

meusForth and xVM

What is meusForth and xVM?

xVM is a true byte-code interpreter. It's small, compact, and portable. It's written in ANSI C99 and can be compiled for a desktop or an embedded environment and used as a standalone interpreter engine. When compiled as a library on OS X, xVM is approximately 24 KB in size.

meusForth is a custom Forth language implementation - Forth is a stack based programming language with an RPN (Reverse Polish Notation) syntax invented by Chuck Moore back in the 1960s. meusForth presents a command-line interface to an outer-interpreter that's used for writing, running, compiling, and debugging code that runs on the xVM interpreter engine. meusForth's core functionality consists of a terminal interface, a token lexer, and a byte-compiler.

With meusForth you do the following from the command-line:

- Interactively program, run, and debug code.
- Run code from a text file.
- Compile byte-code and save as binary data to a file and import the file back in to meusForth.
- Compile byte-code and save as a "C" array to a file.
- Learn a new programming paradigm and have fun! ☺

What meusForth and xVM are not?

meusForth/xVM are not a full blown ANS Forth. meus is Latin for my, this is my own little implementation of Forth. Although I plan on expanding meusForth for use in a desktop environment and adding more primitives to xVM the main goal is to one day do a proof of concept on how to use a VM on an embedded system for testing C/C++ code without requiring debug and/or test conditionals or compilation switches. [add more details?]

Why did I create meusForth and xVM?

The main reason was to learn to program in Forth; it's a really neat language, and environment. The second reason was to have a drop-in Forth replacement for the LeJos Java Virtual Machine - LeJos is a system that runs on the Lego NXT brick. The third reason is that there's something mystical about virtual machines and I thought it'd be cool to write one myself as opposed to doing more porting. I've ported the Pawn, Wonka, and Nano VM's to the C64XX. Yes, much fun...

How would you use meusForth and xVM?

meusForth can be used in [just] a desktop environment but its strength is in its modularity. You can write and debug your Forth code in a desktop environment then compile the code to a binary file or a "C" array. The byte-code in the "C" array can be compiled along with the xVM interpreter as part of a larger program, e.g., on an embedded processor, or the compiled byte-code in the binary file can be delivered as a data stream to a process that's already running the xVM interpreter.

An example. Say xVM is running within a thread on a TI C64XX DSP on the baseband board of a BTS. Say the BTS has an Ethernet connection to the outside world and that there's a General Purpose Process (GPP) that runs a UDP server that accepts incoming data packets. The GPP processes the incoming messages based on a predefined communications protocol and can forward messages for the DSP to the DSP. You write a meusForth program that tells the DSP on the baseband board to blink its LED's once per second. You compile your program to a binary file and use a UDP client to read the data from the file and send it to the BTS. The GPP forwards the message to the DSP, the DSP copies the data to some memory location then forwards the memory's pointer, which is the start of the byte-code, to xVM. xVM consumes the byte-code stream by sending it through its interpreter engine. And Voila, the LED's start blinking.

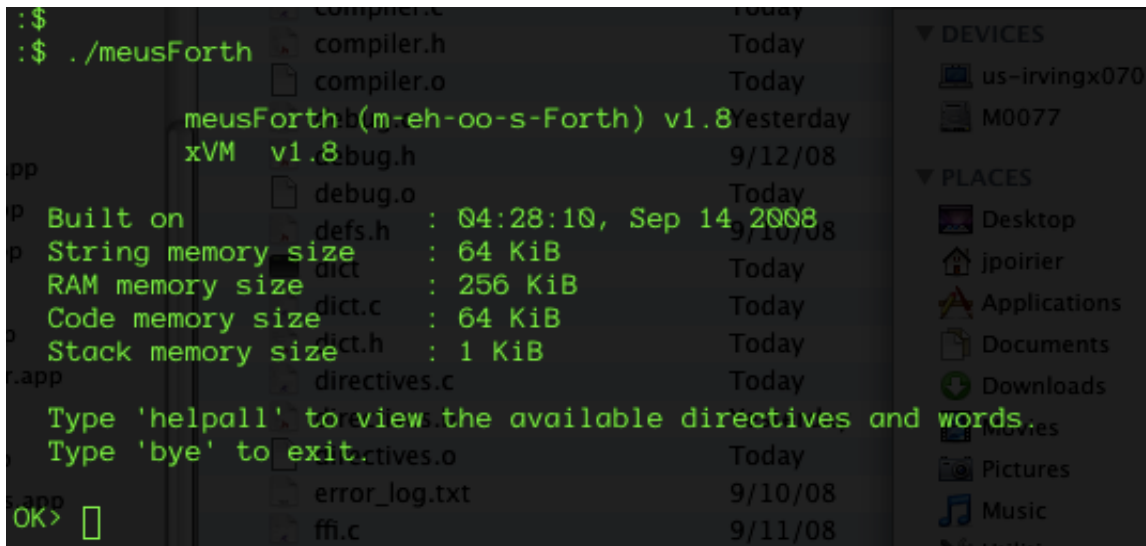
Why you should give it a try?

To expand your mind, or just to help a fella out. Having just one other person pounding on this thing would help tremendously with the debugging effort.

Note that the enclosed “stackflow” tutorial is a really nice (quick and painless) way to learn about stacks as well as the Forth language.

The meusForth command-line

When starting meusForth you’ll be presented with some version and buffer information as well as the interactive command prompt:



TBD: add buffer descriptions, e.g. size as it relates to code development

To view the available directives and words type “helpall” at the command prompt. Note that “words” in forth are like functions in C\C++. Type “bye” to exit and note that spaces are significant.

What are directives?

They’re commands and instructions for the compiler. The following is a list of the current directives that are supported.

DIRECTIVES

HELP	Provides help on a specific word or directive. Ex: HELP MYWORD
HELPALL	List all the existing words and directives
IF	(flag --) Start of conditional 'IF' statement .. IF .. ELSE .. THEN
ELSE	(--) Optional part of conditional 'IF' statement .. IF .. ELSE .. THEN
THEN	(--) Finish of conditional 'IF' statement .. IF .. ELSE .. THEN
BEGIN	(--) Start of BEGIN ... UNTIL loop
UNTIL	(flag --) Finish of BEGIN ... UNTIL loop, where loop repeats if flag equals 0
DO	(Ne Ns --) (R: -- Ne Ns) Starts a counted loop. Cycle from Ns to Ne (also LOOP, I, J) Ex: 10 0 DO
LOOP	(R: Ne Ns --) Finish a counted loop. Cycle from Ns to Ne (also DO, I, J)
I	(-- x) Put the inner loop counter on the stack. Also see DO, LOOP, I
J	(-- x) Put the outer loop counter on the stack. Also see DO, LOOP, J
LEAVE	(--) Immediately exit the current loop. Also see DO, LOOP",
:	(--) Start of a word definition. Ex: : FOO 1 2 + ;
;	(--) Ends a word definition.
FILE	Opens the specified text file. Ex: FILE my_file.f

FILEB	Opens the specified binary file. Ex: FILE my_file.fd
INCLUDE	Include the specified file, files can be nested. Ex: INCLUDE my_file.f
#INCLUDE	Include the specified file. Files can be nested. Ex: #INCLUDE my_file.f
COMPA	Compile vocabulary to a file in array format. Ex: COMPA byte_array.c
COMPB	Compile vocabulary to a file in binary format. Ex: COMPB forthFile.fd
#DEFINE	Create a simple 'C' like macro definition. Ex: #define PortA 0x80
RESET	Reset the compiler
DEBUG	Switch on debug mode
NODEBUG	Switch off debug mode
BASE	Indicate the current base - HEX or DECIMAL
HEX	Change the base to HEX
RECURSE	Recursively call the function that recurse is defined in.
DECIMAL	Change the base to DECIMAL

What are words?

They're the built-in primitives for the actions to be performed. There are two types of words, user defined and internal primitives. The following is a list of the current primitives that are supported.

WORDS

DROP	(x --) Remove x from the top of the stack
2DROP	(x1 x2 --) Drop the cell pair x1 and x2 from the stack
DUP	(x -- x x) Duplicate x
2DUP	(x1 x2 -- x1 x2 x1 x2) Duplicate the cell pair x1 x2
SWAP	(x1 x2 -- x2 x1) Swap the top two cell pairs
PICK	(+n -- x) Put a copy of the n'th stack item on the top of the stack
2SWAP	(x1 x2 x3 x4 -- x3 x4 x1 x2) Swap the top cell pairs on the stack
OVER	(x1 x2 -- x1 x2 x1) Put a copy of the second element on the top of the stack
2OVER	(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2) Put a copy of the second set of elements on the top of the stack
ROT	(x1 x2 x3 -- x2 x3 x1) Rotate the top three elements of the stack
2ROT	(x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2) Rotate the top three element pairs of the stack
NIP	(x1 x2 -- x2) Remove the second element from the stack
TUCK	(x1 x2 -- x2 x1 x2) Put a copy of the top stack item below the second stack item
>R	(D: x --) (R: -- x) Transfer the top of stack to the return stack
2>R	(D: x1 x2 --) (R: -- x1 x2) Transfer the top two elements of the stack to the return stack
R>	(D: -- x) (R: x --) Transfer the top of the return stack to data stack
2R>	(D: -- x1 x2) (R: x1 x2 --) Transfer the top two elements of the return stack to data stack
R@	(D: -- x) (R: x -- x) Copy the top of the return stack to data stack
2R@	(D: -- x1 x2) (R: x1 x2 -- x1 x2) Copy the top two elements of the return stack to the data stack
@	(addr -- x) Reads 'x' from the specified RAM address 'addr' and places it on the top of the stack
!	(x addr --) Writes 'x' to the specified RAM address 'addr'
AND	(x1 x2 -- x1&x2) Bitwise AND of the top two elements of the stack
OR	(x1 x2 -- x1 x2) Bitwise OR of the top two elements of the stack
XOR	(x1 x2 -- x1^x2) Bitwise XOR of the top two elements of the stack
NOT	(x -- ~x) Bitwise NOT of the top two element of the stack
<<	(x u -- x<<u) Shifts 'x' 'u' bits to the left
>>	(x u -- x>>u) Shifts 'x' 'u' bits to the right
1+	(x -- x+1) Increment 'x' by a value of 1
1-	(x -- x-1) Decrement 'x' by a value of 1
+	(x1 x2 -- x1+x2) Add x1 to x2
-	(x1 x2 -- x1-x2) Subtract x2 from x1
*	(x1 x2 -- x1*x2) Multiply x1 by x2
/	(x1 x2 -- x1/x2) Divide x1 by x2

```

*/      ( x1 x2 x3 -- x1*x2/x3 ) Multiply x1 by x2, divide the result by x3 -intermediate value is 64 bits
>      ( x1 x2 -- flag ) If x1 is greater-than x2 then flag equals -1 else flag equals 0
<      ( x1 x2 -- flag ) If x1 is less-than x2 then flag equals -1 else flag equals 0
>=     ( x1 x2 -- flag ) If x1 is greater-than or equal to x2 then flag equals -1 else flag equals 0
<=     ( x1 x2 -- flag ) If x1 is less-than or equal to x2 then flag equals -1 else flag equals 0
=      ( x1 x2 -- flag ) If x1 is equal to x2 then flag equals -1 else flag equals 0
<>     ( x1 x2 -- flag ) If x1 is not-equal to x2 then flag equals -1 else flag equals 0
=0     ( x -- flag ) If x is equal to 0 then flag equals -1 else flag equals 0
<>0    ( x -- flag ) If x is not-equal to 0 then flag equals -1 else flag equals 0
>0     ( x -- flag ) If x is greater-than 0 then flag equals -1 else flag equals 0
<0     ( x -- flag ) If x is less-than 0 then flag equals -1 else flag equals 0
PC      ( -- pc ) Put the 'Program Counter' on the top of stack
SP      ( -- sp ) Put the 'Data Stack Pointer' on the top of stack
RP      ( -- rp ) Put the 'Return Stack Pointer' on the top of stack
FLUSH   ( x1 x2 x3... -- ) Flush the data stack of all elements
.       ( x -- ) Print the top of the stack
." ..." ( x -- ) Print the string encased in quotes. Note a space is required after the first quote.
F1+     ( x1 -- x1+1 ) Increment (floating point) x1 by 1
F+      ( x1 x2 -- x1+x2 ) Add (floating point) x1 to x2
F-      ( x1 x2 -- x1-x2 ) Subtract (floating point) x2 from x1
F*      ( x1 x2 -- x1*x2 ) Multiply (floating point) x1 by x2
F/      ( x1 x2 -- x1/x2 ) Divide (floating point) x1 by x2
F*/     ( x1 x2 x3 -- x1*x2/x3 ) Multiply x1 by x2, divide the result by x3 -intermediate value is 64 bits

```

*The test in parenthesis shows the state of the data stack

**Items that start with a leading underscore are internal to xVM and are shown for byte-code debugging purposes.

***Although text is shown in *all* upper case, meusForth will accept *all* lower case – mixed case is not accepted

****After entering code on the command-line hit “enter” to start the interpretation process

Basic examples

Entering:

```

0
OK> 99 100 *

```

Returns:

```

(9900)
OK>

```

Entering:

```

0
OK> 1 2

```

Returns:

```

(1 2)
OK>

```

Entering:

```

(1 2)
OK> swap

```

Returns:

```

(2 1)
OK>

```

Entering:

0

OK> 100 200 300

Returns:

(100 200 300)

OK>

Entering:

(100 200 300)

OK> HEX

Returns:

(0x0064 0x00C8 0x012C)

OK>

Entering:

(900 2 1800)

OK> */

Returns:

(1)

OK>

Word definition examples

Be sure to preface a word definition with a colon and end it with a semicolon. Also, spaces count, e.g. you need to put a space after the colon to define a word and a space before the semicolon to terminate the word definition.

A Celsius to Fahrenheit converter (multiply by 9, then divide by 5, then add 32).

When the CtoF word is called it expects its parameter to already be on the stack.

Enter:

0

OK> : CtoF 9 * 5 / 32 + ;

Enter:

0

OK> 20 CtoF

Returns:

(68)

OK>

Let's examine how the CtoF word works. A value of 20 is pushed on the stack prior to the CtoF call. When CtoF is invoked it pushes 9 on to the stack then performs a multiply on the top two stack items – remember it's RPN therefore $9 * 20 = 180$. Next, the value 5 is pushed on to the stack and a divide operation is performed on the top two stack items, $180 / 5 = 36$. Then 32 is pushed on to the stack and the last operation, an addition, is done, $36 + 32 = 68$. The result of converting 20 degrees Celsius to Fahrenheit is 68.

Lets define a word that converts Fahrenheit to Celsius (minus 32, then multiply by 5, then divide by 9).

When the FtoC word is called it expects its parameter to already be on the stack. Enter:

(68)

OK> : FtoC 32 - 5 * 9 / + ;

Enter:
(68)
OK> FtoC

Returns:
(20)
OK>

Examining the CtoF word. The value (68) from the CtoF conversion was already on the stack so we just invoke our FtoC word. When FtoC is invoked it pushes 32 on to the stack then performs a subtraction on the top two stack items, $68 - 32 = 36$. Next, the value 5 is pushed on to the stack and a multiplication operation is performed on the top two stack items, $36 * 5 = 180$. Then 9 is pushed on to the stack and the last operation, division, is done, $180 / 9 = 20$. The result of converting 68 degrees Fahrenheit to Celsius is 20, which checks out with the CtoF conversion.

Code from file examples

You can run code from within a text file from the command-line. There're two example file included with meusForth, fib.f and dispfibs.f. fib.f calculates Fibonacci numbers and dispfibs.f displays the first 20 Fibonacci numbers. To import the code from the command-line and run the words do the following.

Enter:
0
OK> file fib.f

Returns:
0
OK>

Enter:
0
OK> file fib.f

Returns:
0
OK>

Enter:
0
OK> file fibs_disp.f

Returns:
0
OK>

To display the words we've imported thus far use the 'words' directive:

Enter:
0
OK> words

Returns:
---- Vocabulary context: FORTH ----
fib fibs_disp
Total 2 words, length 46 bytes

```

0
OK>
-----
Enter:
0
OK> 10 fib

Returns:
(55)
OK>
-----
Enter:
0
OK> disp

```

Returns:

```

1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
0
OK>

```

The fibs_disp word prints out the first 20 Fibonacci numbers. We can see from the list that the 10th number is 55, which is the result of our previous fib test.

Note that the user is responsible for the stack, i.e. to be sure that any values that a word requires as parameters are on the stack prior to invoking the word. Stack comments are normally placed in parenthesis just after the name of a new word definition. The left side of a stack comment shows what the word expects to be on the stack and the right side of the comment shows how the stack looks after the word has been invoked. E.g. say we create the following word definition to square a number:

```
: do_sqr ( x -- r ) dup * ;
```

do_sqr expects some value 'x' to be on the stack; this is the number to be squared, then do_sqr makes a duplicate of x on the stack then performs a multiplication with the result being r. It's always good practice to include the stack comments in all word definitions because most Forth's parse the stack comments along with the word's code, then you can view a word's stack comments from the command line at any time. In meusForth you can view the stack

comments of a user defined word by typing help and the name of the word whose stack comments you want to view. Typing “help do_sqr” on the command line would show (x -- r).

There're some files included that test the include-file functionality

Enter:

0

OK> file inc_test.f

File compilation examples

The interactive command-line normally compiles code on a line-by-line basis, with the exception being word definitions that can span multiple lines. Word definitions are unique in that once compiled they're stored in a code buffer, which allows them to be invoked at a latter time. Once you're finished writing all the words that'll make up your program you can compile then to a file for later use. But, before you can compile the code to a file you need to add one last word called “main” which will be used to bootstrap you application.

For example:

Say you have a word that tells the baseband board to toggle its LED's every second. We need a way to call the LED word so we create a main whose sole purpose is to call the LED word. Example:

: LEDS 1 toggle ;

: main LEDS:

If the byte-code is to be compiled as part of a larger program you can use the COMPA directive. E.g.:

(68)

OK> compa test.c

If the byte-code is to be sent to some embedded or remote system as a data message you can use the COMPB directive. E.g.:

(68)

OK> compa test.f

For both these examples the meusForth compiler will write out all existing user defined words to byte-code and write that byte-code to one of the two types of files.

Note also that meusForth will accept compiled byte-code from a binary file on the command-line via the FILEB directive. This has the advantage of by passing the compilation process; the code has already compiled to byte-code. [TBD: more details on the deltas between text file and compiled file. Also, the checksum for array files.]

Here's an example using the fib.f and fibs_disp.f files. Start meusForth then import the two files, for example:

0

OK> file fib.f

0

OK> file fibd_dsip.f

Now lets have a look at the imported word definitions and their stack comments:

0

OK> words

---- Vocabulary context: FORTH ----

fib fibs_disp

Total 2 words, length 47 bytes

0

OK> help fib

Vocabulary: FORTH, word: fib (x --)

0

OK> help fibs_disp

Vocabulary: FORTH, word: fibs_disp (--)

0

OK>

No create a “main” to drive our application:

0

OK> : main (--) 10 fib disp_fibs ;

Note we push 10 on the stack because the fib word requires a value as its parameter. Now we can compile a binary file:

0

OK> compb fib.d

Then we can run the compiled file. Note, you can view the byte-code in fib.d but you’ll need a Hex editor to do so.

0

OK> fileb fib.d

You should see the results for the 10th Fib number along with the print out of the first 20 Fibs.

Items to be added, the list is not all inclusive

- Additional directives and built-in words (primitives), a modulo operator, -ROT, deletion of an already user defined word, etc...
- Foreign Function Interface (aka FFI). For testing, I usually compile my C/C++ code to a dynamic library and use SWIG (Simplified Wrapper Interface Generator) to create Python bindings. I then use Python to exercise the code for testing purposes; it’s a clean and fast way to get large code coverage without a bunch of grungy pre-compilation switches and debug code polluting the actual production code. Unfortunately SWIG can be a bit of a handful when it comes to more complex data types.
- More arithmetic and floating point operators.