

Retro Programming – Itsy Forth

Original content by John Metcalf (2012)

Compiled by Will Senn

2025-09-22

Contents

Itsy-Forth: the Outer Interpreter of a 1K Tiny Compiler	3
Outer Interpreter	3
Itsy-Forth: the Dictionary and Inner Interpreter	5
Forth Dictionary	5
Macros	5
Macro Examples	6
Register Allocation	7
Itsy’s Inner Interpreter	7
Itsy Forth: Implementing the Primitives	9
Peek and Poke	9
Manipulating the Stack	9
Flow Control	10
Variables and Constants	11
Compilation	11
Maths / Logic	12
Handling Strings	12
Terminal Input / Output	14
Searching the Dictionary	16
Initialisation	17
Up and Running?	17
Itsy Forth: The Compiler	19
Colon Definitions	19
Creating Headers	19
Constants	20
Testing the Compiler	20
Itsy Forth: The Next Step	21
Itsy: Documenting the Bit-Twiddling & Voodoo Magic	35

Wednesday, 14 March 2012

Itsy-Forth: the Outer Interpreter of a 1K Tiny Compiler

Itsy Forth is a tiny indirect-threaded compiler for a subset of the Forth programming language. The compiler contains an interactive interpreter which allows the programmer to define new words (subroutines), execute them and even extend the language.

Each word has an entry in the Forth dictionary. New words are defined with `:` and end with `;`. The following example defines a new word `inc` which calls two words, `1` and `+`:

```
: inc 1 + ;
```

Outer Interpreter

Itsy was developed top-down, first implementing the interpreter. `interpret` takes the next word from the input buffer and searches for it in the dictionary. If the word isn't found, Itsy attempts to convert it to a number. In compile mode, Itsy will add the word or number to the current definition. Otherwise the word will be executed.

```
: interpret
begin
  #tib @ >in @ =
  if
    tib 50 accept #tib ! 0 >in !
  then
    32 word find dup
    if
      state @ =
      if
        ,
      else
        execute
      then
    else
      dup rot count >number
      if
        state @
        if
          last @ dup @ last ! dp !
        then
          abort
      then
      drop drop state @
      if
        ['] lit , ,
      then
```

```
    then  
again  
;
```

The Itsy interpreter uses 23 different words, 5 variables (state, >in, #tib, dp, last) and a constant (tib). Note the variables dp and last are non-standard. dp points to the first free cell in the data area. last points to the last word to be compiled.

Next we'll define the dictionary structure and implement the inner interpreter. In the meantime I'd love to hear any comments on the code so far :-)

Tuesday, 3 April 2012

Itsy-Forth: the Dictionary and Inner Interpreter

Itsy Forth is a minimal Forth compiler implemented in under 1kB. Earlier we examined Itsy's outer interpreter. Now we take a closer look at the dictionary and inner interpreter.

Forth Dictionary

Itsy's dictionary is a linked list holding the name and code for each word (subroutine). Each entry in the list has a header containing a link, counted string and XT (execution token). For example here's the dictionary entry for nip:

```
        ; header
        dw link_to_previous_word
        db 3, 'nip'
xt_nip  dw docolon
        ; body
        dw xt_swap
        dw xt_drop
        dw xt_exit
```

The first line of the header links to the previous word in the dictionary. The second line holds the word's name preceded by its length. The final line contains the XT, a pointer to the routine which performs the actual operation of the word. Itsy uses four different XTs:

- docolon - The word is a list of pointers to XTs. Call each in turn.
- doconst - The word is a constant. Place its value on the data stack.
- dovar - The word is a variable. Place its address on the data stack.
- pointer to body - The word is a primitive (machine code). Execute it.

Macros

I'm not a big fan of macros. They're ugly and lock the code to a particular assembler. On the other hand they can add flexibility and make the code less prone to errors. Compare the definition of + with and without macros:

Without macros:

```
        dw link_to_previous_word
        db 1, '+'
xt_plus dw mc_plus
mc_plus pop ax
        add bx,ax
        jmp next
```

With macros:

```
primitive '+',plus
```

```

pop ax
add bx,ax
jmp next

```

The NASM macros to set up headers and maintain the linked list are pretty simple:

```

%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

```

Macro Examples

constant is used to define a Forth constant. E.g. to define false = 0:

```
constant 'false',false,0
```

variable creates a Forth variable. E.g. to create base and initialise to 10:

```
variable 'base',base,10
```

primitive sets up an assembly language word. E.g. to create drop:

```
primitive 'drop',drop
```

```

    pop bx
    jmp next

```

colon defines a compiled Forth word. E.g. to define nip:

```

    colon 'nip',nip
    dw xt_swap
    dw xt_drop
    dw xt_exit

```

Register Allocation

Itsy's use of the registers is similar to most 8086 Forths. The system stack is used for the data stack while a register is used for the return stack. Note the top element of the data stack is kept in a register to enhance performance:

- sp - data stack pointer
- bp - return stack pointer
- si - Forth instruction pointer
- di - pointer to current XT
- bx - TOS (top of data stack)

Itsy's Inner Interpreter

The Forth inner interpreter needs only three simple routines:

- docolon - the XT to enter a Forth word. Save the Forth IP on the return stack then point it to the word being entered.
- exit - return from a compiled Forth word. exit recovers the Forth IP from the return stack.
- next - return from a primitive (machine code) word and call the next XT.

```

docolon dec bp
        dec bp
        mov word[bp],si
        lea si,[di+2]

next    lodsw
        xchg di,ax
        jmp word[di]

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

```

Next we'll define approx 30 words and finally get the interpreter up and running. In the meantime I'd love to hear any comments on the code so far :-)

Sunday, 15 April 2012

Itsy Forth: Implementing the Primitives

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 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
    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_numl

```

```

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.

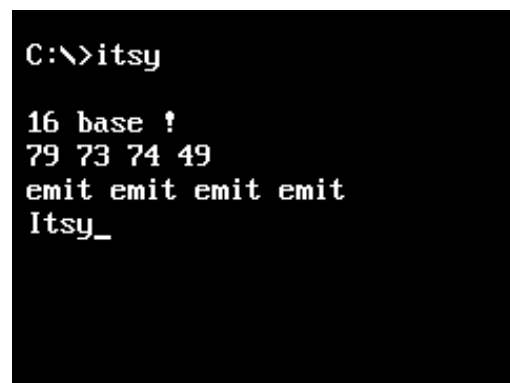
```

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. To verify the interpreter, we perform 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 :-)

Saturday, 23 June 2012

Itsy Forth: The Compiler

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 comes the moment of truth (see Figure 2). Running `A B C` confirms 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, the code for the current version of Itsy is on the following pages.

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_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_numl
to_numz push ax
to_numh push cx
        push di
        jmp next

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

        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

        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
findn   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:

```

Tuesday, 4 September 2012

Itsy: Documenting the Bit-Twiddling & Voodoo Magic

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.

Itsy with all the bit-twiddling hacks and voodoo magic documented by Mike are on the following pages.

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 head'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_ %+ %2 dw %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 2-3 0

```

```

; 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 2-3 0

```

```

; 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_ %+ %2 dw %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_ %+ %2 dw %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.
;-----
org 0100h

;-----
; 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
mov ax,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).
mov word[val_to_in],ax        ; Save the same number to >in.
xor bp,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 OFFFEh and OFFFh,
; the very end of memory space, and
; the stack will grow downward from
; there.

mov word[val_state],bp ; Clear the value of state.
mov sp,stack0          ; Set the stack pointer to the value
                        ; defined above.

mov si,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
    mov di,word[val_dp] ; Put the value of dp into the DI register.
    xchg ax,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.
    mov word[val_dp],di ; Store the incremented value in DI as the
                        ; new value for the dictionary pointer.
    pop bx              ; 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
    push bx             ; 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.
xchg ax,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
pop dx ; Unload "y" from the stack.
pop ax ; Unload "x" from the stack. Remember that "z" is
; already in BX.
push dx ; Push "y" back onto the stack.
push bx ; Push "z" down into the stack on top of "y".
xchg ax,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
pop bx ; 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
push bx ; 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
pop ax ; Pop "x", the number 2nd from the top, into AX.
push bx ; Push "y", the former top of the stack.
xchg ax,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
    pop ax        ; Pop the value of "x" off of the stack.
    add bx,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
    pop ax        ; Get the "x" value into a register.
    sub bx,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...
    sub bx,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.
    sbb bx,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
    mov bx,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
    pop word[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).
    pop bx            ; 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.
    xchg di,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]".
    jmp word[di]       ; Jump and start executing code at the address pointed
                        ; to by the value in the DI register.

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

```

```

; Obranch - ( 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 Obranch 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 'Obranch',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.
    test bx,bx     ; See if there's a 0 at the top of the stack.
    jne zerob_z    ; If it's not zero, jump.
    xchg ax,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 pop bx     ; 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 Obranch,
; 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
    mov si,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
    mov di,bx      ; Move the jump-to address to the DI register.
    pop bx         ; Pop the next number on the stack into the TOS.
    jmp word[di]   ; Jump to the address pointed to by the DI register.

; exit - ( -- ) return from the current word
    primitive 'exit',exit
    mov si,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.
    inc bp         ; Now we have to increment BP twice to do a manual
                    ; "pop" of the return stack pointer.
    inc bp         ;
    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
    inc bx         ; Increment the address past the length byte so
                    ; it now points to the actual string.
    push bx        ; Push the new address onto the stack.
    mov bl,byte[bx-1] ; Move the length byte into the lower half of
                    ; the BX register.
    mov bh,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.
    pop di                      ; Put the address into di.
    pop cx                      ; Put the high word of the double value into cx
    pop ax                      ; and the low word of the double value into ax.
to_numl test bx,bx              ; Test the length byte.
    je to_numz                  ; If the string's length is zero, we're done.
                                ; Jump to end.
    push ax                     ; Push the contents of ax (low word) so we can
                                ; use it for other things.
    mov al,byte[di]             ; Get the next byte in the string.
    cmp al,'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.
    sub al,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 cmp al,'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.
    cmp al,'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.
    sub al,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 sub al,48          ; Convert the digit to its corresponding
                           ; number. This op could also have been
                           ; written as "sub al,'0'"
mov ah,0                  ; Clear the ah register. The AX reg now
                           ; contains the numeric value of the new digit.
cmp al,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.)
xchg ax,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.)
pop ax                   ; Recall the low word of our accumulated
                           ; double number and load it into AX.
push dx                  ; Save the digit value. (The DX register
                           ; will get clobbered by the upcoming mul.)
xchg ax,cx               ; Swap the low and high words of our double
                           ; number. AX now holds the high word, and
                           ; CX the low.
mul word[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.
xchg ax,cx               ; Save the product of the first mul to the CX
                           ; register and put the low word of our double
                           ; number back into AX.
mul word[val_base]       ; 16-bit multiply the low word of our converted
                           ; double number by the base, then add the high
                           ; word of the product to the low word of the
                           ; first mul (i.e. do the carry).
pop dx                   ; Recall the digit value, then add it in to
add ax,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.
dec bx      ; Decrement the length.
inc di      ; Inc the address pointer to the next byte
            ; of the string we're converting.
jmp to_numl ; Jump back and convert any remaining
            ; characters in the string.
to_numz push ax      ; Push the low word of the accumulated total
                    ; back onto the stack.
to_numh push cx      ; Push the high word of the accumulated total
                    ; back onto the stack.
push di             ; 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
pop di      ; Pop the address of the string buffer into DI.
xor cx,cx   ; Clear the CX register.
acceptl call getchar ; Do the bios call to get a chr from the keyboard.
cmp al,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.
mov al,' '   ; Then output a space to wipe out the character that
call outchar ; the user had just typed.
mov al,8     ; Then output another backspace to put the cursor
call outchar ; back into position to read another character.
dec cx       ; We just deleted a character. Now we need to decrement
dec di       ; both the counter and the buffer pointer.
jmp acceptl  ; Then go back for another character.

```

```

acceptn cmp al,13      ; See if the input chr is a carriage return.
        je acceptz     ; If so, we're done. jump to the end of the routine.
        cmp cx,bx      ; Compare current string length to the maximum allowed.
        jne accepts    ; If the string's not too long, jump.
acceptb mov al,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.
        inc cx         ; 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.
        mov al,13      ; Send a carriage return to the display...
        call outchar   ;
        mov al,10      ; ...followed by a linefeed.
        call outchar   ;
        mov bx,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
        mov di,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".
        push di                  ; Save the pad pointer to the stack.
        mov dx,bx                ; Copy the word separator to DX.
        mov bx,word[val_t_i_b]   ; Load the address of the input buffer
        mov cx,bx                ; into BX, and save a copy to CX.
        add bx,word[val_to_in]   ; Add the value of >in to the address
                                   ; of tib to get a pointer into the
                                   ; buffer.
        add cx,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    cmp cx,bx                ; Compare the current buffer pointer to
                                           ; the end-of-buffer pointer.
        je wordz                ; If we've reached the end, jump.
        mov al,byte[bx]         ; Get the next chr from the buffer
        inc bx                  ; and increment the pointer.
        cmp al,dl               ; See if it's the separator.
        je wordf                ; If so, jump.
wordc    inc di                  ; 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.
        mov byte[di],al         ; Write the new chr to the pad.
        cmp cx,bx               ; Have we reached the end of the
                                           ; input buffer?
        je wordz                ; If so, jump.
        mov al,byte[bx]         ; Get another byte from the input
        inc bx                  ; buffer and increment the pointer.
        cmp al,dl               ; Is the new chr a separator?
        jne wordc               ; If not, go back for more.
wordz    mov byte[di+1],32        ; Write a space at the end of the text
                                           ; we've written so far to the pad.
        mov ax,word[val_dp]     ; Load the address of the pad into AX.
        xchg ax,di              ; Swap the pad address with the pad
        sub ax,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.
        mov byte[di],al         ; Save the length byte into the first
                                           ; byte of the pad.
        sub bx,word[val_t_i_b]  ; Subtract the base address of the
                                           ; input buffer from the pointer value
                                           ; to get the new value of >in...
        mov word[val_to_in],bx  ; ...then save it to its variable.
        pop bx                  ; 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
        xchg ax,bx              ; Move our output character to the AX register.
        call outchar            ; Send it to the display.
        pop bx                  ; Pop the argument off the stack.

```



```

        jmp next

getchar mov ah,7   ; This headerless routine does an MS-DOS Int 21h call,
        int 021h   ; reading a character from the standard input device into
        mov ah,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 xchg ax,dx  ; This headerless routine does an MS-DOS Int 21h call,
        mov ah,2    ; sending a character in the DL register to the standard
        int 021h    ; 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
        mov di,val_last      ; Get the address of the link field of the last
                               ; word in the dictionary. Put it in DI.
findl  push di                ; Save the link field pointer.
        push bx               ; Save the address of the name we're looking for.
        mov cl,byte[bx]       ; Copy the length of the string into CL
        mov ch,0              ; Clear CH to make a 16 bit counter.
        inc cx                ; Increment the counter.
findc  mov al,byte[di+2]      ; Get the length byte of whatever word in the
                               ; dictionary we're currently looking at.
        and al,07Fh           ; Mask off the immediate bit.
        cmp al,byte[bx]       ; Compare it with the length of the string.
        je findm              ; If they're the same, jump.
        pop bx                ; Nope, can't be the same if the lengths are
        pop di                ; different. Pop the saved values back to regs.
        mov di,word[di]       ; Get the next link address.
        test di,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 push bx                ; End of dictionary. Word wasn't found. Push the
                               ; string address to the stack.

```

```

        xor bx,bx          ; Clear the BX register (make a "false" flag).
        jmp next          ; Return to caller.
findm   inc di             ; 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.)
        inc bx            ; 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.
        pop bx            ; 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.
        pop di            ; Restore the link field address for the dictionary
                           ; word whose name we just looked at.
        mov bx,1          ; Put a 1 at the top of the stack.
        inc di            ; Increment the pointer past the link field to the
        inc di            ; name field.
        mov al,byte[di]   ; Get the length of the word's name.
        test al,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.
        neg bx            ; 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   and ax,31         ; Mask off all but the valid part of the name's
                           ; length byte.
        add di,ax         ; Add the length to the name field address then

```

```

        inc di                ; add 1 to get the address of the code field.
        push di               ; 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

```

```

docolon dec bp      ; creating.
dec bp              ; Here's the runtime code for colon words.
                    ; 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.
mov word[bp],si     ; Then we write that address directly from the
                    ; instruction pointer to that location.
lea si,[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
; that address as the new value of dp.
dw xt_dp,xt_store
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 push bx ; Push the stack to make room for the new value
; we're about to put on top.
lea bx,[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
    mov di,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.
    mov al,byte[di+2]      ; Get the length byte from the name field.
    and ax,31             ; Mask off the immediate bit and leave only
                           ; the 5-bit integer length.
    add di,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.
    mov word[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.
    mov si,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.
    inc bp                ; Then manually increment the return stack
    inc bp                ; 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 push bx      ; Push the stack down.
    mov bx,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
interpret 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
    dw 50                    ; 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
        dw 0             ; input buffer.
        dw xt_to_in      ;
        dw xt_store      ;
intpar  dw xt_lit        ; Put a 32 on the stack to represent an ASCII
        dw 32            ; 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.

```
