          ; hl = hl × de >> 24
call MULT16BY16
ld a,b
ld b,4
FMSHIFT:
rla
adc hl,hl
djnz FMSHIFT
ret

```

```
SCALE:          ; bc = (hl + e) × zoom
    ld d,0
    add hl,de
    ld de,48     ; zoom

MULT16BY16:     ; hl:bc (signed 32 bit) = hl × de
    xor a
    call ABSVAL
    ex de,hl
    call ABSVAL
    push af
    ld c,h
    ld a,l
    call MULT8BY16
    ld b,a
    ld a,c
    ld c,h
    push bc
    ld c,l
    call MULT8BY16
    pop de
    add hl,de
    adc a,b
    ld b,l
    ld l,h
    ld h,a
    pop af
    rra
    ret nc
    ex de,hl
    xor a
    ld h,a
    ld l,a
    sbc hl,bc
    ld b,h
    ld c,l
    ld h,a
    ld l,a
    sbc hl,de
    ret
```

```
MULT8BY16:      ; returns a:h1 (24 bit) = a x de
                ld hl,0
                ld b,8
M816LOOP:
                add hl,hl
                rla
                jr nc,M816SKIP
                add hl,de
                adc a,0
M816SKIP:
                djnz M816LOOP
                ret

PLOT:           ; plot d = x-axis, e = y-axis
                ld a,7
                and d
                ld b,a
                inc b
                ld a,e
                rra
                scf
                rra
                or a
                rra
                ld l,a
                xor e
                and 248
                xor e
                ld h,a
                ld a,d
                xor 1
                and 7
                xor d
                rrca
                rrca
                rrca
                ld l,a
                ld a,1
PLOTBIT:
                rrca
                djnz PLOTBIT
                or (hl)
```

```

ld (hl),a
ret

ABSVAL:      ; returns hl = |hl| and increments
             ; a if the sign bit changed
ret z
ld b,h
ld c,l
ld hl,0
or a
sbc hl,bc
inc a
ret

XPOS:dw 0
YPOS:dw 0
REAL:dw 0
IMAG:dw 0

```

🚩 **01/03/14--06:00: Fast Z80 Bit Reversal**

0 👍 0 👎  

For years I've been using the following simple code to reverse the bits in the A register by rotating the bits left out of one register and right into another:

```

; reverse bits in A
; 8 bytes / 206 cycles

ld b,8
ld l,a
REVLOOP:
rl l
rra
djnz REVLOOP

```

Recently I wondered if it's possible to save a few cycles. It turns out the bits are at most 3 rotations away from their position in the reverse:

7	6	5	4	3	2	1	0
$\Leftarrow 1$	$\Leftarrow 3$	$3 \Rightarrow$	$1 \Rightarrow$	$\Leftarrow 1$	$\Leftarrow 3$	$3 \Rightarrow$	$1 \Rightarrow$
0	1	2	3	4	5	6	7

With this in mind I devised a bit-twiddling hack to reverse the bits in about a third of the time using only 6 rotates and a bit of logic to recombine the rotated bits. Here's the code, which no doubt has been done many times before:

```
; reverse bits in A
; 17 bytes / 66 cycles

ld l,a      ; a = 76543210
rlca
rlca        ; a = 54321076
xor l
and 0xAA
xor l        ; a = 56341270
ld l,a
rlca
rlca
rlca        ; a = 41270563
rrc l        ; l = 05634127
xor l
and 0x66
xor l        ; a = 01234567
```

➔ **12/08/14--13:29: Z80 Size Programming Challenge #1**

0  0   

A few days ago I issued a Z80 programming challenge for the ZX Spectrum:



Something simple for the first challenge. Write the shortest code to fill the screen with a chequerboard pattern of 1 pixel squares. No RAM/ROM other than the 6144 byte bitmap screen memory should be written to.

Target: under 25 bytes.

- ➡ **Your program shouldn't rely on the initial contents of registers.**
- ➡ **Programs must return. The `RET` instruction is included in the size.**
- ➡ **So everyone has a fair chance comment with the code size not code.**
- ➡ **There are no prizes, just the chance to show off your coding skill.**

## Final Results

Congratulations to all who entered, especially Allan Høiberg and Introspec Zx who both discovered a 15-byte solution. The final results are as follows:

Coder	Size
Allan Høiberg	15
Introspec Zx	15
Jim Bagley	16
Paul Rhodes	16
Krystian Włosek	16
Tim Webber	16
Steve Wetherill	16
John Young	16

Coder	Size
Simon Brattel	16
John Metcalf	16
Dariusz EM	17
Chris Walsh	23

## Winning Entries

Allan was the first to discover a 15-byte solution:

```
                LD BC,22272
                LD A,85
LoopB:          BIT 6,B
                RET Z
LoopC:          DEC C
                LD (BC),A
                JR NZ,LoopC
                CPL
                DJNZ loopB
```

Introspec found a 15-byte solution with only one loop:

```
filloop5:       ld hl,16384+6143
                ld a,h
                rra
                sbc a,a
                xor %01010101
                ld (hl),a
                dec hl
                bit 6,h
                jr nz,filloop5
                ret
```

My own attempts all fell short at 16 bytes:

```
fill:      ld hl,22528-256
           ld bc,24*256+170
           dec l
           ld (hl),c
           jr nz,fill
           rrc c
           dec h
           djnz fill
           ret
```

Entries are archived on [John Young's website](#). Thanks to everyone who entered or otherwise supported the challenge. :-)



➔ **12/15/14--14:09: Z80 Size Programming Challenge #2**

0 👍 0 👎 🔗 ☰

Last week I issued the second Z80 programming challenge:

Something slightly more complex this time. Write the shortest code to mirror the entire Sinclair Spectrum screen (256×192 pixels) left to right including the colours / attributes. The deadline is Monday 15th, midday (GMT).

Target: under 50 bytes.

- ➡ **Your program shouldn't rely on the initial contents of registers.**
- ➡ **No RAM/ROM other than the screen memory should be written to.**
- ➡ **Programs must return. The `RET` instruction is included in the size.**
- ➡ **So everyone has a fair chance comment with the code size not code.**
- ➡ **There are no prizes, just the chance to show off your coding skills.**

## Final Results

We stepped up the difficulty for the second challenge so congratulations to everyone who entered. Introspec ZX and Tim Webber discovered the shortest solutions. Here are the final results:

Coder	Size
Introspec Zx	34
Tim Webber	34
John Metcalf	34
Paul Rhodes	35
Simon Brattel	35
Jim Bagley	36

Coder	Size
Steve Wetherill	38
John Young	49
Chris Walsh	49
Dariusz EM	50

## Winning Entries

Introspec submitted the first 34 byte solution using a couple of neat tricks. Note the use of CP L to check which side of the screen it's working on and the byte saved by setting B to #58:

```
                                ld hl,16384+6912
screenflip:                    ld d,h
                                ld a,l
                                xor #1F
                                ld e,a
                                cp l
                                jr nc,noflip

                                ld a,(de)
                                ld c,(hl)
                                ld (hl),a
                                ld a,c
                                ld (de),a

noflip:                        ld b,#58
                                ld a,h
                                cp b
                                jr nc,skipattr

byteflip:                      rlc (hl)
                                rra
                                djnz byteflip
                                ld (hl),a
```

```
skipattr:    dec hl
             bit 6,h
             jr nz,screenflip
             ret
```

Tim Webber's solution saves a series of addresses on the stack to be used later:

```
start:       ld hl,23296
loop1:       dec hl
             bit 6, h
             ret z
             ld a, 87
             cp h
             jr c, noinv
             ld b,8
doinv:       rl (hl)
             rra
             djnz doinv
             ld (hl), a
noinv:       push hl
             bit 4,1
             jr nz, loop1
             pop de
             pop hl
             ld a,(de)
             ld c, (hl)
             ld (hl), a
             ex de, hl
             ld (hl), c
             jr loop1
```

Although I didn't enter I also found a couple of 34 byte solutions. The first mirrors two bytes in the inner loop:

```
mirror:      ld hl,16384
             ld d,h
             ld a,l
             xor 31
             ld e,a
             ld a,h
             cp 91
```

```

ret z
cp 88
ld a,(de)
ld c,a
jr nc,attrib
ld b,8
rrca
mirrorbits: rl (hl)
rra
djnz mirrorbits
db 1 ; skip the next two instructions
attrib: ld a,(hl)
ld (hl),c
ld (de),a
inc l
inc hl
jr mirror

```

My second has two separate loops. The first loop mirrors bytes, the second mirrors the screen:

```

ld hl,22527
mir: ld a,128
mirrorbits: rl (hl)
rra
jr nc,mirrorbits
ld (hl),a
dec hl
bit 6,h
jr nz,mir
mirror: inc hl
ld d,h
ld a,l
xor 31
ld e,a
ld a,h
cp 91
ret z
ld a,(de)
ld c,a
ld a,(hl)
ld (hl),c

```

```
ld (de),a
inc l
jr mirror
```

## Is 34 Bytes Optimal?

Definitely not! After the deadline a solution was discovered that combines code from Tim Webber and Introspec's entries to mirror the screen in 33 bytes:

```
start:      ld hl,23296 ; Tim Webber/Introspec
loop1:      dec hl
            bit 6, h
            ret z
            ld a, h
            ld b,88
            cp b
            jr nc, noinv
doinv:      rlc (hl)
            rra
            djnz doinv
            ld (hl), a
noinv:      push hl
            bit 4,l
            jr nz, loop1
            pop de
            pop hl
            ld a,(de)
            ld c, (hl)
            ld (hl), a
            ex de, hl
            ld (hl), c
            jr loop1
```

Entries will be available shortly on [John Young's website](http://johnyoung.org/). Thanks to everyone who entered for making the contest a success :-)





➔ **03/30/15--08:29: Z80 Size Programming Challenge #3**

0 0

Recently I issued the third Z80 programming challenge for the ZX Spectrum:

This time the challenge is to write the shortest code to display a masked  $16 \times 16$  pixel sprite. The routine should be called with the address of the sprite data in one register and the X and Y screen coordinates in another. There's no need to save the screen area being overwritten or set the colours / attributes. This is somewhat trickier than previous challenges but more likely to be of practical use.

The deadline is Monday 30th March, midday (GMT).

Target: under 125 bytes.

- ➡ **The X and Y coordinates are in pixels with 0,0 at the top left.**
- ➡ **The sprite needs to be clipped if it goes over the screen edge.**
- ➡ **Sprite data can be formatted however you like within 64 bytes.**
- ➡ **Programs must return. The `RET` instruction is included in the size.**
- ➡ **So everyone has a fair chance comment with the code size not code.**
- ➡ **There are no prizes, just the chance to show off your coding skills.**

Solutions can be emailed to [digital.wilderness@googlemail.com](mailto:digital.wilderness@googlemail.com) or posted here after the deadline.

## Final Results

Congratulations to everyone who rose to the challenge, this was a tough one. Adrian claimed an impressive victory with a neat piece of self-modifying code. Here are the final results:

Coder	Size
Adrian Brown	68
John Metcalf	88
Ralph Becket	97
Arcadiy Gobuzov	99

## Winning Entry

Adrian Brown submitted an ingenious solution in only 68 byte. The code displays a sprite pixel by pixel in approx 18ms. The instruction at DS\_SetResOp is modified to set, reset or leave the appropriate bit.

```
DrawSprite:
    ; At most we want to draw 16 lines (lets store
    ; the 4 onto c as well as its saves a byte)
    ld bc, 01004h
DS_YLoop:
    ; Gotta be able to stop doing all this push/pop
    ; with exx at some point - but hey ho
    push bc
    push de

    ; Splitting is actually helpful as it gives us
    ; the byte increase on clipping :D
DS_XLoop1:
    ; Lets get that data byte
    ld b, 4
DS_XLoop2:
    ; Bit cheaty, roll the actual data, it will end
    ; up back as it started so thats fine
    ld a, (hl)
    rlca
    rlca
    ld (hl), a

    ; Store the data pointer
    push hl

    ; See if we want to draw or not, bit sneaky
    ; because of data layout
    or %10011111
    ld l,a

    ; Now calculate the screen address, start it
    ; here so carry is clear
    ld a,e
    rra
    ; Lets use the check to set the C flag
    cp 96
    jr nc, DS_SkipPixel
```

```
rra
or a
rra
push af
xor e
and %11111000
xor e
ld h,a

; Now work out the opcode for set/res bit (we need
; 01 for bit, 10 for res and 11 for set - so data
; needs to be 10 for bit, 01 for res and 00 for set)
ld a,d
and %00000111
rlca
rlca
; Thats nice, this will do the cpl for us on the
; bit number ;)
xor l
rla
ld (DS_SetResOp + 1),a

pop af
xor d
and %00000111
xor d

; Move across - check for clipping, do it here so
; we can use a as a value > 192
inc d
jr nz, DS_NoClipX

; Stick Y off the bottom so the rest of the line is clipped
; we can use a at this point as its got to be > 192
ld e, a
DS_NoClipX:

rrca
rrca
rrca
ld l,a
```

```

        ; Go set/res the bit
DS_SetResOp:
        set 0, (hl)
DS_SkipPixel:
        ; Store the data pointer
        pop hl

        ; Go do the byte of data
        djnz DS_XLoop2

        ; Now we need to move to the next bytes
        inc hl
        dec c
        jr nz, DS_XLoop1

        pop de
        pop bc

        ; Just increase down
        inc e
        djnz DS_YLoop
        ret

;*****
; Sprite Data twiddled a bit, Mask/Data/Mask/Data
; Mask = 0 we want the screen, set data to 1 means we convert
; the set/res into a bit which is fine, All rolled right
; three bit to get the pixel data i want in bits 3+4
;*****

SpriteData:
        db %10101010, %01001011, %10110100, %10101010
        db %10101010, %11110101, %01011111, %10101010
        db %11001010, %01011111, %11110101, %10110010
        db %01101011, %01010101, %01010101, %10111100
        db %11001101, %11010101, %01010101, %00110111
        db %11001101, %11110111, %01010101, %00110111
        db %01010111, %11010101, %01110101, %11010101
        db %01010111, %01010101, %01010101, %11011101
        db %01010111, %01010101, %01110101, %11010101
        db %01010111, %01010101, %11010101, %11011101
        db %11001101, %01010101, %01110111, %00110111

```

```

db %11001101, %11010101, %11011101, %00110111
db %01101011, %01110101, %01010111, %10111100
db %11001010, %01011111, %11110101, %10110010
db %10101010, %11110101, %01011111, %10101010
db %10101010, %01001011, %10110100, %10101010

```

Here my own solution in 88 bytes. This displays the sprite row by row and is slightly faster, taking approx 5ms.

; called with hl = address of sprite, de = position on screen

putsprite:

ld c,16

nextline:

ld a,d

and 7

inc a

ld b,a

ld a,e

rra

cp 96

ret nc

rra

or a

rra

push de

push hl

ld l,a

xor e

and 248

xor e

ld h,a

ld a,l

xor d

and 7

xor d

rrca

rrca

rrca

ld l,a

ld e,255

```
spd:
    ex (sp),hl
    ld a,(hl)
    inc hl
    ld d,(hl)
    inc hl
    ex (sp),hl

    push bc
    rrc e
    jr noshift
shiftspr:
    rra
    rr d
    rr e
noshift:
    djnz shiftspr

    push hl
    ld b,3
mask:
    bit 0,e
    jr z,bm1
    and (hl)
db 254 ; jr bm2
bm1:
    xor (hl)
bm2:
    ld (hl),a
    inc l
    ld a,l
    and 31
    ld a,d
    ld d,e
    jr z,clip
    djnz mask
clip:
    bit 0,e
    ld e,0
    pop hl
    pop bc
    jr nz,spd
```

```
pop hl
pop de
inc e
dec c
jr nz,nextline
ret

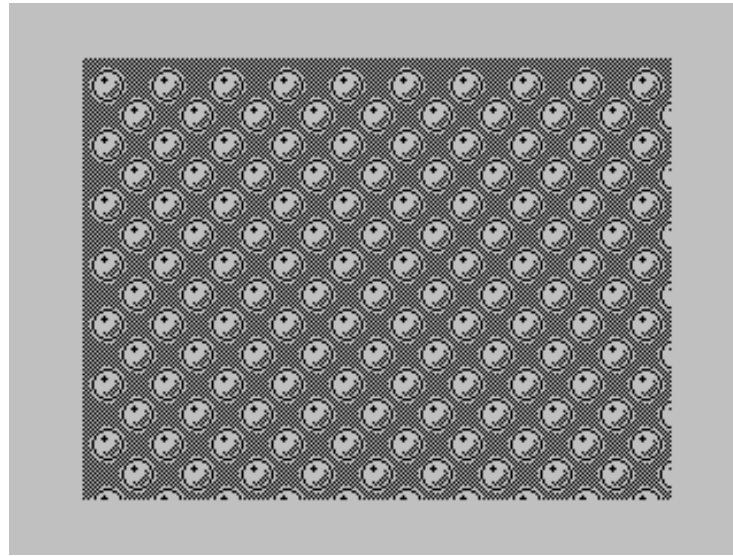
sprite:
db %11111100, %00111111, %00000000, %00000000
db %11110000, %00001111, %00000011, %11000000
db %11100000, %00000111, %00001100, %00110000
db %11000000, %00000011, %00010000, %00001000

db %10000000, %00000001, %00100010, %00000100
db %10000000, %00000001, %00100111, %00000100
db %00000000, %00000000, %01000010, %00010010
db %00000000, %00000000, %01000000, %00001010

db %00000000, %00000000, %01000000, %00010010
db %00000000, %00000000, %01000000, %00101010
db %10000000, %00000001, %00100000, %01010100
db %10000000, %00000001, %00100010, %10100100

db %11000000, %00000011, %00010001, %01001000
db %11100000, %00000111, %00001100, %00110000
db %11110000, %00001111, %00000011, %11000000
db %11111100, %00111111, %00000000, %00000000
```





➔ **04/06/15--15:16: Z80 Size Programming Challenge #4**

0 

0 



The fourth Z80 challenge for the ZX Spectrum was issued last week:

Back to something simple for the next challenge, a diagonal fade-to-white CLS. Write the shortest code to wipe the screen by increasing the ink colour of each character until it reaches white.

The clear should start at the top left and move one character across the screen per frame. The initial screen can be assumed to be monochrome — black text, white background, flash off, bright off. There's no need to clear the screen bitmap. Here's a demonstration of the clear in slow motion:

```

loading error, 0:1 R Tape loading
error, 0:1 R Tape loading error,
0:1 R Tape loading error, 0:1
R Tape loading error, 0:1 R Tape
loading error, 0:1 R Tape loadi
ng error, 0:1 R Tape loading err
or, 0:1 R Tape loading error, 0:
1 R Tape loading error, 0:1 R Ta
pe loading error, 0:1 R Tape loa
ding error, 0:1 R Tape loading e
rror, 0:1 R Tape loading error,
0:1 R Tape loading error, 0:1 R
Tape loading error, 0:1 R Tape l
oading error, 0:1 R Tape loading
error, 0:1 R Tape loading error,
0:1 R Tape loading error, 0:1
R Tape loading error, 0:1 R Tape
loading error, 0:1 R Tape loadi
ng error, 0:1 R Tape loading err
or, 0:1 R Tape loading error, 0:
1 R Tape loading error, 0:1 R Ta
pe loading error, 0:1 R Tape loa
ding error, 0:1 R Tape loading e
rror, 0:1 R Tape loading error,

```

Target: under 50 bytes.

The deadline is Monday 6th April, midday (GMT).

- ➡ **Your program shouldn't rely on the initial contents of registers.**
- ➡ **Programs must halt between frames. The HALT is included in the size.**
- ➡ **No RAM/ROM other than the attribute memory should be written to.**
- ➡ **Programs must return. The RET instruction is included in the size.**
- ➡ **So everyone has a fair chance comment with the code size not code.**
- ➡ **There are no prizes, just the chance to show off your coding skills.**

## Final Results

Congratulations to everyone who entered and Arcadiy Gobuzov who claimed first place with a solution in 26 bytes. Most of the solutions use LDDR to move the attribute data with anonymous and Ralph Becket being the two exceptions. Here are the final results:

Coder	Size
Arcadiy Gobuzov	26
ub880d	27
Bohumil Novacek	27
anonymous	27
Adrian Brown	27
John Metcalf	27
Ralph Becket	30
Jim Bagley	31
Paul Rhodes	31

## Winning Entry

Here's Arcadiy's winning entry in 26 bytes:

```
loop:      xor a ; if comment then 25, but exit if a==56 on start
           ld hl,#5ADF ;
           cp (hl) ;
           ld bc,#02E0 ; 23 lines of attributes
           ld de,#5AFF ;
           lddr ; move down attributes
           ld c,e ; e = #1F
           add hl,bc ;
           lddr ; roll upper line of attributes to right
           halt
           ret z
           ld a,(de) ; de = first address of attributes
           cp #3F ;
```

```
adc a,c      ; add 0 or 1 (carry)
ld (de),a    ; now a in range [38..3f]
jr loop
```

Here's my own solution in 27 bytes. Unfortunately I missed the final CP (HL) to squeeze out the last byte:

```
fadetowhite:
    ld de,23295 ; 90 255
    ld a,(de)
    cp 63
    ret z
    ld hl,23263 ; 90 223
    ld bc,736   ; 2 224
    halt
    lddr
    ld c,e
    add hl,bc
    lddr
    ld a,(de)
    cp 63
    adc a,c
    ld (de),a
    jr fadetowhite
```

Here's an alternative — a fade-to-black wipe (from white ink, black paper, no bright, no flash) in 25 bytes:

```
fadetoblack:
    ld de,23295 ; 90 255
    ld a,(de)
    or a
    ret z
    ld hl,23263 ; 90 223
    ld bc,736   ; 2 224
    halt
    lddr
    ld c,e
    add hl,bc
    lddr
    ld a,(de)
    add a,l
```

```
sbcb a,l  
ld (de),a  
jr fadetoblack
```

➡ **07/26/15--23:18: Z80 Size Programming Challenge #5**

0  0   

Recently I issued the fifth Z80 challenge for the Sinclair Spectrum:

This time the challenge is to write a solid flood fill routine to fill a region of unset pixels, bounded in 4 directions (up, down, left, right) by set pixels or the screen edge. The routine should be called with the X and Y coordinates in a register. There's no need to set the screen attributes.

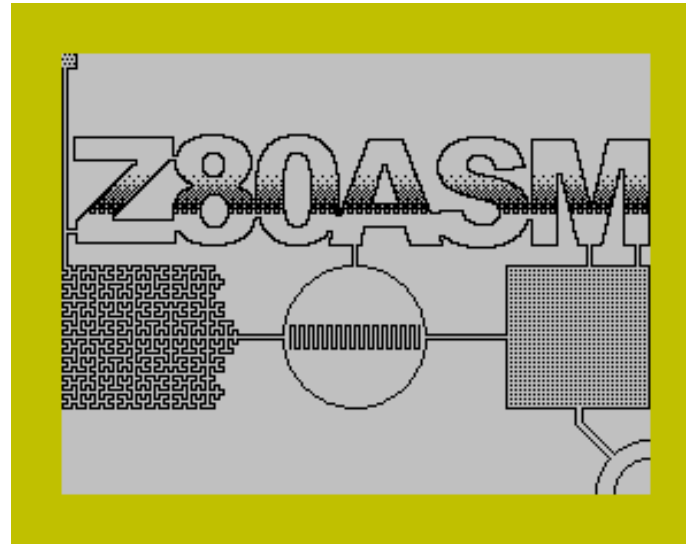
Scoring is multi-objective: a routine will be judged by the code size and stack space required to fill a test image. Your routine will be awarded one point for each competing routine it is smaller \*and\* uses less stack space than. The routine(s) with the most points will be declared winner(s).

The deadline is Wednesday 22nd July, midday (GMT).

- ➡ **The X and Y coordinates are in pixels with 0,0 at the top left.**
- ➡ **No memory other than the screen, stack and your routine can be written.**
- ➡ **If you call a ROM routine it's size will be added to your code size.**
- ➡ **Programs must return. The RET instruction is included in the size.**

- ➡ **So everyone has a fair chance comment with the code size not code.**
- ➡ **There are no prizes, just the chance to show off your coding skills.**

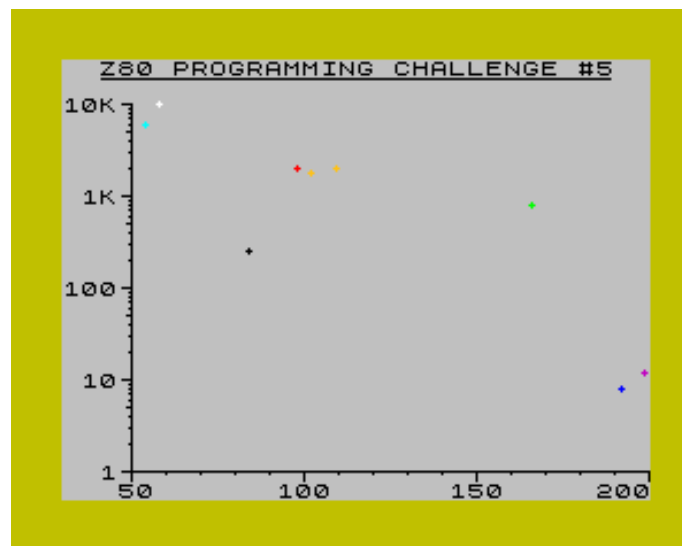
The test image is designed to check correct behaviour at the screen boundary and to be pathological — triggering suboptimal behaviour in some common flood fill algorithms:



## Final Results

Congratulations to everyone who coded a working flood fill and to Dworkin Z Amberu who claimed first place by being shorter and using less memory than competing entries.

Entries have been plotted on this genuine fake Spectrum screenshot. If the graph is empty below and to the left of an entry, that entry is in first place:



Colour	Coder	Code	Memory	Time
Red	John Metcalf	98	~2K	2.1 seconds
Orange	Paul Rhodes	102	~1.8K	3.2 seconds
Yellow	Ralph Becket	109	~2K	8.8 seconds
Green	Miguel Jódar	166	~800 bytes	4.8 seconds
White	Dworkin Z Amberu	58	~9.8K	28.6 seconds
Cyan	John Metcalf	54	~6.1K	28.6 seconds
Black	Dworkin Z Amberu	84	~270 bytes	40 seconds
Blue	Dworkin Z Amberu	192	8 bytes	~40 minutes
Purple	Adrian Brown	199	12 bytes	~3 hours?

## Shortest Entry

The simplest entry is a recursive routine weighing in at 54 bytes. Despite being too heavy on the stack to score well it's one of the easiest to understand. Each time the routine is

called it checks whether or not the pixel at X,Y is set. If not the pixel will be set then the fill routine is called recursively with the pixels up, down, left and right of the current pixel:

; called with e = X horizontal, d = Y vertical

FILL:

```
ld b,e
ld a,d
and 248
rra
cp 96
ret nc
rra
rra
ld l,a
xor d
and 248
xor d
ld h,a
ld a,e
xor l
and 7
xor e
rrca
rrca
rrca
ld l,a
ld a,128
```

PLOTBIT:

```
rrca
djnz PLOTBIT
or (hl)
cp (hl)
ret z
ld (hl),a
inc e
call nz,FILL
dec e
dec de
call ZFILL
inc de
call FILL
inc d
```



```
inc d
ZFILL:
call nz,FILL
dec d
ret
```

The winning entries will be available shortly.

➔ **10/03/15--17:56: The Matrix Digital Rain for the ZX Spectrum**

0  0   

A few days ago I coded The Matrix digital rain effect, a fictional representation of the code for the virtual reality of The Matrix. The technique is simple: fill the screen with random characters and scroll down columns of attributes, occasionally switching between black and green.

Here's the final code - 147 bytes of Z80 using the default Sinclair font:

```
org 08000h

; black border / black attributes

xor a
out (0FEh),a
ld hl,05AFFh
attr: ld (hl),a
      dec hl
      bit 2,h
      jr z,attr

; fill screen with random characters

ld e,a
fillscr:ld d,040h
```

```
fill:   call rndchar
        ld a,d
        cp 058h
        jr nz,fill
        inc e
        jr nz,fillscr

; digital rain loop

frame:  ld b,06h
        halt
column: push bc

; randomize one character

        call random
        and 018h
        jr z,docol
        add a,038h
        ld d,a
        call random
        ld e,a
        call rndchar

; select a random column

docol:  call random
        and 01Fh
        ld l,a
        ld h,058h

; ~1% chance black -> white

        ld a,(hl)
        or a
        ld bc,0247h
        jr z,check

; white -> bright green

white:  cp c
        ld c,044h
```

```
        jr z,movecol

; bright green -> green

        cp c
        ld c,04h
        jr z,movecol

; ~6% chance green -> black

        ld bc,0F00h
check:   call random
        cp b
        jr c,movecol
        ld c,(hl)

; move column down

movecol:ld de,020h
        ld b,018h
down:    ld a,(hl)
        ld (hl),c
        ld c,a
        add hl,de
        djnz down
        pop bc
        djnz column

; test for keypress

        ld bc,07FFEh
        in a,(c)
        rrca
        jr c,frame
        ret

; display a random glyph

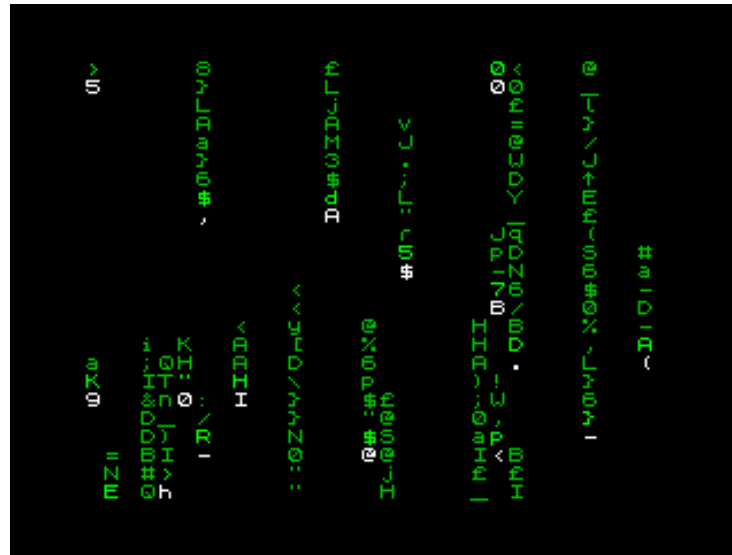
rndchar:call random
crange:  sub 05Fh
        jr nc,crange
        add a,a
```

```
        ld l,a
        ld h,0
        add hl,hl
        add hl,hl
        ld bc,(05C36h)
        add hl,bc
        ld b,8
char:   ld a,(hl)
        ld (de),a
        inc d
        inc hl
        djnz char
        ret
```

; get a byte from the ROM

```
random: push hl
        ld hl,(seed)
        inc hl
        ld a,h
        and 01Fh
        ld h,a
        ld (seed),hl
        ld a,(hl)
        pop hl
        ret
```

seed:



→ 05/27/16--13:31: [Langton's Ant for the ZX Spectrum](#)

0



0



**Langton's Ant** is an automata which creates a complex pattern by following a couple of simple rules:

- **If the ant is on an empty pixel, turn 90° right, set the pixel then move forward**
- **If the ant is on a set pixel, turn 90° left, reset the pixel then move forward**

The ant's path appears chaotic at first before falling into a repetitive “highway” pattern, moving 2 pixels diagonally every 104 cycles.

Here's the code to display Langton's Ant on the ZX Spectrum in 61 bytes. It runs in just over a second so you might want to add a halt to slow things down:

```
org 65472

ld de,128*256+96

ANT:
; halt
ld a,c      ; check direction
and 3
rrca
add a,a
dec a
jr nc,XMOVE

add a,e      ; adjust y position +/-1
ld e,a
cp 192
ret nc
xor a

XMOVE:
add a,d      ; adjust x position +/-1
ld d,a

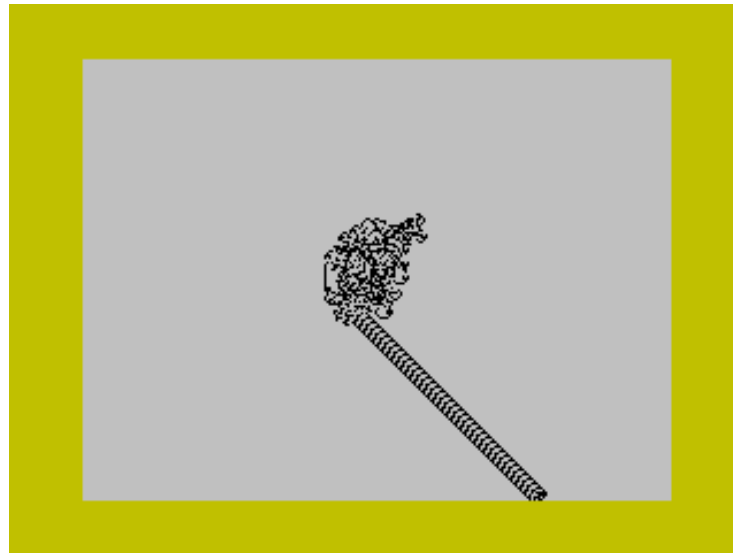
; -----
and 7        ; calculate screen address
ld b,a
inc b
ld a,e
rra
scf
rra
or a
rra
ld l,a
xor e
and 248
xor e
ld h,a
ld a,d
xor l
and 7
xor d
```

```
rrca
rrca
rrca
ld l,a
ld a,1
PLOTBIT:
rrca
djnz PLOTBIT
; -----

ld b,a      ; test pixel
and (hl)

jr nz,LEFT  ; turn left/right
inc c
inc c
LEFT:
dec c

ld a,b      ; flip pixel
xor (hl)
ld (hl),a
jr ANT
```



## ➡ 05/29/16--01:19: Divide and Conquer Line Algorithm for the ZX Spectrum

1  0   

While attempting to write a game in 256 bytes I needed a routine to draw lines, but Bresenham's line algorithm weighs in at approx ~120 bytes. The only suitable alternative I'm aware of is recursive divide and conquer: divide a line into two smaller lines and call the draw routine with each in turn:

```
/* Draw a line from (ax,ay) to (bx,by) */

int draw ( ax, ay, bx, by )
{
    int midx, midy;
    midx = ( ax+bx ) / 2;
    midy = ( ay+by ) / 2;
    if ( midx != ax && midy != ay )
    {
        draw( midx, midy, ax, ay );
        draw( bx, by, midx, midy );
        plot( midx, midy );
    }
}
```

This is significantly smaller than Bresenham's, 32 bytes of Z80. However, there are a couple of compromises: it's slower and the lines aren't perfect because the rounding errors accumulate.

```
; draw lines using recursive divide and conquer
; from de = end1 (d = x-axis, e = y-axis)
; to   hl = end2 (h = x-axis, l = y-axis)
```

```
DRAW:
    call PLOT

    push hl
```



```
; calculate hl = centre pixel

ld a,l
add a,e
rra
ld l,a
ld a,h
add a,d
rra
ld h,a

; if de (end1) = hl (centre) then we're done

or a
sbc hl,de
jr z,EXIT
add hl,de

ex de,hl
call DRAW    ; de = centre, hl = end1
ex (sp),hl
ex de,hl
call DRAW    ; de = end2, hl = centre

ex de,hl
pop de
ret

EXIT:
pop hl
ret

; -----

; plot d = x-axis, e = y-axis

PLOT:
push hl
ld a,d
and 7
ld b,a
inc b
```

```
ld a,e
rra
scf
rra
or a
rra
ld l,a
xor e
and 248
xor e
ld h,a
ld a,l
xor d
and 7
xor d
rrca
rrca
rrca
ld l,a
ld a,1
PLOTBIT:
rrca
djnz PLOTBIT
or (hl)
ld (hl),a
pop hl
ret
```

Alternatively the `de(end1) = hl(centre)` test can be replaced with a recursion depth count to create an even slower 28 byte routine:

```
; draw lines using recursive divide and conquer
; from de = end1 (d = x-axis, e = y-axis)
; to   hl = end2 (h = x-axis, l = y-axis)
```

```
DRAW:
ld c,8
```

```
DRAW2:
dec c
jr z,EXIT
```

```
push de

; calculate de = centre pixel

ld a,l
add a,e
rra
ld e,a
ld a,h
add a,d
rra
ld d,a

call DRAW2 ; de = centre, hl = end1
ex (sp),hl
call DRAW2 ; de = centre, hl = end2

call PLOT
ex de,hl
pop hl
EXIT:
inc c
ret
```

